ABSTRACT

Title of dissertation: STRUCTANT: A CONTEXT-AWARE TASK MANAGEMENT FRAMEWORK FOR HETEROGENEOUS COMPUTATIONAL ENVIRONMENTS

Andrew Pachulski
Doctor of Philosophy, 2019

Dissertation directed by: Professor Ashok Agrawala
Department of Computer Science

The Internet of things has produced a plethora of devices, systems, and networks able to produce, transmit, and process data at unprecedented rates. These data can have tremendous value for businesses, organizations, and researchers who wish to better serve an audience or understand a topic. Pipelining is a common technique used to automate the scraping, processing, transport, and analytics steps necessary for collecting and utilizing these data.

Each step in a pipeline may have specific physical, virtual, and organizational processing requirements that dictate when the step can run and what machines can run it. Physical processing requirements may include hardware specific computing capabilities such as the presence of Graphics Processing Units (GPUs), memory capacity, and specific CPU instruction sets. Virtual processing requirements may include job precedence, machine architecture, availability of input datasets, runtime libraries, and executable code. Organizational processing

requirements may include encryption standards for data transport and data at rest, physical server security, and monetary budget constraints. Moreover, these processing requirements may have dynamic or temporal properties not known until schedule time.

These processing requirements can greatly impact the ability of organizations to use these data. Despite the popularity of Big Data and cloud computing and the plethora of tools they provide users, organizations still face challenges when attempting to adopt these solutions. These challenges include the need to recreate the pipeline, cryptic configuration parameters, and inability to support rapid deployment and modification for data exploration. Prior work has focused on solutions that apply only to specific steps, platforms, or algorithms in the pipeline, without considering the abundance of information that describes the processing environment and operations.

In this dissertation, we present Structant, a context-aware task management framework and scheduler that helps users manage complex physical, virtual, and organizational processing requirements. Structant models jobs, machines, links, and datasets by storing contextual information for each entity in the Computational Environment. Through inference of this contextual information, Structant creates mappings of jobs to resources that satisfy all relevant processing requirements. As jobs execute, Structant builds models of jobs performance and creates runtime estimates for new jobs based on prior execution traces. Using runtime estimates, Structant can schedule jobs with respect to dynamic and temporal processing requirements.

We present results from three experiments to demonstrate how Structant can aid a user in running both simple and complex pipelines. In our first experiment, we demonstrate how Structant can schedule data collection, processing, and movement with virtual processing requirements to facilitate forward prediction of communities at risk for opioid epidemics. In our second experiment, we demonstrate how Structant can profile operations and obey temporal organizational policies to schedule data movement with fewer preemptions than two naive scheduling algorithms. In our third experiment, we demonstrate how Structant can acquire external contextual information from server room monitors and maintain regulatory compliance of the processing environment by shutting down machines according to a predetermined pipeline.

STRUCTANT: A CONTEXT-AWARE TASK MANAGEMENT FRAMEWORK FOR HETEROGENEOUS COMPUTATIONAL ENVIRONMENTS

by

Andrew James Pachulski

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Advisory Committee:
Dr. Ashok Agrawala, Chair/Advisor
Dr. James Purtilo
Dr. A. Udaya Shankar
Dr. Jay Unick
Dr. Min Wu

Acknowledgments

During my first graduate student orientation, one of the presenters drew a large circle on a chalk board and told us that the circle represented all of human knowledge; he then highlighted a small portion of the circle's perimeter and told us that, during graduate school, the small circle would become our entire world. Looking back, I feel fortunate to have had the opportunity to work on a project that changed the way I view the entire world, rather than blinding me to the rest of it. Reaching this point, however, was the result of guidance and support from a multitude of individuals in my academic, professional, and personal lives.

I would, first and foremost, like to thank my advisor, Dr. Ashok Agrawla, without whom Structant would never have been possible. Dr. Agrawala's unrelenting support of both my research and professional endeavors and the different paths they took during our five years of working together, truly helped guide me toward Structant's successful completion. From my work in toxicovigilance, to Big Data, to Context-Aware scheduling, Dr. Agrawala always encouraged me to look at the big picture and to think about how my work fit in with the rest of the world. In addition, I would like to thank all the members of my preliminary and defense committees: Dr. Unick, Dr. Shankar, Dr. Wu, and Dr. Memon, for their time and feedback; I would especially like to thank Dr. Jim Purtilo not only for serving on my defense committee, but for his guidance when I started working on Structant in his software engineering course.

Over the course of my time as a graduate student, I have had the opportunity to work with some amazing colleagues. In addition to being a great student and colleague, Sarah Bullard played an instrumental role in the Structant project. The organization and attention to detail that she contributed to the early stages of design have remained fundamental to Structant's core, allowing it to far exceed our initial expectations. Nelson Pauda-Perez was not only one of my favorite instructors to teach with, but also made sure that I had always teaching assignment when I needed one. I would also like to thank Jennifer Story, Tom Hurst, and Jodie Gray for their tireless patience in resolving issues I created by never submitting paperwork on time.

In my two short years of working with Bobby, I learned a tremendous amount; he fundamentally shaped the way that I view research, showed me how to present data and information in a way that is accessible to the audience, and taught me to judge my own work against exceptionally high standards. Many of the other members of my first lab including Dave Levin, Matthew Lentz, Ramakrishna Padmanabhan, Aaron Schulman, and Neil Spring, also provided me with an abundance of guidance in my early years of graduate school, for which I will always be thankful. I would also like to thank Faizan Wajid, Mara Cai, and Swati Patel for their feedback and comments on the defense presentation, serving as my practice audiences, and making the MIND Lab an enjoyable place work, And lastly, I would like to recognize Claire Caherty for encouraging me to work on projects that would have a lasting effect on the world.

In my professional life, I am thankful for everyone I have had the privilege of working with at TechNet Enterprises over the last fourteen years. I owe a particular thanks to Zack Pole who went above and beyond the call of duty helping to run TechNet and enabling me to work on Structant over the last six years. More recently, I would like to thank Adam Rosenthal for helping keep TechNet running and our clients happy while I was making my final preparations for the defense. I would also like to recognize the many loyal clients of TechNet who were particularly supportive of my dual life as a business owner and graduate student, especially Dr. Eric K. Morrison.

My academic journey extends far beyond my time spend and relationships formed at the University of Maryland. Dr. Janusz Wnek, in addition to teaching my first Computer Science class in elementary school, started me on an incredible journey that afforded me an abundance opportunities to work on world-changing projects with teams of extraordinarily talented people. I would also like to thank Farah Shirazi, Walter Pappas, Frank Caherty, and Alex Blavat for their support and encouragement over the years. In addition, I would like to extend my gratitude to Nathan Sullivan for the abundance of help he provided answering the obscure programming questions that no one else could, and for never being afraid to tell me why I was approaching a problem the wrong way.

Finally, I would like to thank my parents, James and Mary Pachulski, my siblings Michael and Julia, as well as my grandparents Arthur and Alice Pachulski and Carmel and Elizabeth Fenech for their social and financial support throughout my academic career.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The Internet of things has produced a plethora of devices, services, machines, sensors, and networks that generate data at unprecedented rates [81]. Resource management for data processing has evolved alongside the changing landscape of the data environment. In the beginning, mainframe computers allowed users to perform complex calculations over predefined sets of inputs. Computing resources, such as CPU, memory and storage were shared between all users of the mainframe. Users defined the jobs that they wished to run and submitted them to the mainframe operator. Each job represented a single processing step. Users had little to no control over how or when the jobs ran.

As network connectivity and computing technology advanced, users began interacting with centralized computing resources through telecommunication lines [96]. In this time-sharing model, most of the computational operations executed on centralized resources, with minimal processing done on the user's edge. [1] Each job represented a single user session in which the user interacted with a sequence of processing steps. This model afforded users minimal control over the scheduling of the processing steps on the underlying resources.

[1] terminal client

Processing large amounts of data not only provides key insights, but also carries tremendous monetary value. Enterprise organizations began to outgrow single machine (and even clustered) Computational Environments, requiring processing environments with more powerful machines connected with high-speed links. In response to the size and velocity of data that many organizations process daily, cloud computing has grown in popularity. In cloud computing, cloud providers professionally manage a set of resources and provide users with execution environments or Software as a Service (Saas), billing users based on a predetermined pricing model [48]. Computing resources consist of large collections of shareable servers, interconnected locally through a fast network, to large storage farms. Cloud service providers often provide software, and the user supplies inputs and datasets [32].

As data processing continues to drive business, the need to process large and diverse datasets has trickled down to businesses and organizations of all sizes. Smaller businesses, organizations, and research groups are now processing larger and more diverse datasets than they were ten years ago. These small shops may contain few servers and disk farms, have limited Internet connections, and have limited professional management experience. While cloud computing and Big Data solutions undeniably provide benefits to user, research has begun to recognize that they do not offer a one-size-fits all solution. Despite the popularity of these solutions, organizations still face major obstacles when trying to implement cloud computing and Big Data solutions [71]. These obstacles range

from monetary costs of implementation, to cryptic configuration parameters, to organizational policies.

In this dissertation, we present Structant, a context-aware task management framework for managing and coordinating task execution in complex pipelines in heterogeneous Computational Environments. Structant functions as a real-time active manager / monitor operating at the control layer, while controlling the user's computational elements operating at the processing layer across a collection of hardware resources. Some components of Structant are implemented in each computational element which carry out the processing directives they receive from the control layer and monitor the operations of the computational element sending real-time information to the monitor. The monitor maintains a model of the Computational Environment, the current status of all computational elements along with any relevant contextual information. When a user submits any processing request, he or she also provides meta information about the processing request including the software to run, dataset to use, and disposition of the output. Based on this information, Structant develops a processing schedule for all steps, consistent with all the policy constraints of each processing step as well as those of the site.

1.1 Problem Statement

The challenge of scheduling diverse jobs in heterogeneous Computational Environments can be formally stated as follows:

"Given a set of Computational Operations with Dependencies that must be performed over a series of Datasets across multiple Computational Resources while obeying arbitrarily complex User Policies and Processing Requirements, create an Ordered Mapping of Computational Operations to Computational Resources that satisfies both the Operational Dependencies and User Policies with Reasonable Efficiency while allowing the operator to remain a key Decision Maker."

The underlined terms in the problem statement carry specific meanings for this dissertation and will be explained in the following sections. In addition, these terms will be elaborated on in subsequent chapters.

1.1.1 Computational Operation

In this dissertation, Computational Operations refers to a set of executables that a user wishes to run. Computational Operations can range from basic operations, such as creating a copy of a file, to complex operations, such as ingesting data from sensors deployed in the field and performing filtering and analysis to produce a desired result. We treat nearly all Computational Operations as black

box programs, [2] and make the assumption that Structant and its users have limited knowledge of the executables' internal design. This is common in commercial software where the algorithms and source code may be proprietary and not available to the public. [3] If the user has knowledge of the internal workings of the executables, he or she can encode this knowledge into Structant to enhance performance.

1.1.2 Dependencies

Computational Operations specified by the user may have dependencies on the outputs of other Computational Operations. Because of these dependencies, scheduling a particular operation may not be possible until other operations complete. When using Structant, the user must explicitly call out most dependencies. Structant cannot inspect the operations and automatically build a dependency tree with no input from the user.

1.1.3 Datasets

Datasets refer to collections of bytes stored on a Computational Resource, read by executables specified in Computational Operations. Structant makes minimal use of information about the data, except for simple properties including size and cardinality. Datasets can be specifically called out as a dependency (input) of a Computational Operation or listed as an output produced by a Computational

[2] With the exception of certain file copy operations
[3] Such software is referred to as "Black Box software"

Operation. The term Datasets refers to any information store that may house a user's data. Datasets may include structured data, such as electronic medical records, business forms, financial transactions; unstructured data, such as pictures, videos, audio recordings; and semi-structured data, such as emails and business documents. Datasets may include information stored in any number of formats, including flat files, databases, [4] or proprietary storage systems, such as email archives.

1.1.4 Computational Resources

Computational Resources describe hardware used to perform Computational Operations, transfer datasets, or store results. Computational Resources generally fall into one of two categories: Nodes (servers or machines that can run executable code to perform Computational Operations), and Edges (connections between Nodes that allow the transfer of datasets and executables between Nodes). Computational Resources may belong to different organizational units, meaning that different users may manage the underlying hardware. In this dissertation, we assume that the organization running Structant has full administrative control over the Nodes and does not share the execution environment with another tenant. With the prevalence of hardware supporting virtual machines as well as an abundance of cloud service providers, we assume that any user can obtain a private execution environment. For this reason, we consider a Computational Resource to be an execution environment, meaning that we consider both phys-

[4] such as MySQL or NoSQL

ical and virtual machines Computational Resources. While this is not a strictly necessary assumption for correct operation, certain components, such as execution profiling and benchmarking, may not work as expected in a multi-tenant environment. We assume that multiple users and systems share Edges.

1.1.5 User Policies

User Policies model the rules and abstract processing requirements that a user may wish to impose on Structant. In this dissertation, Structant can model and evaluate any policy that a user can specify in a Turing complete programming language. Structant allows users to specify policy functions in Ruby, and evaluates these policies while scheduling Computational Operations to run on Computational Resources. Users can define and modify persistent state associated with Computational Resources and Computational Operations in order to create policies that reflect real-life organizational goals rather than goals based on a preselected set of attributes, properties, or operations. Through the use of User Policies, a user can create a scheduling policy that dictates *"Only schedule this Computational Operation on this Computational Resource if it's raining in Washington, D.C."* if he or she so wishes.

1.1.6 Ordered Mapping

When Structant loads information from a user about the Computational Operations that he or she wishes to run, it constructs a directed acyclic graph (DAG)

representing the dependency structure of the operations. Structant will only run a Computational Operation if all of its dependencies have completed successfully and the User Policies have been satisfied. Structant will always run the Computational Operations in the correct order and will not violate scheduling policies.

1.1.7 Reasonable Efficiency

Due to the challenges introduced by allowing users to create complex policies that modify persistent state, Structant cannot feasibly examine the entire problem space and find the most efficient Ordered Mapping. User Policies may depend on temporal information (policies may be in effect at certain times), they may limit the number of Computational Operations that may run on a machine, or they may completely impede progress by specifying incompatible policies. Despite this obstacle, Structant can still find efficient Ordered Mappings using *a priori* knowledge of how well Computational Operations performed under previous mappings to similar Computational Resources. Structant will not schedule a Computational Operation on a Computational Resource that it predicts will be unable to complete the operation. Instead, it will pick a mapping of Computational Operations to Computational Resources, given the information available at scheduling time, such that the operation will likely complete successfully.

1.1.8 Decision Maker

Despite Structant's ability to estimate performance and evaluate policies, User Policies interact with Structant's contextual models and with external variables in a way that prohibits Structant from fully exploring execution paths in any meaningful way. In addition, treating Computational Operations as Black Box executables introduces additional complexity to the exploration of execution paths. Because Structant does not know and cannot evaluate the preconditions for the executables, it cannot check whether an executable will require external input from the user. An operation can require external input from the user under certain circumstances. In one situation, Structant may not be able to make progress on a Computational Operation because of a policy restriction. In this case, Structant would require input from the operator in order to decide whether to override the policy, schedule the operation on a different machine, or continue waiting. In another example, a user could have forgotten to remove files from a previous iteration. Upon finding outputs from a previous iteration, one or more executables may prompt the user whether they wish to overwrite the old files or skip writing output. In both cases, Structant keeps track of the current state of the running Computational Operations. If Structant finds that either the executable running as part of the operation has become stuck waiting for input, or if the operation has been repeatedly unable to make progress due to a policy constraint, it will alert the user and ask for intervention.

1.2 Contributions

Structant takes a new, context-aware approach to solving the problem statement. As a result of this new approach, we encountered and solved problems, cases, and scenarios not addressed by prior work. In this dissertation, we present our contributions to the domains of context-aware computing, performance modeling, and scheduling.

1.2.1 Processing Requirements

We exhibit a formalization of processing requirements. By incorporating contextual models of all entities in the Computational Environment, we demonstrate a technique for specifying and evaluating processing requirements. Structant models physical, virtual, and organizational requirements as functions computed over the contextual information of one or more entities. These techniques allows system designers develop systems that separate definition and inference of processing requirements from the application logic.

1.2.2 Dynamic Requirements

We demonstrate a technique for scheduling jobs with dynamic processing requirements. Structant allows users to specify functions that run before, during, and after scheduling to encode and evaluate information associated with context stores. By allowing these functions, Structant provides a way for users to pro-

vide contextual inference functions as inputs to a scheduling algorithm, allowing Structant to schedule based on dynamic and interdependent requirements.

1.2.3 Runtime Estimation

We demonstrate techniques for incorporating relevant contextual information into runtime estimation. By storing a snapshot of each entity's context store at schedule time, we demonstrate techniques that allows scheduling algorithms to perform relevant context selection for runtime estimation. This algorithm extends existing profiling techniques to include comparison of performance across machines described by different sets of attributes. By extending profiling in this fashion, we provide a foundation for incorporating user knowledge into the performance prediction of unseen jobs on unseen hardware, based on historical observations.

1.3 Necessity of Solving This Problem

Designing and implementing a system that satisfies the problem statement has useful implications in both academia and industry. Despite advances in the fields of high performance computing, parallel computing, and Big Data, there still exists a need for a processing framework that considers the contextual information surrounding the data, operations, and Computational Environment. We refer to this problem as the Goldilocks problem.

1.3.1 The Goldilocks Problem

The Goldilocks Problem describes the fact that data processing and analysis has become such a relevant need for business and organizations of all sizes, yet single machine processing solutions are too small and Big Data processing solutions are too big and inflexible. Due to the diversity, speed, and volume of data, there exist data processing pipelines that cannot run on a single machine. Similarly, without well-defined pipelines, domain expertise, enough historical data, deep pockets, and data architects, cloud computing may not be a realistic option for some organizations. The reasons for solving this problem include:

1. Despite the rising popularity of cloud computing, organizations still face challenges when attempting to migrate to cloud computing architectures; these challenges include difficulty in projecting computational costs on the cloud, need to re-architect existing data processing pipelines to run on new platforms, and challenges with moving data to and from the cloud [16, 71].

2. Current processing frameworks are not suited to the increasingly diverse nature of real-world problems [16]. Every dataset, operation, and computation has contextual information [4] surrounding it which may be relevant for scheduling time and location for processing data.

3. Current systems are often too specific to a single problem or computing environment [16]. Big Data processing stacks process specific types of data using specific algorithms. Often, moving to a hosted Computational En-

vironment requires a complete redesign of existing ingress, preprocessing, processing, and analytical steps, which can cost both time and money [84].

4. Despite advances in automatic processing and analysis of data, some aspects of data analytics still require human interaction. Creating a task execution and scheduling framework that allows users to make decisions during pipeline execution provides three major benefits. First, the user can see intermediate results which helps prevent costly mistakes in hosted computing environments [43]. Second, it allows for the incorporation of iterative algorithms into pipelines because users can interact with running instances of a program to decide when to terminate execution of a step [50]. Third, it makes the entire pipeline into a more iterative process by allowing the user to create checkpoints for inspection and adjustment [66].

5. The physical Computation Environment has become more relevant. As organizations continue to store more critical data in electronic format, government and regulatory organizations impose restrictions on the physical properties of the Computational Environments that house this data. The National Credit Union Administration (NCUA) imposes restrictions on the temperature and humidity of server rooms housing member data [7], the Health Insurance Privacy and Portability Act (HIPAA) dictates encryption methods, at rest storage requirements, and data locality requirements [91], Payment Card Industry Data Security Standard (PCI DSS) Compliance also requires Computational Environments to conform to specific standards [34].

Allowing users to model physical properties of the Computational Environment and define organizational policies based on these properties, allows scheduling algorithms to help organizations maintain and document objective compliance with institutional requirements.

Structant enables users to coordinate, manage, and distribute the processing of existing pipelines to process large amounts of data, quickly deploy pipelines running across multiple machines, and maintain organizational and regulatory compliance. Structant utilizes diverse and dynamic contextual information to effectively model the complex nature of real-world problems and computing environments. Structant recognizes and enforces arbitrarily complex policies to enable the user to easily specify which machines should process data, what operations should occur on the data, where the data should reside, when the operations should run, and informs users when any of these policies restrict system progress.

Chapter 2: Approach

In a pipeline, each step may have specific processing requirements that dictate when the step can run, and what machines can run the step. Processing requirements define a set of preconditions that must be satisfied before a machine can execute a job. We generalize processing requirements into three distinct categories: physical, virtual, and organizational.

Physical processing requirements dictate what physical characteristics the underlying hardware of a machine must have in order to process a job successfully. The required characteristics depend on the job that a user wishes to run. If a user wishes to perform graphical modeling or run an algorithm that performs deep learning, the physical processing requirements may include a machine with a Graphics Processing Unit (GPU).

Virtual processing requirements dictate the properties of the execution environment that a job needs for successful execution. Like physical processing requirements, the required characteristics depend on the job that the user wishes to run. Virtual processing requirements may include access to runtime libraries, correct system architecture, and installed dependencies, as well as the completion of antecedent jobs [1].

[1] Also referred to as Task Dependence

Organizational processing requirements encompass the set of rules and regulations that dictate what, when, where, and how jobs can run. These rules can come from owners of the hardware, owners of the data, owners of the software, business policymakers, government and regulatory agencies, or the user of the system. These rules have an effectively unlimited scope. As an example, organizational policies may dictate that medical data must only traverse encrypted links, or that queries of customer data should only occur in server rooms with secure physical access.

Prior work has focused on either a single or limited scope of processing requirements. Structant introduces new techniques that schedules jobs with respect to a wide variety of processing requirements. The Structant processing environment consists of two main components, a Dispatcher application, responsible for modeling policies, data and task dependence, performance metrics, system state, and making scheduling decisions; and one or more Workernode applications, responsible only for receiving work assignments from the Dispatcher, executing the work on the underlying hardware, and providing feedback to the Dispatcher. By storing, updating, and interpreting contextual information, Structant provides task management and scheduling services that can recognize and schedule tasks with respect to complex physical, virtual, and organizational processing requirements.

The Dispatcher application utilizes six virtual components to model state, predict execution performance, enforce polices, and assign work to the Computational Resources. Throughout execution, the Dispatcher maintains a complete

model of every piece of contextual information in the entire Computational Environment. Every scheduling decision made by the Dispatcher application is the result of performing one or more functions over the data stored its models, and adds additional information to the internal models.

One instance of the Workernode application runs on each Computational Resource in the Computational Environment and provides a mechanism for the Structant Dispatcher to interact with the underlying hardware on the Computational Resource. The Workernode application utilizes simplified versions of the six models to track the work assignments it has received from the Dispatcher.

2.1 The Virtual Components

Structant has six virtual components that the Dispatcher and Workernode executables instantiate to represent the states of all entities in the Computational Environment. These components consist of the Ontology, Fabric Model, Policy Broker, Execution Observer, Context Broker, and Scheduler.

2.1.1 Ontology

The first virtual component of Structant is the Ontology. The Ontology serves as Structant's model of the Computational Operations, Datasets, and their Dependencies. The Ontology abstracts the pipeline by modeling every Computational Operation as a series of interrelated and dependent steps. Tasks and Transformations model Computational Operations.

2.1.1.1 Transformations

Transformations model the most basic step in a Computational Operation. A Transformation models a single execution, from start to finish, of an executable piece of software specified by the user. Each Transformation may have one or more Datasets associated with it, specified either as inputs or outputs. An input to a Transformation describes the fact that the Transformation reads in the referenced Dataset, while an output models the fact that the Transformation produces (or modifies) the Dataset. Structant considers Transformations as atomic operations; this means that Structant will not allow dependent Transformations to use the output of a Transformation until the Ontology indicates that the original Transformation has completed successfully.

2.1.1.2 Tasks

Tasks enforce atomicity and locality of a collection of one or more Transformations. Like Transformations, Tasks have an atomic nature because Structant will not dispatch dependent Tasks until the Ontology indicates that the original Task has completed successfully. Structant considers a Task complete only after all its Transformations have completed successfully. Grouping two or more Transformations into a Task enforces Transformation proximity; the Scheduler will schedule all Transformations within a Task on the same set of Computational Resources. A Task may have one or more Datasets associated with it as either inputs or outputs. An input to a Task describes the fact that one or more Trans-

formations in the Task read the referenced Dataset, while an output describes the fact that a Transformation in the Task produces the Dataset as output. The inputs and outputs of a Task do not equate to union of the inputs and outputs of the Transformations within the Task. Structant uses Task inputs and outputs to ensure that the Node scheduled to run the Task has access to the necessary Datasets at runtime. In some cases, Transformations within a Task may transfer data between each other through the use of input and output files; in these cases, the user may choose not to list these files as inputs or outputs to Task.

2.1.2 Fabric Model

The second virtual component of Structant is the Fabric Model. The Fabric Model models Computational Resources as nodes and edges in a graph. Nodes represent machines such as servers, workstations, or virtual machines that can physically run applications. For each Node in the graph, Structant stores contextual information that describes the machine. The context associated with a Node may include anything from physical resources such as CPU count, total memory, memory speed, available storage space, to organizational metadata such as physical location, machine name or cluster id. Edges represent connectivity between Nodes in the graph. An Edge between two Nodes indicates that running instances of the Workernode application on the two machines can communicate directly. Every Edge in the graph has contextual information that describes the connection. The context associated with an Edge may include physical properties

such as bandwidth, latency, loss rate, as well as organizational information such as cost per unit of data or security classifications [60]. The Fabric Model stores Edges in the graph asymmetrically because communication may have different properties depending on the direction. For example, the Google Cloud platform charges users for egress data from Google Cloud but not for ingress data [48].

2.1.3 Policy Broker

The third virtual component of Structant is the Policy Broker. The Policy Broker models, evaluates, and enforces the User Policies associated with every entity in the system. Together, these policies describe the physical, virtual, and organizational processing requirements of every job in the pipeline.

Structant models policies as evaluable relationships between the contextual information of one or more entities in the Computational Environment. Policies can have either local or global scope. Local policies apply to a specific entity, while global policies apply to all entities of a specific type. In order for the Policy Broker to consider a mapping valid, the mapping must satisfy every policy of every entity in the mapping. The versatility of Structant's Policy Broker affords users not only the ability to prohibit undesirable behaviors but also to achieve specific desirable behaviors.

2.1.4 Execution Observer

The fourth virtual component of Structant is the Execution Observer. The Execution Observer maintains historical performance information for all Transformation to Node mappings. The Execution Observer captures context-aware benchmarks throughout execution, and uses these benchmarks to make predictions about future executions under similar contexts. During scheduling, the Execution Observer selects relevant contextual information and makes runtime prediction for new Transformation to Node mappings. The Execution Observer enables Structant to achieve reasonable efficiency. By consulting the Execution Observer to get a runtime estimate for a given mapping, the Scheduler can prevent scheduling a Task or Transformation that it may later have to preempt for policy violation.

2.1.5 Context Broker

The fifth virtual component of Structant is the Context Broker. All virtual components use the Context Broker to either read or write to the contextual information stored in other virtual components. The Context Broker provides a safe way for the user to create functions that describe processing requirements by interacting with the virtual components. Additionally, the Context Broker serves as a critical interface between external context such as environmental sensors, APIs, the user, and Structant's virtual components.

2.1.6 Scheduler

The sixth and final virtual component of Structant is the Scheduler. The Scheduler considers all information in the virtual components, and attempts to find a mapping of Tasks and Transformations to Nodes and Edges that satisfy all processing requirements and promote reasonable efficiency and progress. The Scheduler uses information from the Execution Observer to estimate runtime and consults the Policy Broker to determine whether a given mapping satisfies all relevant policy. The Scheduler takes a greedy approach and attempts to schedule the longest running jobs first on the Node that it predicts will complete the job most quickly.

2.2 The Physical Components

The Structant processing environment consists of two physical components, the Dispatcher and the Workernode. Both the Dispatcher and the Workernode use the virtual components as oracles to infer correct behavior. The Dispatcher application runs on a single machine (usually one not considered a Computational Resource). The Workernode runs on all Computational Resources in the Computational Environment.

2.2.1 Dispatcher

The Dispatcher instantiates all virtual components. All Task assignments in the processing environment occur in the Dispatcher. Using the Fabric Model, the Dispatcher builds a representation of the Computational Environment and con-

tinually updates this model as execution takes place. By issuing directives to the Workernode executables, the Dispatcher coordinates all interactions between Computational Resources in the Computational Environment.

2.2.2 Workernodes

The Workernode client allows the Dispatcher to manage the execution of Tasks on Computational Resources. When a Workernode executable receives a Task assignment from the Dispatcher, it executes the Task on the underlying hardware without any local decision-making. During execution, the Workernode executable periodically reports information back to the Dispatcher. Information reported back to the Dispatcher can include execution time, success or failure of a job, whether a job has become stuck, and the list of jobs running on the underlying hardware.

2.3 Other Components

While developing and testing Structant, we created additional components to aid in its deployment and use.

2.3.1 Scripting Language

For large jobs, it can become tedious for the user to enter all the information required by Structant for the correct specification of Tasks, Transformations, and dependencies. In order to alleviate the burden of specifying pipelines, Structant

includes a lightweight scripting language that can quickly generate Tasks and Transformations from a single file.

2.3.2 GUI

Structant also includes a simple Command Line User Interface to allow the User to monitor Computational Resources. The user can see basic information about the Computational Environment, including connected Workernodes, running Tasks and Transformations, and Transformations blocked either by policy or need for user input.

2.4 System Overview

Figure 2.1 shows a detailed overview of the virtual and physical components of Structant. In this overview, the user interacts with the Dispatcher executable through the user interface; the user interface allows the user to submit jobs, check status, and interact with running processes. The Dispatcher stores a model of the jobs that the user wishes to create and their attached metadata, in the Ontology. Through the Workernode interface, the Dispatcher maintains constant communication with the Workernode executables controlling the execution environments on the Nodes in the Computational Environment in order to assign jobs and receive results and performance metrics. The Fabric Model maintains up-to-date metadata for all Nodes and Edges in the Computational Environment. As jobs complete, the Dispatcher stores performance metrics in the Execution Observer in

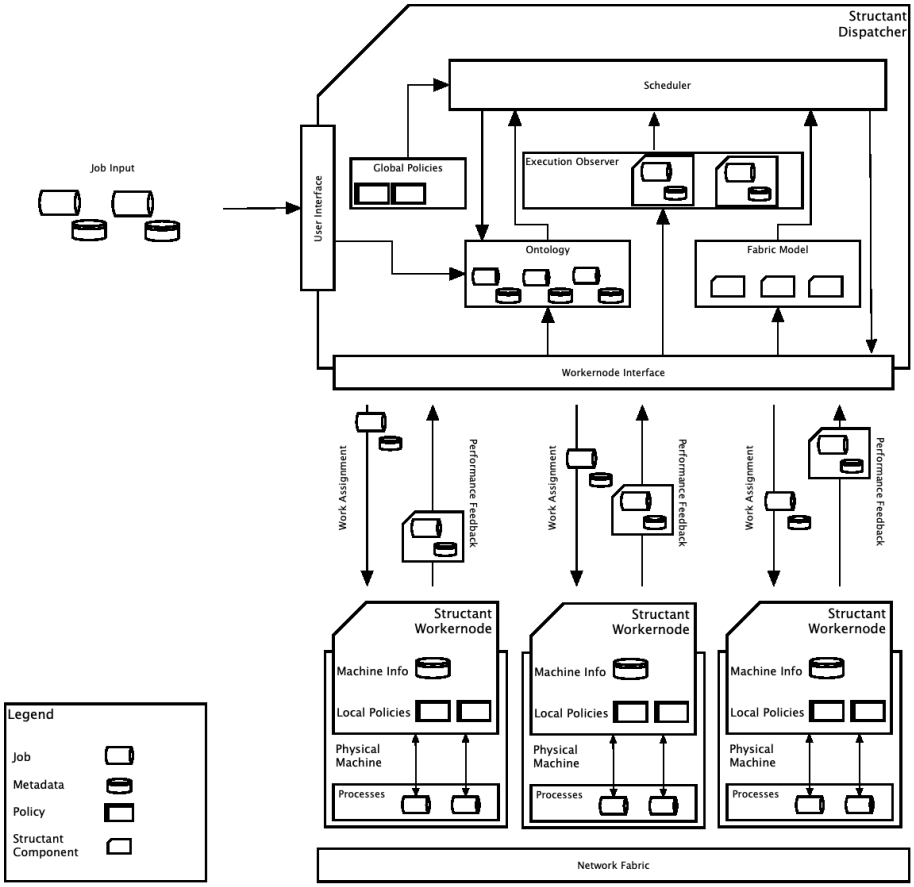


Figure 2.1

order to make future predictions about how jobs from the Ontology will perform when assigned to a particular Node. The Scheduler interprets policies, queries execution history, and examines the Fabric Model, to infer processing requirements and runtime estimates in order to find resources that satisfy processing requirements and can run jobs.

Chapter 3: Related Work

Structant transcends many domains including grid computing, context-aware computing, graph scheduling, execution profiling, and task scheduling. In addition to employing design paradigms from these domains, Structant introduces new challenges into each domain. In this chapter, we discuss related work that applies to the techniques used by Structant, and discuss new challenges that Structant introduces.

3.1 Context Aware Computing

Context aware-computing refers to the practice of encoding information about the operating environment into a system such that the system can automatically detect and respond to its environment while minimizing efforts of the user [51]. Traditionally, context-aware computing has been heavily represented in the pervasive and mobile computing environments [28, 51–54, 62]. Challenges in Context-Aware Computing often stem from the plethora of available contextual information and the number of devices that need to interact. Context refers to the set of attributes and information that describe a particular person, place, or entity relevant to the system. At this point, the research community has established that

context-aware systems not only need the ability to store contextual information, but also to recall and use it at a later point to infer some interpretation of the current environment [52, 54]. Researchers have repeatedly exhibited the use of well-defined data structures as an effective technique for storing contextual information and sharing it across devices. Countless groups have attempted to define standardized ontologies to facilitate sharing of contextual information between context-aware systems, but so far, there does not exist a universally recognized standard.

Researchers have recognized the importance of separating the representation and storage of contextual information from the inference and processing. Most designers of context-aware systems regard allowing the system to directly access sensors or other sources of context directly as poor practice and instead suggest that contextual information exist separately from inference logic. To this day, there exists no universally accepted design paradigm for storing and inferring context. Centralized ontologies such as COBRA-ONT [28] function as context brokers, where agents query a known entity running at a well-known location to receive relevant contextual information about its situation. In other systems [62], each entity maintains its own contextual information and performs contextual inference locally. In the current era of smart-phones and wearable devices, centralized ontologies have risen to popularity as large, cloud-based controllers receive information from and give directions to lightweight thin clients.

3.2 Distributed Computing

The terms grid computing, cloud computing, and distributed computing all carry rather loose meanings. IBM defines grid computing as "the ability, using a set of open standards and protocols, to gain access to applications and data, processing power, storage capacity and a vast array of other computing resources over the Internet. A grid is a parallel and distributed system that enables the sharing, selection, and aggregation of resources distributed across multiple administrative domains based on their (resources) availability, capacity, performance, cost, and users' quality-of-service requirements" [57]. According to Ian Foster, cloud computing is, "a large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet." In this dissertation, we distinguish grid computing, cloud computing, and distributed computing in the following way: computational resources in grid computing are managed by different organization units, shared among multiple tenants, and generally not configurable by the user; computational resources in cloud computing are managed by a single cloud service provider and provide a private and configurable execution environment to the user; distributed computing refers to concept of enabling interdependent tasks to correctly run across multiple machines or cores.

3.3 Graph Scheduling

The goal of task scheduling is to assign tasks to all available processors in a way that satisfies precedence requirements between tasks and minimizes makepan [1]. Studies have proven that the task of finding an optimal schedule is NP-Complete [33, 55]. Graph scheduling refers to the process of creating a directed acyclic graph with nodes representing computational operations and edges representing dependency and communication between operations, and mapping the operations onto a set of computational resources. Widespread use of Big Data processing stacks, such as Hadoop [45], Tez [87], and M3R [92] have given rise to an abundance of work on the topic of scheduling tasks represented in directed acyclic graphs. The simple min-min heuristic, used to schedule tasks without dependencies, selects the shortest running task and maps it onto the computational resource with the minimal completion time [56]; proposed modifications enable min-min to work on groups of tasks with dependencies. Chaining [40] overcomes the limitations of list-scheduling algorithms by allowing the scheduling algorithm to consider non-ready tasks; chaining algorithms split the task graph among the processors, without duplication. Genetic algorithms encode the scheduling problem as a genome, with start times and sequence represented as genes; the algorithm removes infeasible solutions, and creates children by crossing features from the remaining solutions and small amounts of randomness. Genetic algorithms model natural selection and expect each successive generation to have bet-

[1] Overall runtime required to execute tasks

ter fitness (i.e., lower makespan). The Artificial Intelligence based best-first search algorithm A* has been used for scheduling, but does not account for communication delays. Simulated annealing [42] is a Monte Carlo algorithm for scheduling optimization functions; Tabu [82, 90] is a neighborhood search algorithm which tries to find a global minimum by avoiding local minima; Highest Level First with Estimated Times (HLFET) [5] is a list scheduling algorithm which assigns priority to tasks based on the longest path from each node to an exit node (i.e., the total number of generations below the task) [68] (but does not consider the communication costs). Several groups have utilized task duplication in scheduling, [11, 21, 79, 86] which enables certain tasks with large numbers of dependents to be duplicated and run on multiple resources so that output is available to downstream tasks more quickly. The computational costs associated with task duplication has traditionally limited its practicality [10, 14, 36, 63, 100, 106].

Many of the high-quality MapReduce job schedulers focus only on how to schedule collections of independent jobs [59, 88, 105, 107], while others provide job dependency analysis [9, 26, 27, 45, 58, 78, 98]. Recent work, such as PISCES [29], has focused on the movement of data between dependent jobs, though limited in scope to MapReduce.

3.4 Runtime Estimation

Accurate runtime prediction is an important part of task scheduling. Many estimating, modeling, and simulation techniques have been developed to model the

performance of distributed and parallel applications [15, 19, 31, 73, 76, 101]. We categorize the problem of runtime prediction into two distinct categories: simulation and modeling.

3.4.1 Performance Simulation

When given a dataset and a collection of operations to execute over the dataset, one technique used to predict runtime is to create a simulator that models the different computational operations, then run a simulation of the desired computation. Performance simulation more often appears in system design than in application scheduling. Although many corporations have published their approaches and techniques for Big Data architecture and processing [93, 95], the platforms have remained convoluted, and often require skill and labor intensive implementations [13]. Simulators such as BigDataNetSim [13], dagSIM [61], and cloudSim [24] have proven to be effective in modeling and predicting job runtime, and have shown to be useful for performance and parameter tuning. BigDataNetSim has extended the simulation model to include the underlying network fabric.

3.4.2 Performance Modeling

Another technique used for runtime prediction is performance and resource modeling. This technique combines traces of prior executions with known properties of the algorithms, resource requirements, and underlying hardware to predict the runtime of new jobs. Prior execution traces can either come from full executions

of programs in an isolated environment [74] or running on a sample the input space [8, 46, 70].

3.5 Seemingly Similar Applications

Due to the broad nature of the problem statement (see Section 1.1), there are several applications similar to Structant. We discuss their similarities and difference below.

- **NimRod/G**, *1999* [22]: NimRod/G is a distributed processing system for scheduling parametrized jobs on grid computing resources. NimRod's scheduler assumes a grid computing environment where resources are under different administrative domains, and the user pays each organization based on the amount of computational resources utilized for a particular job. The scheduler uses resource information, cost per computational resources, and user deadlines to schedule jobs on computational resources. NimRod facilitates an auction for resources to schedule with minimal monetary cost to the user for scheduling long-running jobs. NimRod uses a predefined set of resources (CPU frequency, memory amount, memory speed) for scheduling purposes, and allows the user to specify how much of each resource type a job takes.

  Structant employs a much more robust model of the Computational Environment to facilitate far more advanced scheduling. Resources and attributes in Structant are dynamic, meaning Structant can infer meaning from, and schedule based on, more complex processing requirements. Structant automatically

observes, models, and predicts the performance of jobs based on input size, output size, prior contextual information. NimRod makes little use of data dependency and focuses on embarrassingly parallel jobs. Policies in Structant are far more fine grained; NimRod allows users to set a deadline for jobs, while Structant allows policies based on dynamic contextual information, user supplied functions, and external context. Structant automatically selects relevant context and uses prior execution observation and forward policy exploration to determine if a mapping will satisfy relevant policy.

- **StarPU**, *2010* [17]: StarPU is a high performance computing platform for scheduling tasks on heterogeneous multicore architectures. StarPU is a runtime system that provides unified access to underlying resources to allow numerical kernel designers to generate parallel tasks and develop scheduling algorithms across heterogeneous systems. StarPU provides unified access to underlying CPU and GPU resources and abstracts much of the underlying architecture, allowing for greater portability.

  While Structant also schedules operations on heterogeneous hardware, Structant facilitates no memory or resources sharing between machines. Structant provides no libraries to machines, and is able to run user supplied operations in a distributed fashion.

- **Resource-Aware Task Scheduling**, *2015* [74]: The work described in this paper use techniques similar to Structant's execution modeling. One major different between the two techniques is their use of the performance model. In this

work, the authors attempt to model each task's sensitivity to a specific resource and then schedule tasks to minimize contention for resources. Structant's performance modeling attempts to model the performance of the task as a function of the attributes specified on the task and on the computational resource based on prior execution observations.

- **Facebook BISTRO**, *2015* [47]: After recognizing shortcomings in the ability of BigData processing stacks to communicate and schedule complex jobs, Facebook engineers wrote custom schedulers to coordinate jobs in BigData processing environments. They combined a collection of schedulers and release them to the open source community as Bistro. While Bistro recognizing the problems that Structant addresses, Bistro has far less support for dynamic policies, no contextual models of machines or jobs, limited resource consideration, limited ability to run in diverse computational environments, and no job profiling. Structant provides a superset of the Bistro's features, implementing most of the features Facebook suggested should be added to Bistro, as well as a collection of new features.

3.6 New Challenges

By introducing arbitrary contextual information, user policies, and applications, Structant introduces design challenges not considered by the prior work. One of the most powerful features of Structant comes from its ability to combine rich contextual information about the Computational Environment with rich contex-

tual information about the Computational Operations. Through the introduction of this additional contextual information, scheduling and modeling both become more difficult. The modeling techniques described in the related work all required some known set of properties about the tasks and the resources (e.g., number of cores, CPU Clock frequency, bandwidth). Structant allows the addition and modification of contextual information during execution. Similarly, none of the scheduling algorithms discussed can handle the complex policies and processing requirements that Structant models.

Chapter 4: Illustrative Problems

In this dissertation, we present three scenarios to highlight how Structant can help a user facilitate running pipelines with complex processing requirements. These scenarios illustrate real-life scheduling problems that Structant solves using acquisition, modeling, and inference of contextual information. They highlight how Structant's context modeling, profiling, and user policies allow its Scheduler to solve a variety of problems across multiple domains, from pipeline scheduling, to link utilization, to controlling the physical environment of a server room.

4.1 Opioid Data Processing

In recent years, America's Opioid epidemic has been a subject of national interest, starting after the FDA approved Oxycontin in 2010 [6]. We began development of Structant to address a shortcoming in available processing solutions that we discovered while working on data mining techniques for this problem. In our prior work, we investigated the possibility of using data mining, modeling, and machine learning techniques to predict cities in the United States that were likely to have steep increases in opioid related deaths. In order to build a more robust model than other groups at the time, our work combined data points from mul-

tiple, disparate data sources, and included modern machine learning and processing techniques. One of the challenges that arose during this experiment was the difficulty in creating and running the pipeline to ingest and process the data. The data came from a variety of sources, contained inconsistent formatting, and required calls to external APIs. Complex processing requirements greatly slowed the progress of the work, as job, task, and machine dependencies required user interaction. In this scenario, we demonstrate how Structant can help the user manage the pipeline by scheduling operations with respect to machine and data dependence as well as virtual processing requirements.

4.2 Off-Site Data Copy

In this scenario, we demonstrate how Structant's User Policies and Execution Observer work together to schedule Tasks with respect to temporal organizational policies, by making forward predictions of execution time. We conducted this experiment at a local small business in response to a real-life problem that they wished to solve. The business needed to perform an off-site backup to maintain compliance with corporate data retention policies. However, the only Internet connection available to the business offered considerably limited upload bandwidth. Because of the limited bandwidth, running the backup interfered with the ability of users working remotely to access files on the on-premise file server, so the backup could only run during non-business hours. The business had numerous files that took an hour or more to back up. Since employees could mod-

ify the files during the day, we could not pause and resume file transfers. The ability of the users to work remotely required termination of any operations still in progress when the business opened at 08:00. In this experiment, we demonstrate that by encoding a small amount of contextual information, Structant can infer the processing requirements and make appropriate scheduling decisions that avoid preemption.

4.3 External Context

In addition to manual user input and execution observation, Structant can also acquire contextual information about the Computation Environment from sources such as external sensors and APIs. In this scenario, we show how Structant's robust contextual models and policies can help satisfy organizational processing requirements. We present the problem of monitoring and preserving safe server room operating temperatures.

In this experiment, Structant acquires contextual information about the Computational Environment Structant using data from a Raspberry Pi connected to five temperature probes. Using this information, Structant schedules a predetermined shutdown pipeline based on industry-standard regulatory compliance; Structant repeatedly queries the Raspberry Pi, updates its contextual model, and checks if the shutdown pipeline should begin. We demonstrate how Structant can automatically schedule the appropriate corrective actions to stabilize the temperature in the server room.

Chapter 5: Using Context

Context-awareness has consistently remained a popular area of researcher interest, and numerous groups have published papers on the topic of Context-Aware Systems. Despite extensive prior work, researchers still disagree on key definitions. While disagreeing on exact definitions, most researchers generally agree on the desired behaviors and properties of well-designed context-aware systems.

In this dissertation, we adopt a modified version of Dey's 2001 definition of context; "any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including location, time, activities, and the preferences of each entity" [39]. Structant uses contextual information to model the Computational Environment and make scheduling decisions. In Structant, entities are Tasks, Transformations, Nodes, and Edges. Each virtual component of Structant models different pieces of the contextual information, that together describe Structant's Computational Environment. The Ontology models the work that the user would like Structant to execute, the Policy Broker models the rules and constraints associated with the work and resources, the Fabric Model models the physical resources available to complete the work, the

Execution Observer models prior results of executing work on resources. Figure 5.1 illustrates the flow of contextual information between Structant's virtual components. Table 5.1 denotes where Structant stores contextual information for each entity.

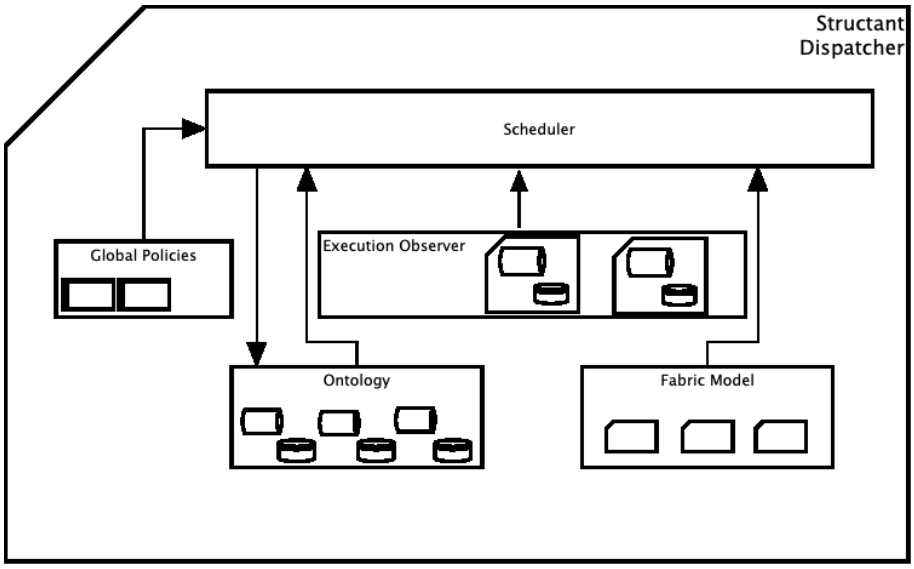


Figure 5.1: The flow of contextual information between the virtual components in Structant's dispatcher.

In addition to storing contextual information, Structant must extract, interpret, and use stored contextual information to automatically adapt its behavior to the current context of the Computational Environment [20, 23, 39, 108]. By our definition of context [39], each entity in the Computational Environment has an effectively infinite number of properties that can describe it. Every machine has a multitude of properties, such as number of cores, amount of physical memory, serial number, physical location, but also much more specific properties, such

EntityType	Component	Context Source	Models
"TRANS."	Ontology	Pipeline Description, User Policies/Functions	An executable piece of code a user wishes to run
"TASK"	Ontology	Pipeline Description, User Policies/Functions	Collection of distinct executable codes a user wishes to run
"NODE"	Fabric Model	Workernode Config. File, User Policies/Functions	A single computational resource in the Computation Environment
"EDGE"	Fabric Model	Dispatcher Config. File, User Policies/Functions	A single computational resource in the Computation Environment

Table 5.1: Each of Structant's virtual components stores context to model different parts of the Computational Environment. **Note**: "TRANS." = "TRANSFORMATION"

as the manufacture date, the physical weight of the machine, the major and minor number of the network card driver. Similarly, physical connections have an abundance of properties, including physical medium, length, ownership, date installed. Contextual information generally has imperfections, alternate representations, interrelations, and a range of temporal characteristics [53]. Properties of an entity may be challenging to observe. A property such as "date of next hard disk failure" could describe a machine, but such a property may be nearly impossible to observe in practice. Due to the infinitely large number of and difficulty in observing all of an entity's properties, Structant, like other context-aware systems, models only a selected subset of each entity's context. An entity's contextual model refers to the set of named properties and associated values that Structant has stored for the particular entity.

Context-aware applications use these models to facilitate complex queries over multiple types of contextual information, decouple the details of the model

from the application logic, and provide a common framework for representing context derived from different sources [52]. Each component in Structant derives contextual information from different sources, and the Scheduler interprets this contextual information to make informed scheduling decisions. By scheduling based on interpretation of current contextual models, Structant can separate the details of the Computational Environment from the scheduling logic, allowing the user to rapidly adapt Structant to changes in the Computational Environment.

Structant's components store a limited amount of contextual information about each entity in the Computational Environment. Structant stores two types of contextual information for every entity: System Context and User Context. System Context is maintained internally by Structant's virtual components and used primarily for enforcing operational correctness, whereas User Context is maintained by the user through PreCompute, PostCompute, and Evaluate functions.

5.1 Contextual Inference

Structant stores contextual information as a collection of (key $\rightarrow$ value) pairs, where the key identifies the named property and the value represents a piece of data associated with the property. In this dissertation, we refer to the key as a Contextual Identifier, the value as a Contextual Value, and the key value pair as a piece of context; we refer to a collection of key value pairs as contextual information. Without inference, the contextual information is essentially meaning-

less [4]; for example, ("TransformationType" $\rightarrow$ "SCP1") has no intrinsic meaning attached to it. Structant's components compute functions over the contextual models stored in one or more components in order to infer meaning from the context. As an example, consider the local policy shown in Table 5.2.

EntityType	Operator	Values	ContextLabel	FailOnMissing	Time
"TRANSFORMATION"	$\neq$	["SCP1"]	"TransformationType"	TRUE	ALL

Table 5.2: This local policy prohibits the Node to which it is attached from running Transformations with the piece of context ("TransformationType" $\rightarrow$ "SCP1")

During scheduling, the Scheduler considers a mapping and consults the Policy Broker to see if policy allows the mapping. The Policy Broker infers that the Node is ineligible to run the Transformation due to policy constraints (see Section 10.2). The Scheduler infers from the Policy Broker's response that the user does not want the Transformation run on the Node.

Another example of contextual inference occurs during Execution Observation. When the Execution Observer queries its database of performance metrics for previous mappings, it uses the Contextual Identifiers to quantify the similarity between two mappings in order to judge how well one will likely predict the performance of the other (see Section 9.2). Table 5.3 shows the types of context inference done by each of Structant's virtual components.

Component	Contains	Infers
Ontology	Tasks, Transformations	The execution ordering and correctness
Execution Observer	(Transformations → Node: Runtime)	Runtime of a (Transformation → Node) mapping, based on prior observations
Policy Broker	Policies	Correct behavior with respect to policy constraints
Fabric Model	Nodes, Edges	Current state of Nodes, Edges in the Computational Environment
Scheduler	(Transformations → Node: Runtime)	Which Nodes and Edges a Task or Transformation should be mapped onto

Table 5.3: Each of Structant's virtual components infers different information about the Computational Environment from the context storage.

5.2 Sources of Context

Structant's virtual components acquire contextual information from multiple sources. During setup, the user may specify values for certain pieces of System Context. Tables A.1, A.2, A.3 and A.4 denote the System Context that the user can specify in configuration files during setup.

During execution, the user may add, remove, and search User Context for an entity via User Policy functions (see Section 8.2.1). During User Policy evaluation, the Scheduler passes the User Policy function a copy of the Global Context API (see Section 5.3); the user's function can use this API to search and modify contextual models. After the Dispatcher instantiates a virtual model for an entity, the user cannot modify its System Context since the Global Context Broker does not provide write methods to System Context stores.

5.2.1 Predefined System Context

Structant's virtual components store information about the entities they model in the System Context store. The Scheduler, Fabric Model, and Ontology all work by accessing contextual information associated with the entities they model, and taking action based on interpretation of the relevant contextual information. This paradigm of separating the contextual information from the application logic [52], helps facilitate changes to Structant's design since Structant's internal logic can be easily modified by simply changing Contextual Identifiers or the interpretation logic. Tables A.1, A.2, A.3, and A.4 describe the System Context that Structant's virtual components store for each entity.

5.2.2 User Context

User Context enables the user to encode his or her knowledge of the Computation Environment by creating, storing, and modifying mappings of Contextual Identifiers to Contextual Values. User Context is used primarily in the Execution Observer and the Policy Broker. Structant's Execution Observer employs User Context during relevant context selection to determine the similarities between two mappings. The Policy Broker uses contextual information from User Context stores when evaluating Local, Global, and User Policies.

By specifying User Context, the user can create distinctions between two entities that would otherwise appear similar. Providing correct and relevant User Context has two primary benefits. First, relevant information helps the Execution

Observer make more accurate runtime estimates. If a user knows that one machine has X number of CPU cores, and another machine has X' number of CPU cores, encoding this knowledge in the Fabric Model's contextual model of each machine will help the Execution Observer find CPU sensitive Transformations and will help the Scheduler pick the correct Workernode for a Task. Second, relevant information allows Structant to recognize complex and dynamic processing requirements. If a user knows that a particular Transformation requires GPU capabilities, encoding this information in the User Context store of the Ontology's contextual model of the Transformation, creating appropriate User Context in the Fabric Model, and defining appropriate policies will help Structant's Scheduler select a mapping that will allow the Transformation to run successfully.

5.2.3 External and Environmental Context

In addition to contextual information entered by the user before execution and contextual information generated by executing Tasks and Transformations on different Nodes, Structant can also obtain contextual information from external sources. Previously, we introduced an example scheduling policy that suggested Structant could model and implement the user policy that a Transformation should only run if it is raining in Washington, D.C. By querying external APIs and data sources, Structant can obtain and model contextual information surrounding any Computational Resource. The ability to model and evaluate external contextual information greatly expands the utility of Structant as it gives

the user the ability to make Structant schedule Tasks with respect to external inputs, such as weather, environmental, financial, or social information. Using contextual information from external sources allows the user to schedule operations that execute when certain external conditions or events occur.

5.3 Global Context Broker

The PreCompute, PostCompute, and Evaluate functions (see Section 8.2) allow the user to interact with the System and User Context stores through function pointers. In order for contextual information to remain useful to Structant, it must be accessed, modified, and stored in a structured and consistent way [51]. Allowing users to enter and execute code can have dangerous and undesired consequences. Directly exposing the System Context stores could result in undesired operation or system instability if a user function incorrectly modified the store. All of Structant's virtual components rely on certain assumptions about the information in the System Context stores. For example, the Ontology assumes that the Contextual Value associated with the Contextual Identifier "PRECENDENTS" in a Transformation entity is an array of Strings corresponding to the TransformationIDs of Transformations that must successfully complete before the current Transformation can run. If a User Function incorrectly modifies the array associated with the "PRECEDENTS" by adding another String, the Ontology will treat that String as a TransformationID corresponding to an additional Transformation that must successfully complete before the current Transformation can run.

If no Transformation with a TransformationID corresponding to the new String that the user entered into the Ontology exists, the Ontology will never allow the Scheduler to map the Transformation to a Node since the Ontology will infer that the Transformation has unmet dependencies. Another issue arises if the user incorrectly replaces the array with an object of another type. At Scheduling time, the Ontology will attempt to access the value pointed to by the "PRECEDENTS" key and then treat that object as an array of Strings. If an array of Strings no longer exists at the memory location, the Ontology could become unstable and crash. While likely possible to implement the virtual components in a way that checks all values from the System Context store before use, this introduces an additional layer of complexity and makes the system more difficult to design and more resource intensive to execute. Additionally, this would not solve the problem of the user being able to *free()* [64] memory associated with the Context store.

In order to limit the amount of interaction that user functions can have with the User and System Context stores, Structant enforces the following rules:

- **The System Context store is ready only**: User functions cannot modify the System Context store as doing so could result in undesired or unstable operation. If the user needs to modify an exposed value in the System Context store, the user can create a copy in the User Context store and modify the copy.

- **Portions of the System Context store are not accessible**: The System Context store contains pointers to memory locations associated with other entities in the system. A user function could deference these pointers and directly access

and modify System Context stores associated with other entities, violating the principle that "the System Context store is read only."

- **The user cannot directly reference the memory associated with the User Context store or System Context store**: If user functions can directly access the memory associated with the User Context store or the System Context store, then the user function can modify either of them directly. Accessing the memory associated with the System Context store would violate the principle that "the System Context store is read only." Accessing the memory associated with the User Context store would allow user functions to put Structant in an unstable state. If the user damaged the User Context store (for example by *free()*ing [64] it), then the Dispatcher could become unstable if it tried to access the memory location.

5.3.1 Global Context API

The Global Context Broker provides consistent and correct access to Context stores in each virtual component through the use of the System Context API and the User Context API. The System Context API provides read access to certain Contextual Identifiers and associated Contextual Values in the System Context stores. The User Context API provides read and write access to the Contextual Identifiers and associated Contextual Values in the User Context stores in Structant's virtual components. The Global Context Broker provides a simplified API that facilitates correct access to both the User and System Context API. Figure 5.2

shows how the Global Context Broker facilitates access to contextual information storage.

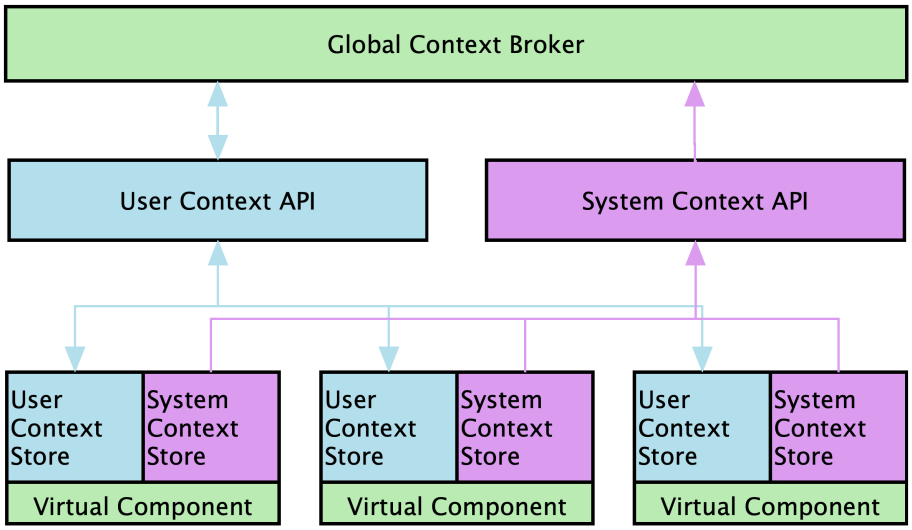


Figure 5.2: The Global Context Broker interacts with the User Context API and the System Context APIs to facilitate correct access to contextual information.

The Global Context Broker creates a separate User Context and Global Context API object for each PreCompute, PostCompute and Evaluate function call. After each function call, the Global Context Broker immediately destroys the API object. This behavior protects system integrity by not allowing Structant to rely on the user not corrupting the API object (or *free()*ing [64] it).

System Context API

The System Context API provides the following functionality:

- **SELECT_SYSTEM_CONTEXT_HAS(EntityType, CIdentifier)**: This function returns a list of Strings containing the entity identifiers for entities of type EntityType which have a value in their System Context store for the label identified

by CIdentifier. The entity type is used to select the type of entity that should be queried; EntityType can be any one of: “TRANSFORMATION”, “TASK”, “NODE”, “LINK”.

- **SYSTEM_CONTEXT_GET(EntityType, EntityID, CIdentifier)**: This function returns a deep copy [1] of the Contextual Value identified by CIdentifier stored in the System Context of the entity of type EntityType identified by EntityID.

User Context API

The User Context API provides the following functionality:

- **SELECT_USER_CONTEXT_HAS(EntityType, CIdentifier)**: This function returns a list of Strings corresponding to the identifiers for entities of type EntityType which have a value in their user Context store for the label identified by CIdentifier. EntityType can be any one of: “TRANSFORMATION”, “TASK”, “NODE”, “LINK”.

- **USER_CONTEXT_GET(EntityType, EntityID, CIdentifier)**: This function returns a shallow copy [2] of the Contextual Value identified by CIdentifier stored in the User Context of the entity of type EntityType identified by EntityID.

- **USER_CONTEXT_CREATE(EntityType, EntityID, CIdentifier)**: This function allocates memory for the Context Identifier identified by CIdentifer in the User Context store of the entity of type EntityType identified by EntityID.

[1] A deep copy copies all fields, and dynamically allocated memory pointed to by the fields.
[2] A shallow copy references the same dynamically allocated memory.

- **USER_CONTEXT_STORE(EntityType, EntityID, CIdentifer, CValue)**: This function associates the object identified by CValue as the Contextual Value associated with the Contextual Identifier CIdentifier in the User Context store of the entity of type EntityType identified by EntityID.

Global Context API

In addition to the functions provided by the User Context API and the System Context API, the Global Context Broker provides the following additional functionality:

- **SELECT_ENTITIES(EntityType)**: This function returns a list of Strings corresponding to the identifiers for entities of type EntityType. EntityType can be any one of: "TRANSFORMATION", "TASK", "NODE", "LINK".

- **GET_MAPPING_INFO(Entity)** This function returns the entity identifier of the entity in the system identified by the role Entity. Entity can be one of "NODE1", "NODE2", "LINK", "TASK", "TRANSFORMATION"

5.4 Example: Defining a Cluster

The following example illustrates how a user may define a model of the Computational Environment by creating appropriate User Context. Given a collection of machines spread across disparate geographic locations (see Figure 5.3), a user may wish to model the fact that certain subsets of the machines belong to distinct clusters. In order to model the clusters, the user can create a Contextual Identifier

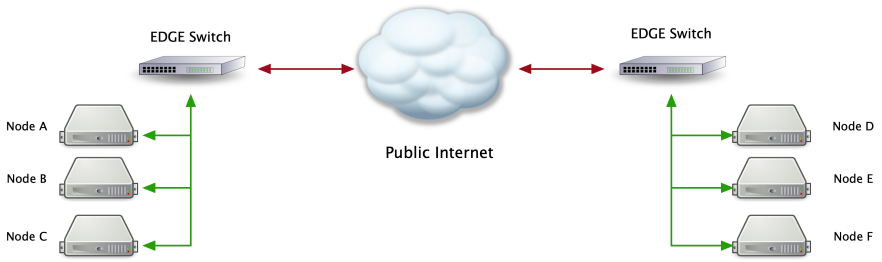


Figure 5.3: Two clusters connected via the WAN.

in the User Context store of each Node [3]. The User Context store for each Node in the cluster should have a Contextual Identifier that maps to the same value. For example, the user may elect to create the Contextual Identifier "ClusterID". Nodes describing machines in the first cluster may have contextual information in their User Context stores such as ("ClusterID" → "Cluster1"); Nodes describing machines in the second cluster may have the contextual information in their User Context stores such as ("ClusterID" → "Cluster2"). The user could repeat this process for each remaining cluster. The User and System Context store in the Fabric Model may look similar to the following:

```
1
2  #System Context store
3  SC_Node[A] = {"NodeID":"NodeA", ... }
4  SC_Node[B] = {"NodeID":"NodeB", ... }
5  SC_Node[C] = {"NodeID":"NodeC", ... }
6  SC_Node[D] = {"NodeID":"NodeD", ... }
7  SC_Node[E] = {"NodeID":"NodeE", ... }
8  SC_Node[F] = {"NodeID":"NodeF", ... }
9
10 # User Context store (Cluster 1)
11 UC_Node[A] = {"ClusterID":"Cluster1"}
12 UC_Node[B] = {"ClusterID":"Cluster1"}
13 UC_Node[C] = {"ClusterID":"Cluster1"}
```

[3]Structant stores Node Context in the Fabric Model

```
14
15 # User Context store (Cluster 2)
16 UC_Node[D] = {"ClusterID":"Cluster2"}
17 UC_Node[E] = {"ClusterID":"Cluster2"}
18 UC_Node[F] = {"ClusterID":"Cluster2"}
```

In order for the Execution Observer and Scheduler to derive useful information from this contextual information, the user must not only define what it means to be a cluster, but also what desirable scheduling and data flow properties exist for clusters in a Computational Environment. For this example, clusters may have the following properties:

- **Identical Configuration**: Machines in the clusters of this example have identical configurations. They have the same storage, CPU capacity, memory capacity, and software.
- **High Bandwidth**: Nodes belonging to the same cluster have high bandwidth, low latency connections to all another Nodes in the cluster. They can transfer large files rapidly with low communication overhead.
- **Low Latency**: Machines belonging to the same cluster reside in close physical proximity to each other, so the connections between them have low latency.
- **Organizational Policy** Machines belonging to the same cluster have identical organizational policies and administrative staff.

Given these properties of a cluster, the user must next define the scheduling and data flow behavior that these properties should induce. As an example, the user may wish to achieve execution consistency. Suppose a user conducting a perfor-

mance study wishes to run a collection of Tasks to benchmark results. The user does not care which cluster the Tasks run on, but only that all Tasks run on the same cluster. The user can easily achieve this with a User Policy (see Section 8.2). At a high level, the user creates a Contextual Identifier in all the User Context stores of all Tasks that need to run on the same cluster that maps to the same Contextual Value. For example, ("TaskGroup" $\rightarrow$ "Task1"). Once a Task with context ("TaskGroup" $\rightarrow$ "Task1") has run on a Node in a cluster, the PostCompute function (see Section 8.2.1) enters contextual information in an appropriate User Context store [4] indicating that a Node in the first cluster ran the Task. Next, the user attaches User Policy to subsequent Tasks that enforces the rule that Tasks with contextual information ("TaskGroup" $\rightarrow$ "Task1") must run on a machine with the same Cluster ID as the first Task.

5.5 External Context for Distributed Management

External Context can model physical properties of the Computing Environment. Structant can obtain External Context from APIs, sensors, external programs, or any source of information that a User Function can query. One popular example of External Context in a Computational Environment is ambient server room temperature. Servers have certain ambient temperature ranges specified by their manufacturers that specify the range of operating temperatures under which the hardware can operate without sustaining damage. For financial reasons, small

[4] There are multiple correct ways of specifying this behavior. The user could store the information in a Task or Node User Context store.

business may run servers in rooms without dedicated cooling. To enforce Quality of Service, users may wish to schedule certain jobs only during off-peak hours such as at night or on weekends. This desire may be problematic as running massive collections of jobs in Computational Environments with numerous machines may raise the ambient temperature in the server room to a point that could trigger automatic shutdown or cause hardware damage. By incorporating contextual information from external sensors, Structant can enforce policies that a job should only run if the server room has an ambient temperature below a certain threshold and preemption should only occur if the ambient temperature rises above a different threshold.

Structant provides a simple mechanism for defining these types of organizational policies. A user can define a Task with two Transformations. The first Transformation contains a NO-OP operation [5] with a single policy attached to it. The policy enforces that the Scheduler can only assign the Transformation if the server room has an ambient temperature below the lower threshold for job scheduling. The second Transformation contains the desired operation, and contains (at a minimum) a policy that states the job can only run if the server room has an ambient temperature below the upper threshold. Figure 15.2 illustrates the Tasks in this pipeline.

Users should exercise caution when defining multiple jobs of this type. If the Scheduler assigns two Tasks, the heat generated between the two machines may cause the preemption of both jobs. As a simple solution, the user could

[5] An operation that takes no input, produces no output, and performs no computation

define a Task with a single Transformation between the two computations with a sleep for a certain amount of time to allow the ambient temperature to stabilize. Additionally, the user could put the sleep Transformation as an antecedent to the second Transformation, instead of the NO-OP, and define a policy that it should only run if the ambient temperature remains below the lower threshold. Since Structant will try to reschedule failed jobs, this will force Structant to wait for the amount of time specified in the sleep for the temperature to stabilize before trying to start a second job.

Chapter 6: Ontology

The Ontology contains Structant's abstract representation of the Computational Operations that the user would like to execute within the Computational Environment. Structant models every Computational Operation and its dependencies in the Ontology. Structant's Ontology stores Computational Operations as a series of interconnected Tasks each containing one or more Transformations. Transformations represent a single Computational Operation (usually the running of a piece of executable code) and all contextual information that Structant has for the associated operation. Tasks represent a collection of Transformations that must execute on the same Node. Task and Transformations form a series of hierarchical dependency graphs [18], with Tasks having dependence relations with other Tasks, and Transformations having dependence relations to other Transformations within the same Task.

As Nodes execute Tasks and Transformations, the Dispatcher constantly updates the Ontology with new contextual information resulting from executions. This contextual information includes runtime, exit code (i.e., success or error code), failure count, and output file size. The Ontology enforces scheduling correctness by examining the contextual models of Tasks and Transformations to

ensure satisfaction of virtual processing requirements of a Task or Transformation prior to scheduling. These processing requirements can include successful completion of precedent operations, availability of input data, and lack of policy blockage. Although the Scheduler and Policy Broker perform these precondition checks, the Ontology stores all information required for the checks.

6.1 Representing a Pipeline

Structant stores pipelines as a set of Tasks connected to each other by one or more dependencies. When represented as a directed acyclic graph, all elements of a pipeline will form a single connected component. We chose this definition of a pipeline to facilitate memory management and cleanup; it has minimal impact on the user experience.

A pipeline represents a specific chain of Computational Operations that Structant should schedule on Computational Resources. As previously stated, the user must specify pipelines manually for correct operation (see Section 1.1.2). The following example illustrates how a user can convert a shell script into a Structant pipeline. In the problem statement, we established that Structant should not require the modification of the user's existing executables. Creating a pipeline in Structant requires only calling out dependencies, interactions, and call scripts [1] for the user's existing code.

[1] Scripts that invoke operations

6.1.1 Data Dependence

The current version of Structant cannot convert shell scripts or external programs into an Ontological representation automatically for two primary reasons. First, pipelines may include black box software (see Section 1.1.1). Without the ability to examine the code, Structant cannot predict the outputs of the program. As an example, consider the following simple shell script:

```
./preprocess.exe myData.dat
./model.exe
```

In this example, both preprocess.exe and model.exe are black box programs. Suppose that the preprocess.exe program creates an output file in the local directory called "myData.out" and the program model.exe uses this file as the input. Observing this implicit data dependence requires knowledge of the preprocess.exe either from code inspection or an execution trace. Second, given the first reason, we assume that the user has already called out dependence during initial design of the pipeline. In order for a user to construct a pipeline like the one in the example, the user must first recognize the dependence between the two executables. For these reasons, we assume that calling out dependencies when specifying a pipeline in Structant does not incur additional difficulty.

6.1.2 Creating Transformations

The first step in converting an existing set of Computational Operations into an Ontological representation is to recognize the Computational Operations. Suppose a user wishes to convert the following script into to a Structant pipeline:

```
wget www.example.com/db1.txt
wget www.example.com/db2.txt
wget www.example.com/db3.txt
cat db*.txt > joined.txt
```

Since each line of this shell script makes a single call to a piece of executable code, the user can model each line of this script as a single Transformation. The names of the Transformations carry no specific meaning, but the user must use them consistently and correctly for proper dependency recognition. Figure 6.1 shows the Transformations and Task that model this user input.

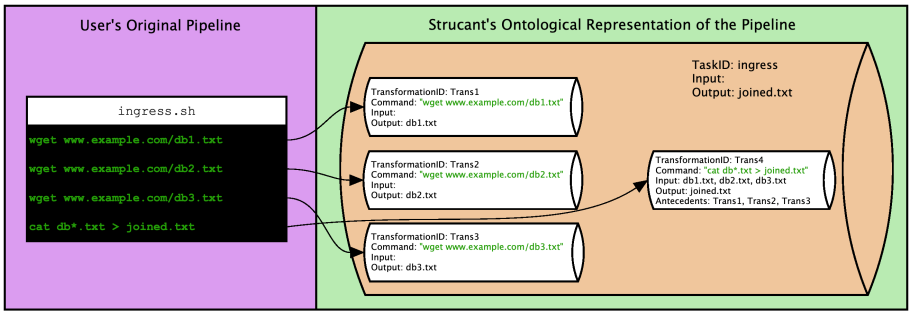


Figure 6.1: Structant models each step in the original pipeline as a separate transaction. Operations that should occur together or are highly dependent on each other should belong to the same Task.

6.1.3 Recognizing Dependence

Given the Task and Transformations shown in Figure 6.1, the user must next recognize and call out the dependence between the Transformations. In each of the first three steps of the original shell script, the executable "wget" downloads an external file to the local directory. In the forth step, the executable "cat" reads the files from the local directory, and writes the joined output to the local directory. In the original script, the first three operations do not depend on any input. While one may argue that the operations depend on the availability of the "db1.txt," "db2.txt", and "db3.txt" files, under Structant's definition of Dataset (see Section 1.1.3), they do not, because they come from outside of the Computational Environment. Figure 6.2 shows the dependence in the user's original script.

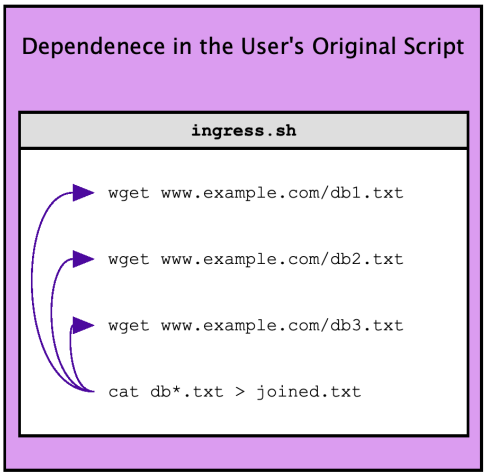


Figure 6.2: The dependencies in the user's original shell script are denoted by arrows. Arrows point from the operation requiring the data to the operation providing the data. All of these relationships are modeled in Structant's Ontology.

Given the dependence in the original script, the user can indicate the dependence between the Transformations. In the user's original script, the first

three Transformations can execute at any time and do not require the Scheduler to run them in a specific order. The first Transformation produces the output "db1.txt", the second Transformation produces output "db2.txt", and the third Transformation produces output "db3.txt". The fourth Transformation requires inputs "db1.txt", "db2.txt", and "db3.txt" and produces output "joined.txt". In order to schedule the fourth Transformation, the first three Transformations must complete successfully, so the user must list these Transformations as antecedents. Figure 6.3 shows the dependency information that Structant models in its Ontology.

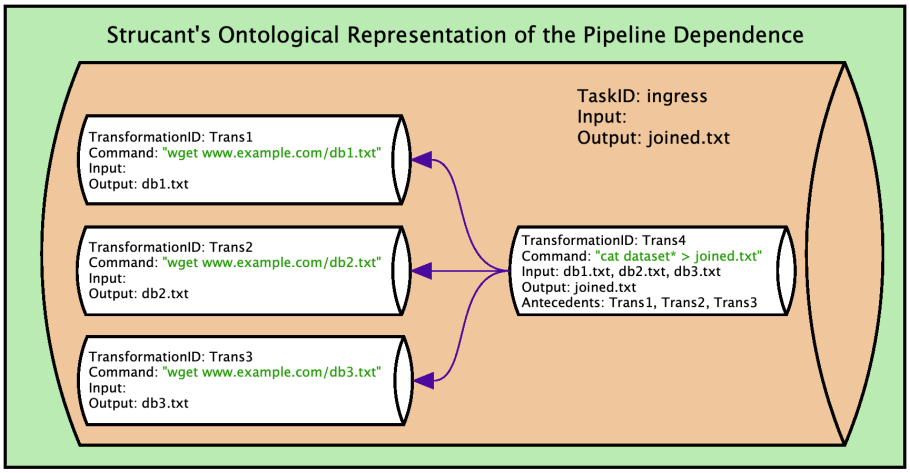


Figure 6.3: Transformations that require the outputs from or otherwise depend on other Transformations cannot be scheduled until all of their antecedents are "COMPLETED." Transformations with no antecedent relations between them can be executed concurrently.

6.2 Execution Environment

The command specified in the Transformation's System Context store indicates the executable code and associated arguments that the user wishes to execute

on a Computational Resource controlled by a running instance of the Workernode. When the Workernode executes a command on a Computational Resource, it is analogous to the user executing the command in the system shell as the user account running the Workernode executable. This design decision has both advantages and disadvantages.

The command will run with the permissions and environment variables of the user account running the Workernode executable. This feature allows flexibility as it may be important to run certain Computational Operations on specific machines due to configuration or software versions. It may also lead to the same Transformation producing different results when mapped to different Computational Resources due to different configurations or local environments. Structant allows this behavior to facilitate greater flexibility and processing in scheduling. With appropriate contextual information and policies, a user can specify the machines that the Scheduler should allow to run Tasks and Transformations.

These constraints comprise both virtual and physical processing requirements. A simple example of a virtual processing requirement occurs when the Computational Environment contains both Windows and Unix machines. Windows executables will not run on Linux and vice versa. In such a case, the user should specify Transformation policy that specifies attributes of machines capable of running each type of executable. Allowing diverse Computational Environments gives the user the ability to run a greater variety of Computational Operations and automatically facilitate data movement between operating different operating systems. The user must include additional contextual information only

when machines in the Computational Environment cannot fulfill the processing requirements of all Tasks or Transformations. Another reason that Structant uses the execution environment on the Workernodes is to allow the user to configure the local execution at runtime. A user can easily specify Transformations in a Task that set Operating System environmental variables [25] or install software. Allowing each Computational Resource to have distinct and differently configured execution environments allows Structant to control a greater variety of Computational Environments.

Chapter 7: Fabric Model

Structant's Fabric Model models the Computational Resources as Nodes and Edges in a directed graph. Nodes model Computational Resources running instances of the Workernode executable. Edges model connectivity between the Workernode executable on two Computational Resources. The Fabric Model stores all the contextual information that Structant has about the Computational Resources in the Computational Environment.

7.1 Connectivity

Structant attempts to observe and profile the *connectivity* between the Nodes, rather than physical connections. For the purposes of Structant, a physical connection refers to a physical medium that provides connectivity between two or more machines (such as Ethernet or Wi-Fi). Connectivity refers to the ability of two machines to communicate through one or more underlying physical connections. Nodes directly connected to the same Layer 2 switch have both a physical connection and connectivity between them. In contrast, Nodes in geographically separate data centers that communicate across more than one router have only connectivity.

Generally, the same physical properties can describe both connections and connectivity. Physical properties such as end-to-end delay (latency) and bandwidth are observable and quantifiable for both connections and connectivity. In the case of connections, the delay and bandwidth are usually just the delay and bandwidth of the underlying physical medium [1] [96]. With connectivity, the delay and bandwidth are some function computed over the intermediate physical connections. The delay can usually be roughly approximated by the sum of all the intermediate delays, and the bandwidth is usually the MIN(x....z) over all the intermediate links [96].

From this, it follows that physical connections likely remain stable over time, whereas connectivity likely changes. Connectivity may change in terms of both bandwidth, when routing changes cause traffic to take different paths, and delay, when large amounts of data saturate the network. This distinction plays a crucial role in understanding the modeling that Structant performs. Structant runs in Computational Environments more diverse than the heterogeneous environments discussed in the prior work [17, 21, 41, 67, 82, 86, 99]. As a prerequisite to responding to the dynamic and diverse nature of these Computational Environments, Structant must construct an accurate model of the underlying network [53].

In oder to facilitate scheduling in these diverse environments, we designed Structant using connectivity as the basis for Edges in the Fabric Model. We made this decision for two reasons. First, observing underlying physical connections

[1] Plus some small processing delay

presents a major logistical challenge. Within a data center, a system administrator may have the ability to enumerate all the physical connections between machines. However, machines in different data centers may not have direct physical connections between them [2]. Moreover, the physical connections may change as underlying routing protocols update routing tables, have incorrect (or no) documentation. For these reasons, we figured it infeasible to enumerate all the physical connections between the Nodes.

Second, having a physical connection does not imply that two applications running on the machines can communicate. Structant's Dispatcher runs at the application layer and needs TCP connections to Workernodes at the application layer on other machines. Connecting two machines at the Network Layer [37, 38] does not guarantee that the instances of the application can communicate. Firewalls, system misconfiguration, or other errors can still prevent applications from communicating. The final reason for choosing connectivity results from the simple observation that Structant needs Application Layer [37, 38] connectivity to function. Modeling physical connectivity provides no useful information if the two Workernode applications cannot connect with each other.

7.1.1 Making Connections (Edges)

Structant takes a hybrid approach to creating Edges in the Fabric Model. The user can *suggest* connections to make and *prohibit* Structant from making other connections. We chose this approach based on the fact that the user should re-

[2] Except in the case of dark fiber

main a key decision maker, but also that Structant needs a correct model of the network; the user can attempt to model available connections, but the Edges will not become part of the Fabric Model until the Structant has verified connectivity between the Nodes. Allowing the Fabric Model to contain incorrect information about Edges can lead Structant to have an inaccurate representation of the state of the Computational Environment. If Structant's Network Model does not accurately represent the Computational Environment, the Scheduler may make misinformed decisions, delaying or impeding pipeline progress. As a simple example, consider two machines, Machine A and Machine B. Machine A has a routable public facing IP address, but Machine B connects to the Internet through a NAT router [49]. While the Workernode on Machine B may have the ability to connect to the Workernode executable on Machine A, the reverse may not hold true. Without verifying connectivity in both directions, the Scheduler may schedule Transformations on Machine B that produce outputs that no other Workernode can access. It is worth noting that Structant can copy files between Computational Resources even without defining Edges in the Fabric Model. Because Structant treats all executables specified in Transformations as black box programs, a user could easily have Structant copy data between Computational Resources using FTP or SCP. Explicitly asking Structant to copy the data [3] provides more robust Execution Observation and better Scheduling.

[3] either with a CopyTransformation, or allowing Structant to PreStage or PostStage the data

7.2 Model Synchronization

Periodically, the Workernodes send information about their current state to the Dispatcher in the form of heartbeat messages. The Dispatcher then updates information in the Fabric Model, Execution Observer, and Ontology with information from these heartbeats. The Heartbeats contain information shown in Table A.5.

7.2.1 Byzantine Fault Tolerance

In order to operate correctly, Structant's Dispatcher must have up-to-date information from all Workernode processes. Because of the requirement for Structant to run in a variety of scenarios (including disaster response), we assumed that Structant will run on networks with high load, high loss rates, and high latency. For this reason, Structant must remain resilient to the Byzantine General's problem [69]. In order to combat this problem, Structant takes a reactive approach. When the Dispatcher and a Workernode communicate, they each form requests and updates based on their current contextual models, without consideration of what information the other process may have. We assume that any message between a Workernode and the Dispatcher can fail to reach its destination or be improperly acknowledged (i.e., only making it partway up the networking stack) [37, 38, 96]. Since the Dispatcher has a model of the entire Computational Environment, and the Workernode has a model of its local environment, they both assume the correctness of their own models. The Dispatcher makes requests based on its own information, and the Workernode provides updates based on

its own information. They both update their models with information received, then make new requests and updates based on the updated models. The following describes the process that Structant uses to dispatch a Transformation to a Node.

1. **Bind Transformation in Ontology**: The Dispatcher associates the Transformation with the Node that will run the Transformation and updates information in its local Ontology to indicate that the Transformation will run on the target Node.

2. **Dispatch Transformation to Node**: The Dispatcher creates a message with all necessary information to run the Transformation and sends it to the Workernode running on the Computational Resource.

3. **Workernode Updates Local Ontology**: The Workernode updates its local Ontology by creating a model of the Transformation received from the dispatcher, and entering it into the runnable queue.

4. **Workernode Heartbeats**: The Workernode informs the Dispatcher about its local states. In this update, the newly assigned Transformation shows "PENDING."

5. **Dispatcher Updates Context**: The Dispatcher updates its local models based on the information received from the heartbeat. The newly assigned Transformation shows "PENDING" in the Fabric Model and Ontology, to indi-

cate that the Workernode has updated its local Ontology to show the Transformation as pending.

6. **Workernode EXECs Transformation**: The Workernode begins executing the command in the Transformation and updates its local Ontology to reflect this.

7. **Workernode Heartbeats**: The Workernode informs the Dispatcher about its local state. In this update, the newly assigned Transformation shows "RUNNING".

8. **Dispatcher Updates Context**: The Dispatcher updates its local models based on the information received from the heartbeat. The newly assigned Transformation shows "RUNNING" in the Fabric Model.

After the Transformation has completed, the Dispatcher and Workernode use the reverse of this process to deliver the result to the Dispatcher. The following describes this process:

1. **The Workernode marks the Transformation completed**: The Workernode updates it local Ontology to reflect the fact that the Transformation has completed.

2. **Workernode Heartbeats**: The Workernode informs the Dispatcher about its local states. In this update, the newly completed Transformation shows "COMPLETED."

3. **Dispatcher Updates Context**: The Dispatcher updates its local models based on the information received from the heartbeat. The newly completed Transformation shows "COMPLETED" in the Fabric Model and Ontology, to indicate that the Workernode has updated its local Ontology to show the Transformation as having completed.

4. **Dispatcher sends UNBIND**: The Dispatcher sends the Workernode a message indicating that it should remove all information about the Transformation from its local models.

5. **Workernode Updates Context**: The Workernode updates its local models based on the information received from the heartbeat removing the newly completed Transformation from its local Ontology.

6. **Workernode Heartbeats**: The Workernode informs the Dispatcher about its local states. This update does not include the newly completed Transformation since the Workernode removed it from its Ontology.

7. **Dispatcher Updates Context**: The Dispatcher updates its local models based on the information received from the heartbeat and removes the newly completed Transformation from the Fabric Model.

This process helps Structant operate correctly in environments subject to Byzantine faults. In this process, both the Workernode and the Dispatcher will continually communicate with the other based only on their local context models. Once the Dispatcher has bound the process to the Node in its Ontology, it will con-

tinually dispatch the Transformation as long it has updated the status to "RUNNING," "COMPLETED," or "FAILED." The Workernode will always include the Transformation in its updates until it removes the Transformation from its local Ontology via an unbinding process.

Chapter 8: Policy Broker

Policies specify the functions that Structant uses to infer processing requirements from the contextual information and make correct scheduling decisions in response to the current state of its environment [4]. In the Structant processing environment, policies encode all physical, virtual, and organizational processing requirements. Policies govern the interactions of each entity in the Computational Environment with every other entity by and dictating whether processing requirements should allow the Scheduler to map a Task or Transformation to a particular Node. Users can attach policies to any entity in the Computational Environment. Node policies determine the jobs the Node may run, Task and Transformation policies determine the Nodes that can run the Task or Transformation, and Edge policies determine what types of data may traverse the Edge. Structant's Scheduler relies on policies to infer meaning from contextual information to determine the validity of mapping under consideration.

Structant provides three policy scopes: local, global, and user. Local Policies apply only to a specific entity, and are evaluated only when considering a mapping involving the entity. Global Policies apply to all entities of a specific type, and are evaluated during all mapping considerations. User Policies can apply to

one or more entities in the system. Local and Global Policies have the form:

EntityType	Operator	Values	ContextLabel	FailOnMissing	Time

User Policies are specified as Ruby functions that return either TRUE or FALSE.

Structant evaluates the Local Policies of an entity against a foreign entity. In other words, when attempting to schedule a Transformation on a Node, Structant checks the Transformation's policies against the Node under consideration. In this check, the local entity refers to the entity with the attached policy (i.e., the Transformation), and the foreign entity refers to the entity that Structant is checking the policy against (i.e., the Node). When performing Global Policy checks, Structant evaluates the policies against all compatible entities, meaning that Structant treats all entities as foreign entities during global policy check. Fields in User Policies have the following definitions:

1. **EntityType**: This corresponds to the ENTITY TYPE label in the foreign entity's context store. This field must correspond to one of the acceptable entity types: "NODE", "EDGE", "TASK", "TRANSFORMAITON". This field indicates the type of entity to which the policy applies. Structant will only evaluate a policy against a foreign entity's context store if the foreign entity's ENTITY TYPE matches the EntityType field of the policy.

2. **Label**: This field specifies the Contextual Identifier that the Policy Broker should look up in the foreign entity's context store.

3. **Relationship**: This field defines the relationship between the Contextual Value identified by "Label" in the foreign entity's context store, and the values specified in "Values". Table 8.1 provides formal definitions of the relationship operators.

   - **MEMBEROF**: The Contextual Value on the foreign entity must be a member of the set specified by the values field.
   - **NOTMEMBEROF**: The Contextual Value on the foreign entity must not be included in the set specified by the values field.
   - **DOESNOTINCLUDE**: No member of the Contextual Value on the foreign entity may equal to value specified by the values field.
   - **ATLEAST**: The Contextual Value on the foreign entity must be equal to or greater than the value specified by values.
   - **EXACTLY**: The Contextual Value, on the foreign entity, must be the same as the value specified by values.
   - **ATMOST**: The Contextual Value on the foreign entity must be equal to or less than the value specified by the values field.
   - **LESSTHAN**: The Contextual Value on the foreign entity must be strictly less than the value specified by the values field.
   - **GREATERTHAN**: The Contextual Value on the foreign entity must be strictly larger than the value specified by the values field.

- **NOTEQUALTO**: The Contextual Value on the foreign entity must be any value other than the value specified by the values field.

4. **Values**: This field specifies the values that the Contextual Value in the foreign entity is related to by the relationship operator. This field takes tuples with one or more elements.

5. **Schedule**: This field indicates when to enforce the policy. This field has the form HH:MM:SS-HH:MM:SS-DDD. The first section indicates when the Policy Broker should start enforcing the policy. The second section indicates when the Policy Broker should stop enforcing the policy. The third section indicates the days of the week when the Policy Broker should enforce the policy. HH denotes the two digit hour written in twenty-four hour format. MM denotes the minutes. SS denotes the seconds. DDD denotes the day of the week and as one of: "MON", "TUE", "WED", "THU", "FRI", "SAT", "SUN". In addition, the user can specify "ALWAYS" to indicate that the Policy Broker should *always* enforce the policy. The Policy Broker can handle time granularity down to seconds, and will round down to the nearest second.

Operator	Symbol	Formal Definition
"MEMBEROF"	$\in$	Foreign[Y] $\in (V_1, ...V_i)$
"NOTMEMBEROF"	$\notin$	Foreign[Y] $\notin (V_1, ...V_i)$
"DOESNOTINCLUDE"	$\nsubseteq$	Foreign[Y] $\nsubseteq (V_1, ...V_i)$
"ATLEAST"	$\geq$	Foreign[Y] $\geq V_1$
"EXACTLY"	$=$	Foreign[Y] $= V_1$
"ATMOST"	$\leq$	Foreign[Y] $\leq V_1$
"LESSTHAN"	$<$	Foreign[Y] $< V_1$
"GREATERTHAN"	$>$	Foreign[Y] $> V_1$
"NOTEQUALTO"	$\neq$	Foreign[Y] $\neq V_1$

Table 8.1: These operators can be used to control the Scheduler's mapping of Transformations and Tasks to Nodes and Edges Foreign[Y] is the value of the Contextual Identifier in the remote entity's context store.

8.1 Policy Examples

At this point, it is worth giving some examples of how policies can enforce real-life organizational processing requirements.

8.1.1 Example 1: Machine Policy

Consider a system shared by users in a research laboratory. Users of this system typically run short, high CPU usage jobs. The system in this example has multiple CPUs so users can share the system with minimal interference despite the high CPU usage nature of the jobs they tend to run. In this system, users must run hundreds of jobs per day. Suppose also that one user has a large job that will utilize all CPU capacity on all cores for a duration of eight or more hours. Scheduling this job to run on the shared system during work hours (09:00 to 17:00) could have a noticeable impact that interferes with the ability of other users to work efficiently. A user could create a Local Policy that would prohibit the Scheduler

from scheduling the large job on the system during business hours. Table 8.2 provides an example of policies a user could define to enforce this behavior.

EntityType	Operator	Values	ContextLabel	FailOnMissing	Time
"TRANSFORMATION"	$\neq$	[LargeJob]	"JobDesc"	TRUE	9:00:00,17:00:00,MON
"TRANSFORMATION"	$\neq$	[LargeJob]	"JobDesc"	TRUE	9:00:00,17:00:00,TUE
"TRANSFORMATION"	$\neq$	[LargeJob]	"JobDesc"	TRUE	9:00:00,17:00:00,WED
"TRANSFORMATION"	$\neq$	[LargeJob]	"JobDesc"	TRUE	9:00:00,17:00:00,THU
"TRANSFORMATION"	$\neq$	[LargeJob]	"JobDesc"	TRUE	9:00:00,17:00:00,FRI

Table 8.2: These policies specify that a Node may not run a large CPU Intensive Task during normal business hours.

These policies will prohibit Structant from scheduling Transformations where the JobDesc in the Transformation's User Context store equals "LargeJob", on the Node during normal business hours, Monday through Friday from 09:00 until 17:00. This type of policy relies on the user to correctly identify the large job and create appropriate contextual information. This policy will apply to any Transformation that has a Contextual Identifier "JobDesc" that maps to the Contextual Value "LargeJob", but will not apply to Transformations with a Contextual Identifier "JobDesc" that map to any Contextual Value other than "LargeJob" or that do not have the Contextual Identifier in User Context. This type of policy allows the system to run other, non-interfering jobs during those hours, rather than sitting idle.

8.1.2 Example 2: Transformation Policy

In the previous example, we showed how a user may create a policy to minimize impact to other users of a machine by attaching a Local Policy to a Node in the system. In this example, we show how a user may use a policy to enforce

monetary policies by only allowing Transformations of a certain type to run on certain Nodes. Consider the example of a hybrid cloud [32] with some of an organization's Computational Resources residing on-premise, and some residing in a remote data center managed by a third party. In this example, the organization pays for bandwidth to and from the remote data center as well as CPU and disk usage. When scheduling Transformations and Tasks, Structant will try to find the Node with the lowest runtime estimate for the job. However, organizational policies and goals may create situations where processing speed has a lower priority than monetary cost. For example, the organization might not want to schedule disk or bandwidth intensive jobs on the remote data center because of the cost of using Computational Resources. Even though the job will likely complete sooner if run on the remote data center, the user may find that cost savings from processing locally justifies the slower execution time. The policies attached to the Transformation may look something like:

EntityType	Operator	Values	ContextLabel	FailOnMissing	Time
"NODE"	=	[ONPREMISE]	"MachineLocation"	TRUE	ALL

Table 8.3: These policies specify that Transformation must run on a Node tagged as "ONPREMISE"

8.1.3 Example 3: Link Policy

User can also attach policies to Edges in the Fabric Model. A user may require such a policy when he or she needs to control what type of data traverses a link. Besides bandwidth and speed considerations, the organization may also need to maintain regulatory compliance [34, 91]. Consider an example where an organi-

zation needs to transfer data between two data centers for processing. The data centers have two links between them as shown in Figure 8.1. The first link uses

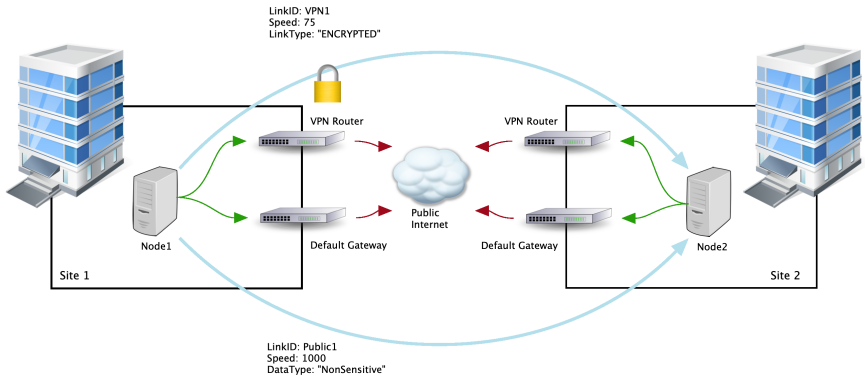


Figure 8.1: Two data centers connected via the public Internet and an encrypted VPN tunnel. Node 1 can use two gateways to access Node 2, represented by two virtual links to Node 2 with different properties and described by contextual information.

the public Internet with a benchmarked speed of (1000 Mbps / 1000 Mbps), while the second link uses a private, encrypted VPN tunnel with a benchmarked speed of (75 Mbps / 75 Mbps). When scheduling for speed, the Scheduler should utilize as much of the public Internet link as possible, and then utilize the encrypted VPN link only after saturating the first link. However, when considering regulatory compliance and transferring sensitive data (e.g., healthcare [91] or financial data [34]), the user may wish to utilize only the encrypted VPN tunnel. In this case, the user can create a policy on the public link to disallow the transfer of protected data. The policies attached to the public link may look something like the following:

EntityType	Operator	Values	ContextLabel	FailOnMissing	Time
"COPYTRANSFORMATION"	=	[NONSENSITIVE]	"DataType"	TRUE	ALL

Table 8.4: These policies specify that a Link may only be used to facilitate a Copy Transformation that has been classified as "NONSENSITIVE"

The user could have written this policy another way and achieved the same result. Alternately, the user could attach a policy to the CopyTransformation that would require the system to select a link identified as secure for the transfer. A policy that would have the same effect could have been written as follows:

EntityType	Operator	Values	ContextLabel	FailOnMissing	Time
"LINK"	=	[ENCRYPTED]	"LinkType"	TRUE	ALL

Table 8.5: These policies specify that a Copy Transformation may run on a Link classified as "ENCRYPTED"

While the policies provide identical functionality (provided that the Link and the CopyTransformation both have correct information in their respective context stores), there is a difference. The first policy, as it applies to the Link, requires that *ALL* traffic sent over it be explicitly designated as NONSENSITIVE. While a subtle difference, this may be desirable as users may be more likely to forget to attach the policy to the CopyTransformation than to attach an incorrect policy. In the first example, the system will err on the side of caution and redirect all traffic that hasn't been specifically designated as NONSENSITIVE to the encrypted VPN tunnel.

8.1.4 Example 4: Global Policies

Global Policies function in a nearly identical fashion to Local Policies, with the key difference that Global Policies apply to all entities of a specific type and have more than one rule. Creating a Global Policy for an entity type has the same effect as creating a Local Policy with the same rules, and attaching the Local Policy to every entity in the system of the entity's type. Global Policies offer no additional

features over Local Policies, but instead afford the user a quick way to create policies that apply to more than one entity.

As an example of the utility of Global Policies, consider an organization that collects photographic data from unmanned aerial vehicles (i.e., drones), then processes the data to form 3D models of a geographic region. The first step in the pipeline involves a step called "point matching" that requires the use of a dedicated graphics processing unit (GPU). The second step involves a step called "model combination" that requires large amounts of memory. To simplify creation of the pipelines, the organization could create Transformation policies as shown in Table 8.6:

EntityType	Operator	Values	ContextLabel	FailOnMissing	Time
"NODE"	=	[TRUE]	"GPUEnabled"	TRUE	ALL
"TRANSFORMATION'	=	["PointMatch"]	"TransType"	TRUE	ALL
"NODE"	$\geq$	[1024000]	"MemoryMB"	TRUE	ALL
"TRANSFORMATION'	=	["ModelCombination"]	"TransType"	TRUE	ALL

Table 8.6: These policies globally specify that a Node must be GPU Enabled to run Point Matching Transformations, and have 1024000MB or more memory to run Model Combination Transformations.

The two global policies shown in Table 8.6 govern that PointMatch Transformations can only run on GPU Enabled Nodes, while ModelCombination Transformations can only run on Nodes with more than 1024000MB of memory.

8.2 Advanced Policies

At this point, it should be evident that the language used to specify Global and Local Policies cannot specify complex policies. While the language used can specify policies such as "Do not schedule Transformations of type wget1 dur-

ing business hours" they cannot encode policies such as "Do not schedule more than four Transformations of type wget1 in a 24-hour window." Users may need the ability to specify advanced policies such as these in order to model complex and dynamic processing requirements. Advanced policies have countless uses in real-world computing environments. As an example, consider an API that serves responses to queries. In some cases, the API may rate limit and allow a single IP address to make N queries in a 24-hour period. Without the ability to encode this information into the Structant's models, Structant may keep scheduling the same job repeatedly on the same node, until the 24-hour period expires and the system API allows more requests. While Structant can assess the likelihood that a Transformation will succeed on a given Node based on prior runs, this technique presents two issues. First, allowing the Transformation to repeatedly fail incorrectly skews the probability of success. The Node will repeatedly fail running the Transformation and it will take time for the success probability to increase, so Structant may not schedule the Transformation on the Node as soon as it should. Second, it may slow down the overall progress if the failed Transformation is a small part of a long-running Task. If the request to the API is the last step in a Task that takes three hours to run, the entire Task will have to re-run if the Scheduler reassigns the Task to a different Node.

As another example of a policy that Local and Global Policies cannot define, consider the previous example of the 3D reconstruction. In this example, some machines may have both enough memory and GPU capabilities, allowing them to run both PointMatch and ModelCombine. However, a user may wish

to disallow machines from running PointMatch and ModelCombine at the same time. While these types of policies may seem trivial, the difficulty lies with both capturing and exposing the correct contextual information.

8.2.1 Defining Advanced Policies

To enable advanced policies, Structant exposes a portion of the User and System Context stores through the Global Context Broker (see Section 5.3), and allows users to interact directly with the context store, by defining their User Policy functions. This design decision limits the amount of information that Structant must keep and simplifies the models by allowing the user to define functions to perform context management. User Policies are functions specified by the user that interact directly with User and System Context stores. Similarly to function pointers, these functions comprise user supplied functions with well-defined inputs and output. Structant defines three types of user functions: PreCompute, PostCompute, and Evaluate. The following describes when Structant calls these functions:

1. **PreCompute**: The Scheduler calls the PreCompute functions of all entities in a mapping after selecting the mapping for execution, and before dispatching the Transformation to the Workernode. PreCompute functions allow users to manipulate the User and System Context stores of the entities in the mapping before the Workernode executes the Transformation. These functions provide the user with the ability to encode advanced ac-

counting information such as tracking when a particular Task or Transformation started, the number of running Tasks or Transformations, the types of Transformations running on a Node, and the number of Transformations that have run on this Node.

2. **PostCompute**: The Scheduler calls the PostCompute function on all entities in a mapping after the assigned Node has completed execution of the Task or Transformation. PostCompute functions allows users to encode advanced accounting information after the Task or Transformation has completed and to track statistics about the Task or Transformation. These functions can facilitate tracking how long a job has run or that it has ended, counting successes or failures of a Task or Transformation, or cleaning up obsolete contextual information.

3. **Evaluate**: The Scheduler calls the Evaluate function on all entities in a mapping during policy evaluation. Evaluate functions allows the user to specify functions that Structant should compute over the User and System Context stores to evaluate whether a given mapping is allowed. Like User and Global policies, Evaluate functions must return TRUE to indicate a valid mapping; the Scheduler interprets any return value other than TRUE as indication of an invalid mapping.

The following examples illustrate how Structant utilizes User Policy functions to infer meaning from contextual information to facilitate advanced scheduling operations by recognizing complex processing requirements.

8.2.2 API Example: Rate Limiting 1

In a previous example, we considered a scenario where Nodes in the system needed to make queries to an external API. In this example, the API service rate limited requests to N requests per IP address within a Y hour period. For this example, let N = 1024 and Y = 24. A user can model this processing requirement using PreCompute, PostCompute, and Evaluate functions. In the PreCompute function, the user defines a Contextual Identifier to store Contextual Values representing previous execution times of Transformation that queried external API. To accomplish this, the PreCompute function accesses the Global Context API and calls the USER_CONTEXT_CREATE function to create the ContextIdentifier "API1RequestTimes". The following code snippet provides pseudocode for this function.

```
1  def pre_compute_api1_request_times(global_context_broker)
2      #request the API1RequestTimes information from the context store
3      this_node = global_context_broker.GET_MAPPING_INFO("NODE1")
4      request_times = global_context_broker.USER_CONTEXT_GET("NODE",
           this_node, "API1RequestTimes")
5
6      #create the information if the context store does not include it
7      if(request_times == nil)
8          global_context_broker.USER_CONTEXT_CREATE("NODE", this_node,
               "API1RequestTimes")
9          global_context_broker.USER_CONTEXT_STORE("NODE", this_node,
               "API1RequestTimes", [])
10     end
11 end
```

The PostCompute function accesses the Global Context API and calls the USER_CONTEXT_GET function to retrieve the current list of execution times. The PostCompute function then appends the current time to the list, and calls the

USER_CONTEXT_STORE function to store the new list under the "API1RequestTimes" identifier. The following code snippet provides pseudocode for this function.

```
def post_compute_api1_request_times(global_context_broker)
    #request the API1RequestTimes information from the context store
    this_node = global_context_broker.GET_MAPPING_INFO("NODE1")
    request_times = global_context_broker.USER_CONTEXT_GET("NODE",
        this_node, "API1RequestTimes")

    #append the current time
    request_times << Time.now.to_i

    #store the new list in the context store
    global_context_broker.USER_CONTEXT_STORE("NODE", this_node,
        "APIRequestTimes", request_times)
end
```

The Evaluate function retrieves the current list of execution times using the USER_CONTEXT_GET function, then iterates over the list of execution times, counting the number of items in the list with a value less than 24 hours prior to the current time. If it counts less than the threshold of N=1024 in the past 24 hours, then the Evaluate function returns TRUE, otherwise it returns FALSE. The following code snippet provides pseudocode for this function.

```

def evaluate_api1_request_times(global_context_broker)
    #request the API1RequestTimes information from the context store
    this_node = global_context_broker.GET_MAPPING_INFO("NODE1")
    request_times = global_context_broker.USER_CONTEXT_GET("NODE",
        this_node, "API1RequestTimes")

    #create a count variable
    times_run_in_last_24 = 0
    time_now = Time.now.to_i

    request_times.each{ |rt|
        if(time_now - rt < 24*60*60)
            times_run_in_last_24 += 1
        end
    }
```

```
16
17      if(times_run_in_last_24 >= 1024)
18          return false
19      else
20          return true
21      end
22  end
```

8.2.3 API Example: Rate Limiting 2

Consider a more complex version of the previous example where a collection of ten Nodes shares one public IP address through Network Address Translation (NAT) [49]. Figure 8.2 provides an illustration of this environment. To the API

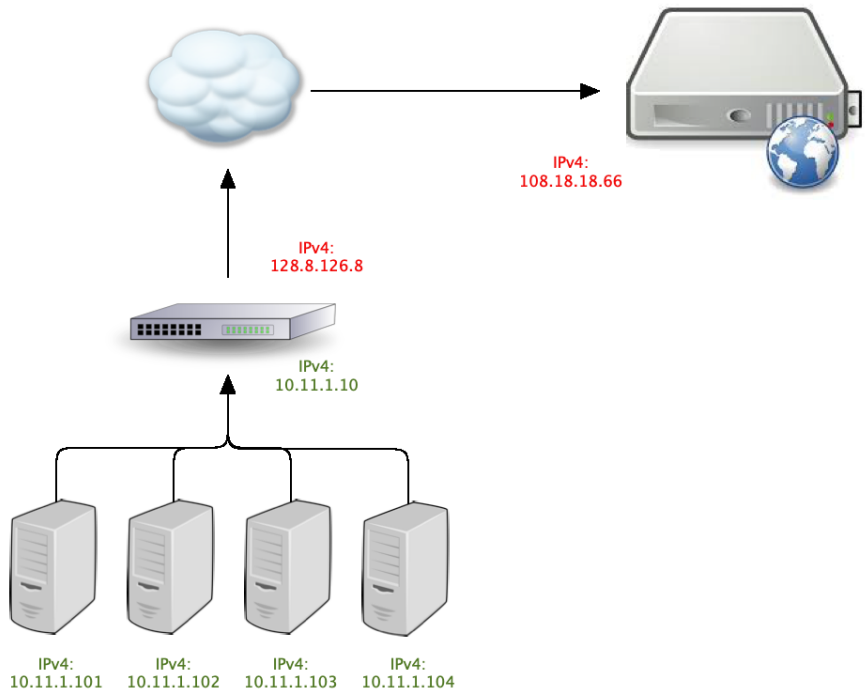


Figure 8.2: Nodes in this cluster are behind a router performing NAT [49]. To machines outside of the local subnet, requests all appear to come from the same location. The API rate limit will apply to all of the Nodes in the cluster as a whole.

server, all requests from Nodes in this cluster appear from the same IP address. Using the PreCompute, PostCompute, and Evaluate functions in the previous example, API requests will start to fail, because the Evaluate function does not account for other Nodes behind the NAT router that may have already run the Transformation. To resolve this, the user might divide the threshold by the number of Nodes in the cluster and allow each of the Nodes to make its fair share of requests. While this approach provides correct behavior, it introduces a problem. This approach may slow down overall progress by imposing unnecessary resource constraints. Consider a Task that has 1024 Transformations that all make a single request to the external API server. The definition of a Task states that the Scheduler must run all 1024 Transformations on the same Node. When running on a four Node cluster, it would take four days to process the Task because the policy would only run 256 API requests per day. The User Policy functions described in the previous example can easily overcome this limitation. When starting the Workernode, the user needs only to create a piece of contextual information in the System Context stores of the Workernodes that share the NATed IP address. The modified pseudocode for the Evaluate is shown below.

```
1
2  def evaluate_api1_request_times(global_context_broker)
3      #request the API1RequestTimes information from the context store
4      this_node = global_context_broker.GET_MAPPING_INFO("NODE1")
5      node_ip = global_context_broker.GET_SYSTEM_CONTEXT("NODE",
           this_node, "EXTERNAL_IP")
6
7
8      #create a count variable
9      times_run_in_last_24 = 0
10     time_now = Time.now.to_i
```

```
11
12      #Get the list of all Nodes in the system
13      for node_ in global_context_broker.SELECT_ENTITIES("NODE")
14        if(node_ip == global_context_broker.GET_SYSTEM_CONTEXT("NODE",
            node_, "EXTERNAL_IP"))
15
16        request_times = global_context_broker.GET_USER_CONTEXT("NODE",
            node_, "API1RequestTimes")
17        request_times.each{ |rt|
18            if(time_now - rt < 24*60*60)
19               times_run_in_last_24 += 1
20            end
21        }
22
23      if(times_run_in_last_24 >= 1024)
24         return false
25      else
26         return true
27      end
28  end
```

From this example, it should be clear that Structant does not store contextual information outside of entity User and System Context stores. This design decision reflects the principle that the Global Context is simply the aggregation of the User and System Context of all entities in the system.

8.2.4 Policy Limitations and Correctness

Policies are a double-edged sword. While policies enable advanced scheduling decisions, they can put the Scheduler in a situation where it cannot make progress or where it makes progress slowly. As a simple example of policies that inhibit the Scheduler from making progress, consider the policies shown in Table 8.7. By

EntityType	Operator	Values	ContextLabel	FailOnMissing	Time
"TRANSFORMATION"	$=$	[TASK1]	"JobDescription"	TRUE	ALL
"TRANSFORMATION"	$\neq$	[TASK1]	"JobDescription"	TRUE	ALL

Table 8.7: These policies conflict and will prevent the Scheduler from ever scheduling the Transformation.

creating these two Local Policies on a Node, the Scheduler will never assign it any Tasks. The first policy will prohibit any Transformation with a JobDescription not equal to "Task1" and the second policy will prohibit any Transformation with a JobDescription equal to "Task1". These policies are not necessarily incorrect; they are simply what the user specified. While this is a trivial example, policies can be complex and spread across different entities, so the conflicts may not be immediately obvious.

Legitimate policies in the real-world may create a situation where the Scheduler cannot proceed. For example, a Computational Environment may include multiple clusters, but the administrators of every cluster have prohibited the Nodes in the clusters from running Transformations with a JobDescription equal to "GPUCompute1" and the pipeline includes a Transformation with a JobDescription equal to "GPUCompute1". In this scenario, the Scheduler cannot map the Transformation to a Node until the user lifts one of the policies.

Reconsidering the example of a cluster of machines sharing a NAT'd IP address and needing to make queries against an external API server, the Scheduler may schedule two Tasks with N=1024 requests on the machines in the cluster. The Policy Broker would prohibit the Scheduler from scheduling more than 1024 of the Transformations on the cluster in a 24-hour period. Since Structant cannot explore future conditions of User Policy functions, the Scheduler relies on the user to provide specific, precise, and meaningful user functions to assist with context inference and induce correct scheduling behavior based on current contextual

information. By introducing these known limitations, Structant enables extraordinarily complex scheduling behavior while achieving reasonable progress.

Chapter 9: Execution Observer

Accurate runtime prediction is a critical component in many scheduling algorithms [35]. Previous work has addressed runtime prediction in a variety of ways. Input sampling is one popular technique. Before running a set of computations over a large dataset, a scheduler randomly samples a small portion of the input dataset, computes the set of operations over the sample, and uses the observed runtime to compute runtime estimate of the same set of computations over the full dataset [102]. This technique's popularity comes from its simplicity and accuracy. Despite these benefits, restrictions limit its generalizability. If the scheduler cannot randomly sample the dataset (e.g., binary data), or if the input data size does not predict the performance of the full algorithm, input sampling will not work.

Static resource allocation is another technique used for runtime prediction. This technique is popular in data centers where a service provider needs to fulfill a set of Service Level Objectives agreed upon by a client. In static resource allocation, system engineers define performance models [30], then allocate machine resources such that the runtime of steps in the pipeline statistically fall within the parameters specified in the Service Level Agreement. This technique has tradi-

tionally involved manual design of the performance models and hand tuning of the resource allocation. Recent work has focused on characterizing the performance of each component in the resource model as a function of resource allocation, and then automatically tuning the allocation based on the Service Level Agreement [30]. This technique does not generalize well to black box executables or truly heterogeneous hardware.

The domain of Big Data has given rise to simulators that can predict the runtime of large jobs on popular Big Data stacks. Simulators such as BigDataNetSim [13], dagSIM [61], and cloudSim [24] have demonstrated efficacy at modeling and predicting runtime estimates for Big Data jobs. These simulators model the components of popular Big Data stacks (e.g., HDFS, Yarn, MapReduce) and predict the performance of the components processing different inputs and queries. BigDataNetSim extended the model by also including information about the network and switching fabric. While these simulators can reduce the time and cost of scheduling jobs running on Big Data stacks, they do not predict performance for any system other than the well-defined components of popular Big-Data stacks.

Research has also explored techniques for modeling and predicting the runtime of more general applications. Using historical runtime observations and relevant metadata, recent schedulers compute both coarse and sophisticated predictions of runtime without modifying existing applications. Researchers have applied *a priori* prediction models on both Big Data stacks [8] and on multicore machines [74]. Prior work has demonstrated performance increases when modeling a task's sensitivity to certain resources and scheduling accordingly. In one

technique, the user specifies resources to track using configuration files, then indicates how much of each resource each task takes. The scheduler then models each task's sensitivity to certain resources as a system of linear equations [74]. These modeling techniques have also been augmented by modeling the dependencies between tasks, and scheduling for data locality [41, 80]

Structant employs a hybrid approach to runtime prediction, using both performance modeling based on prior execution traces and simulation of network performance. Unlike prior work, Structant automatically selects the resources and features to use when modeling execution, and automatically performs relevant feature selection.

9.1 Profiling Challenges

Structant employs modeling and simulation techniques from prior work but introduces new challenges because of its complex scheduling goals. While prior work has explored profiling in heterogeneous Computational Environments, the environments discussed have been far less diverse than Structant's intended operating environments. Heterogeneity has often carried an implicit assumption that computational resources varied mainly in the amount of each resource they provided [17, 41, 67, 82]. These models often fail to encompass the vastly different hardware performance of these resources in a truly heterogeneous environment. Consider, as an example, the benchmarks [94] of running the same application on a first generation Intel i7 CPU and on a current ninth generation i7 CPU. Both

CPUs provide the same number of resources (e.g., CPU cores), but they exhibit vastly different performance characteristics. Similar examples can be exhibited for different types of memory (e.g., DDR2 vs DDR4), different types of disk (e.g., Hard Drive Drive (HDD) vs Solid State Drives (SSDS)). For these reasons, it is important for the Execution Observer to also account for properties of underlying hardware.

9.2 Context in Execution Profiling

In place of predefined attributes [74], Structant's Execution Observer uses relevant context to model entities in the environment. This technique allows Structant to build more complex and diverse models of entities, and to model subtle differences between entities. At a high level, the Execution Observer records the execution performance of every (Transformation $\rightarrow$ Node) mapping, and uses machine learning algorithms to make runtime estimates for new (Transformation $\rightarrow$ Node) mappings based on these observations. Historical execution observations include a complete copy of the both the User and System context stores of the Transformation and the Node at the time of execution. The Execution Observer stores observations as show in Equation 9.1, where $Trans_t$ denotes the User and System Context stores of a given Transformation t, $Node_n$ denotes the

User and System context stores of a given Node *n*, and $Time_\delta$ denotes the performance (makespan), of the *t* under a mapping to *n*.

$$\text{History} = \{((Trans_t \rightarrow Node_n) = Time_\delta), ..., ((Trans_{t'} \rightarrow Node_{n'}) = Time_{\delta'})\} \tag{9.1}$$

Since the Scheduler maps "EXEC" Transformations single Node and "COPY" Transformations to two Nodes and a Link, $Node_n$ represents the context stores the source and destination Node as well as the Link.

9.2.1 Collision Avoidance in Context Storage

Because Structant maintains two context stores for each entity (see Chapter 5) and enforces no restrictions on Contextual Identifiers in the User Context store, the same Contextual Identifiers may appear in both the User and System context stores of an entity. Even without consideration of the User Context stores, all Node entities have identical Contextual Identifiers in the System Context stores, meaning that all "COPY" Transformations will result in context collision. To address this issue, the Execution Observer deterministically modifies the Context Identifiers of the entities in mappings by prepending unique Strings before inserting their information into the execution history.

The Execution Observer modifies Context Identifiers according to Table 9.1.

EntityType	System Context	User Context
“Node1”	`"N1_SC_"`	`"N1_UC_"`
“Node2”	`"N2_SC_"`	`"N2_UC_"`
“Link”	`"L1_SC_"`	`"L1_UC_"`

Table 9.1: Each Contextual Identifier is modified by prepending a String to avoid collisions

In the case of regular Transformations, the single Node is referenced as Node1. In the case of CopyTransformations, the source Node is referenced as Node1, the destination Node is referenced as Node2, and the link is referenced as Link. By modifying Contextual Identifiers in this way, the Execution Observer can distinguish and map corresponding Context Identifiers across distinct observations. For example, a Transformation that copies data from NodeA to NodeB will have different context store identifiers than a “COPY” Transformation that copies data from NodeB to NodeA. In the first example, the Execution Observer will prepend NodeA’s User Context and System Context Identifiers with “N1_UC_” and “N1_SC_” respectively, and will prepend NodeB’s User and System Context Identifiers with “N2_UC_”and “N2_SC_” respectively. Structant makes this distinction between the Nodes because in CopyTransformations, Contextual Identifiers may have different meanings between Nodes. Modifying the context identifiers before inserting into the execution history not only avoids collisions, it also helps Structant build more granular models of the execution environment.

9.2.2 Assumptions

In order to facilitate machines learning, the Execution Observer makes implicit assumptions about the values stored in the user and system context stores of the entities.

Associations Across Execution Environments

Structant assumes that identical Context Identifiers in the execution history encode the same type of information. Structant's modeling mechanisms distinguish between source and destination Context Identifiers and between Context Identifiers in the User and System Context stores, but treats identical prepended Contextual Identifiers as comparable. In order words, when modifying User Context, users should not use the same Contextual Identifier on different Nodes, Tasks, or Transformations if the Contextual Value encodes a different type of metadata. If Contextual Values associated with the same Contextual Identifier across different entities of the same type encode different information, the models build by the Execution Observer may not correctly capture the interrelations of the attributes of the entities in a mapping which may result in inaccurate runtime predictions.

Numerical vs Nominal Context Values

Structant also assumes that the user will provide numeric Contextual Values for Contextual Identifiers that the Execution Observer should include in the runtime prediction model. Regression models have shown to be effective in execution pre-

diction models [30, 89, 103]. The Execution Observer use Logistic Regression to predict the runtime of a given Transformation in a given $(Trans_t \rightarrow Node_n)$ and logistic regression cannot handle nominal [1] inputs. For this reason, the Execution Observer only uses Contextual Identifiers with numerical Contextual Values during logistic regression. The Execution Observer uses nominal values for relevant context selection (See Section 9.3.1), but not in the building of the model. To facilitate use of a nominal value in the computation and prediction of future runtime, the user must encode the value as a numerical Contextual Value. For example, the Execution Observer will not use the contextual information ("COMP_TYPE" $\rightarrow$ "SQL") in the logistic regression model. To force the Execution Observer to use this type of contextual information, the user should simply encode the contextual information as ("COMP_TYPE_SQL" $\rightarrow$ 1). Doing this may have certain undesired affects (such as shrinking the size of Structant's *a priori* model).

9.3 Modeling Execution

When the Scheduler binds a Transformation to a Node, the Execution Observer creates an execution observation object for the mapping. The execution observation includes a complete copy of the User and System Context stores of the Nodes and Links in the mapping at the time of dispatch. When the Node(s) complete the Transformation, the Execution Observer stores the runtime in the associated observation entry and adds the observation to the history.

[1] non-numeric value such as Strings, Booleans, Characters

During scheduling, the Scheduler generates a set of valid and eligible mappings for consideration (see Section 10.2). These mappings include not only the identifiers of the Nodes, Links, and Transformations in the mapping, but all the contextual information in each entity's associated User and System Context stores. The Scheduler passes each current mapping to the Execution Observer and requests a runtime prediction. The Execution Observer uses the contextual information in the current mapping, and the contextual information in all historical execution observations to predict the runtime of the current mapping. The Execution Observer uses a multi-step process for predicting the runtime of a Transformation; the steps are relevant context selection, model instantiation, and runtime prediction. The Scheduler and Execution Observer repeat this process for each mapping under consideration by the Scheduler.

9.3.1 Relevant Context Selection

When attempting to predict the performance of a given mapping, the Execution Observer first needs to find relevant contextual information to make the prediction. Relevant context selection begins when the Scheduler passes the Execution Observer a proposed mapping. This mapping includes all Contextual Identifiers in the User and System Context stores of all entities in the mapping (i.e., the set of all Contextual Identifiers that can describe the mapping). The Execution Observer then needs to find enough historical observations to build an accurate model for the mapping described by the Contextual Identifiers in the new map-

ping. Due to the large number of Contextual Identifiers that may be present in the new mapping, the Execution Observer may not have enough (or any) observations in the observation history that match this mapping exactly. The Execution Observer applies an iterative algorithm to discount contextual identifiers from the current mapping.

9.3.2 Context Filtering Algorithm

Given a proposed mapping, and set of historical observations, the Execution Observer must select a set of Contextual Identifiers that describe the proposed mapping such that it can model the mapping with historical observations. Historical observations may contain Contextual Identifiers not present in the proposed mapping, and vice versa. The Execution Observer uses a context filtering algorithm to select a set of observations from the history and Contextual Identifiers from the current mapping such that every observation in the set has Contextual Values for every Contextual Identifier. The Execution Observer uses Strings, Booleans, and other non-numerical Contextual Values in the context stores for relevance selection, but not for logistic regression. The context filtering algorithm begins by enumerating all Context Identifiers and their associated Contextual Values for every historical observation. Table 9.2 provides an example of raw contextual information stored in the historical observations. In order to handle nominal Contextual Values, the Execution Observer creates a new matrix to represent the presence or absence of different contextual information in each

	N1UCTransID	N1UCDataAmount	N1UCDatatype	N1UCLinkspeed
Obs1	"CP1"	5000	"-"	50
Obs2	"PROC1"	4500	"JPG"	-
Obs3	"CP1"	3500	"-"	50
Obs4	"CP1"	250	"-"	50
Obs5	"CP1"	150	"-"	75

Table 9.2

historical observation. The Execution Observer creates a column in the matrix for each numeric Contextual Identifier and a column for each nominal (Contextual Identifier → Contextual Value) pair. Table 9.3 illustrates this procedure.

	N1UCTransID: CP1	N1UCTransID: PROC1	N1UCDataAmount	N1UCDatatype: JPG	N1UCLinkspeed
Obs1	1	0	1	0	1
Obs2	0	1	1	1	0
Obs3	1	0	1	0	1
Obs4	1	0	1	1	1
Obs5	1	0	1	1	1

Table 9.3: Each Contextual Identifier is either present 1 or absent 0

Each observation (row) in the new matrix encodes information about the mapping under with the Transformation previously executed. The Execution Observer needs to filter this information in a way that retains as much of the information as possible. In the example of Table 9.3, the Execution Observer needs to reduce the table by dropping rows and columns until it has removed all 0s from the table. The Execution Observer makes repeated passes through the matrix; whenever it finds a 0, it calculates the information loss of dropping the associated Contextual Identifier (column), or the historical observation (row). The Execution

Observer calculates information loss by summing the rows and the columns, as shown for an N×M Matrix in equations:

$$\text{Information Loss for Col i} = \sum_{n=1}^{n=N} observations[n][i] \tag{9.2}$$

$$\text{Information Loss for Row j} = \sum_{m=1}^{m=M} observations[j][m] \tag{9.3}$$

The algorithm then drops either the row or the column depending on which incurs less information loss. When dropping row and the observation incur the same information loss, the algorithm drops the observation. This subtle decision means that when at least half of the observations come from a single Node, the algorithm will favor observations from that Node. Favoring a Node means that if there is enough information, Structant's observation algorithm will make predictions for a given Node, based only on historical observations from that Node. Also, if algorithm finds the Context Identifier "TransType" in the User Context store, the algorithm will never drop the column.

This algorithm may eliminate all usable context and prevent modeling the current mapping as a function of historical observations. This can happen if the algorithm drops all contextual information with numerical Contextual Values. This behavior indicates the absence of enough historical observations to model the current mapping accurately.

The following pseudocode shows the context filtering algorithm:

```
# mapping:
#  - 1xN matrix with Contextual Identifiers from the proposed mapping
# observations:
#  - XxY matrix with Observations (rows) and Context Identifiers (cols)
def context_filter(mapping, observations)

   for(column_id in mapping)
      for(obs in observations)
         # observation obs is missing context identifier column_id
         if(observations[obs][column_id] == 0)
            #Count how many observations contain this Context Identifier
            ci_points = sum(observations[*][column_id])

            #Count how many Context Identifiers this observation contains
            ob_points = sum(observations[obs][])

            # The context identifier contributes less information
            # Prefer to drop the observation
            if(ci_points < ob_points)
               #Drop the context identifier
               drop(observations[*][context_id])
            else
               #Drop the observation
               drop(observations[obs][*])
            end
         end
      end
   end
end

end
```

9.4 Modeling

After filtering relevant contextual information from historical observations, the Execution Observer attempts to build a model of execution behavior under different mappings. Prior work has attempted to model runtime prediction as a

function of attributes in the system [74]. Structant filters the contextual information in Table 9.2, to include only columns with all numerical observations.

9.4.1 Feature Selection

The Execution Observer treats each Contextual Identifier as a feature, and each row as an observation. It then augments each row with a column for runtime. In cases when the Execution Observer lacks enough features or observations to build a reliable model, it returns -1 to inform the Scheduler that the estimate is not reliable. The Execution Observer then builds a logistic regression model, and makes a prediction for the current mapping.

9.4.2 Data Staging

In cases where the Node does not have a local copy of the inputs to a Transformation, the Node will need to PreStage the data (if allowed) (see Section 11.7). The mapping passed into the Execution Observer includes the amount of data that the Node will need to stage. The Execution Observer computes the time required to PreStage the data based on the speed between the Nodes. In cases where the Node must PostStage the data, it performs the same computation using the Edge between the Nodes. The Execution Observer only calculates the PostStage estimate if the user has defined an IO Ratio parameter in the Transformation (see Section 11.8). To provide more precision in runtime estimates, the Execution Ob-

server will return -1 if it cannot compute either PreStage or PostStage estimates in cases where the Node will need to PreStage or PostStage.

9.4.3 Context Pollution

When using the Global Context API to modify User Context during operation, the user may inadvertently pollute the User Context store. Pollution occurs when the user fills up the context stores of an entity with contextual information that either becomes stale or does not facilitate runtime prediction. As the context stores become polluted, the Execution Observer has more difficulty making useful predictions. We have considered two solutions for this phenomenon. The first solution, which we implemented during testing, is simply to maintain a list of Context Identifiers to use in execution observation (which can be stored in User Context, so that the user can modify the values). While this solution works correctly, it requires extra effort from the user. Another solution considered, but not yet implemented is to limit the number of features used in model creation (NumFeatures = N), and then perform automatic feature selection based on the top N most predictive features.

Chapter 10: Scheduler

Structant's Scheduler receives contextual information from every other virtual component and interprets this information to infer the current state of all entities in the Computational Environment and make informed scheduling decisions based on processing requirements. At a high level, the Scheduler queries the Ontology for Tasks and Transformations that need scheduling, queries the Fabric Model for a list of available Nodes and Edges, consults the Global Policy Broker for a list of applicable policies, then uses the Execution Observer to predict how quickly each Computational Resource may be able to complete the Task or Transformation. Structant's Scheduler uses current contextual information during scheduling. This subtle aspect is important to observe because Structant's contextual models may change after the Scheduler maps a Task or Transformation to a Computational Resource in a way that causes the mapping to violate one or more policies.

10.1 Scheduling Challenges

Structant's scheduling goals introduce challenges not present in other distributed processing environments. Because of Structant's complex contextual models, the

ability for the user to interact with the models, and external sources of context, a mapping of Task or Transformation to Computational Resource may begin to violate a policy at any point during execution and get preempted. We refer to this phenomenon as the *Hidden Deadline Problem*. While prior work has focused on scheduling algorithms to avoid hard deadlines [85], Structant introduces new scheduling challenges because in addition to needing runtime estimates, the Scheduler does not know hidden deadlines at schedule time.

Similarly, prior work has utilized performance prediction in task scheduling [74]. Structant introduces new challenges in this domain as well. Unlike schedulers discussed in the prior work [8, 29], Structant has no information about the nature of the underlying program except for the metadata attached to the program by the user. The diversity of Structant's Computational Environment introduces additional challenges to runtime prediction. While prior work has examined the possibility of scheduling in heterogeneous computing environments, Structant operates in Computational Environments far more diverse than previously discussed [17, 21, 41, 67, 82, 86, 99].

Prior work has also attempted to recognize communication overhead when scheduling data-dependent jobs. Task duplication relies on the observation that if more than one job depends on the output of a given job, it may be faster to run the given job in more than one location so that antecedent jobs can access output data more quickly [11, 21, 79, 86]. Structant cannot use task duplication because Transformations may be non-deterministic. Running two antecedent Transformations on different versions of the output may result in downstream data corruption.

Another approach is to recognize data dependence and calculate data transfer overhead at scheduling. In Structant, the output size of a Transformation may not be known in advance (see Section 11.8)

Given these challenges, it would be infeasible to create an optimal schedule, or even one that guarantees completion. As a heuristic, Structant prioritizes long-running jobs that it predicts will complete without policy violation. Structant uses a greedy scheduling algorithm, meaning that it will always make the best decision at the current time, given the available information.

10.2 Generic Scheduling Operations

The following sections describe operations that Structant's Scheduler performs during scheduling. Structant schedules Task and Transformation differently, but uses the same generic operations.

10.2.1 Compatibility

Structant's Scheduler queries the Fabric Model to obtain a list of all Nodes and Edges to consider during this iteration of scheduling, and checks the resources for compatibility. Compatibility refers to the ability of the resource to physically perform the operation. The Scheduler performs the compatibility check using only information available in System Context stores [1]. The Scheduler performs the following checks to test for compatibility:

[1] User Context is not used in compatibility checks; it is only used in eligibility checks

1. **Correct Resource Type**

   The Scheduler first checks to ensure that the entity under consideration is capable of running the requested Transformation. When attempting to schedule an ordinary Transformation, the Scheduler queries the Fabric Model to ensure that the entity under consideration is a Node. If the entity is an Edge, the Scheduler discards the entity from consideration.

2. **Resource Availability**

   Next, the Scheduler checks to see if the entities will have access to the necessary resources at runtime. When attempting to schedule an "EXEC" Transformation, the Scheduler queries the Fabric Model to determine if the inputs listed to the Transformation are local to the Node. If the inputs are not local and the Transformation is not allowed to PreStage, then the Scheduler discards the Node from consideration. When attempting to schedule a "COPY" Transformation, the scheduler will query the Network Model to determine if the inputs or outputs listed to the Transformation are local to the Node. If neither the inputs nor the outputs are listed, then the Node is discarded. Similarly, Edges are discarded if the input data is not local to the source Node or the output data is not local to the destination Node. At this time the Scheduler does not allow Nodes to PreStage or PostStage "COPY" Transformations.

3. **Load Limits**

   Each Node and Edge in the Fabric Model have a predetermined number of

resource slots available for scheduling, similar to the MapReduce paradigm [45]. Slots for Tasks and Transformations are tracked separately and represent the total number of Tasks and Transformations that can be running on any resource at a given time. If a resource does not have an available slot for the Transformation, then the resource is discarded from consideration.

The list of resources left at the end of this operation represents the list of compatible resources.

10.2.2 Eligibility

Beginning with a list of all Computational Resources under consideration, the Scheduler checks all possible mappings for eligibility. Eligibility refers to the existence or absence of policies that either allow or prohibit the Scheduler from mapping the operation to the resources. Structant considers a mapping eligible if all relevant policies evaluate to "VALID" under the proposed mapping and prohibited if any policy evaluates to "INVALID". The Scheduler performs the following checks for each mapping under consideration to determine eligibility:

1. **Global Policy**

   The Scheduler checks that the mapping satisfies all global policies. Mappings which violate a global policy are removed from consideration.

2. **Local Policy**

   The Scheduler checks that the mapping satisfies the local policies of every

entity in the mapping. If any relevant Local Policy evaluates to "INVALID," the mapping is removed from consideration.

3. **User Policy**

   The Scheduler checks the User Policies attached to each entity. If the function specified in the User Policy evaluates to anything other than "VALID" under the proposed mapping, the mapping is removed from consideration.

The list of mappings that left at the end of this operation represents the list of eligible mappings.

10.2.3 Runtime Prediction

Given a list of mappings for a Computational Operation, the Scheduler queries the Execution Observer to obtain a runtime estimate for each mapping. Runtime refers to the total elapsed time from when the Dispatcher marks the Transformation's status in its Ontology to "PENDING" to the time it marks the status as "COMPLETED." For each mapping under consideration, the Scheduler passes the Execution Observer the identifiers of the entities in the mapping, and requests a runtime prediction for the mapping based on historical observations. For a detailed explanation of how the Execution Observer models and predicted runtime, see Section 9.3.

10.2.4 Forward Policy Check

Given a list of mappings and a predicted runtime for each mapping, the Scheduler checks all mappings for eligibility throughout the predicted runtime, plus a buffer of 10%. The Scheduler performs the following checks during Forward Policy Check:

1. **Global Policy**

   The Scheduler checks that no Global Policy will cause the mapping to become invalid during the execution. The Scheduler removes from consideration mappings that it predicts will violate a Global Policy at any point during execution.

2. **Local Policy**

   The Scheduler checks that no Local Policy attached any entity in the mapping will cause the mapping to become invalid during execution. The Scheduler removes from consideration mappings that it predicts will violate a Local Policy at any point during execution.

Notably, the Scheduler does not make forward policy checks against User Policy. Since User Policies may depend on external context or have complex temporal dependencies, Structant cannot predict whether a mapping will violate a User Policy with any reasonable degree of certainty. To discourage incorrect use of Forward Policy Checks, Structant does not attempt to check User Policies using runtime estimates.

10.3 Task Scheduling

Because Node selection does not occur during Transformation scheduling, Structant uses a more complex algorithm for Task scheduling. Once the Scheduler selects a Node to execute a Task, it must map all Transformations within the Task to that Node. When scheduling a Task, the Scheduler may select any Node that meets the following criteria based on the *currently available* contextual information:

1. **Compatibility**

   Every Transformation in the Task is compatible with the Node. This criterion enforces that Structant will not Schedule a Task on a set of resources that cannot complete all Transformations in the Task. Structant schedules "COPY" Transformations on the source Node; compatibility refers to the set of source Node, destination Node, and Edge in the mapping.

2. **Eligibility**

   The Node is eligible to run every Transformation in the Task.

3. **Forward Eligibility**

   The Node is expected to remain eligible to run all Transformations in the Task for the predicted makespan of the Task, if all Transformations in the Task have valid runtime predictions.

10.3.1 Task Scheduling Algorithm

Structant's Scheduler uses the operations discussed in Section 10.2 to perform Task scheduling. The following algorithm describes how the Scheduler uses the operations to select a Node on which to schedule a Task.

1. **Select All Available Nodes**

   The Scheduler begins the Task scheduling algorithm with a list of all Nodes in the Fabric Model.

2. **Reduce to Compatible Nodes**

   The Scheduler reduces the list of Nodes under consideration by checking the compatibility of every Transformation in the Task with each Node. The Scheduler removes from consideration any Node incompatible with any Transformation in the Task.

3. **Reduce to Eligible Nodes**

   The Scheduler reduces the list of Nodes under consideration by checking the eligibility of each Node to run every Transformation in the Task. The Scheduler removes from consideration any Node ineligible to run any Transformation in the Task.

4. **Compute Expected Runtime**

   For every Node under consideration, the Scheduler queries the Execution Observer to obtain a runtime estimate for every Transformation. If the Execution Observer cannot model the expected runtime for any Transforma-

tion, then the Scheduler considers the runtime estimate for the Node running the Task invalid. Otherwise, the Execution Observer returns the runtime estimate as E_n.

5. **Check Forward Eligibility**

   For every Node under consideration with a valid runtime prediction, the Scheduler checks that the Node will remain eligible to run every Transformation for the makespan of the Task. Because scheduling may cause the Transformations to execute serially, the Scheduler employs this safety precaution to reduce the risk of preemption. The Scheduler removes from consideration any Node which may not remain eligible.

6. **Candidate Selection**

   The Scheduler computes an adjusted runtime estimate by penalizing Nodes according to their load. Equation 10.1 shows the formula the Scheduler uses to compute adjusted runtime estimate.

$$ARE_n = E_n \times (1 + \frac{UsedTransformationSlots}{AvailableTransformationSlots}) \tag{10.1}$$

   If all Nodes have valid runtime prediction, the Scheduler assigns the Task to the Node with the lowest adjusted runtime estimate. Since the Execution Observer has a greater chance of making a valid runtime estimate for Nodes with a large number of historical observations, the Scheduler does not rely solely on runtime prediction for scheduling. In order to avoid starving new

Nodes which may not have valid runtime predictions, the Scheduler takes the following actions in order until a Node is selected, whenever one or more Nodes do not have valid runtime predictions:

- If possible, assign to the Node with the lowest adjusted runtime estimate with more than 50% Transformation slots available.
- If possible, assign to the Node with the lowest PreStage and PostStage time with more than 70% Transformation slots available.
- Number all available Task slots on all Nodes from (1,...,n)
- Randomly chose (1,...,n) and assign to the Node with the corresponding slot

This algorithm balances between selecting unloaded Nodes with known execution time and new, idle Nodes for which the Execution Observer cannot predict runtime. If all Nodes are under heavy load, the Scheduler randomly selects a Node based on available slots. It is important to note that Structant may not always make the best long term schedule. Randomly selecting a new Node to perform Transformations may give a suboptimal schedule, but yields critical observations for future runtime prediction.

10.4 Transformation Scheduling

Unlike Task scheduling, Transformation scheduling does not involve Node selection since the Scheduler must already have selected a Node to run a Task in

order to schedule a Transformation in the Task. During Transformation scheduling, the Scheduler only considers the Node that it has already selected to run the enclosing Task. In contrast to Task scheduling where the Scheduler tries to find the most suitable Node to run a set of Transformations, during Transformation scheduling, the Scheduler attempts to find the most suitable Transformation to run on a Node.

10.4.1 Transformation Scheduling Algorithm

The Scheduler uses the operations discussed in Section 10.2 to perform Transformation scheduling. The following algorithm describes how the Scheduler uses these operations to select a Transformation to run on a Node.

1. **Select Available Transformations**

   The Scheduler begins by selecting all Transformations in the Ontology available to run on the current Node. The Scheduler considers a Transformation available when all of the following criteria have been met:

   - The enclosing (parent) Task is bound to the current Node.
   - all of the Transformation's antecedents have statuses equal to "COMPLETED".
   - the Transformation's status is equal to "NEW".

2. **Reduce to Eligible Transformations**

   The Scheduler checks the eligibility of all Transformations to run on the current Node, and removes any ineligible Transformations from the list.

3. **Compute Expected Runtime**

   For every Transformation under consideration, the Scheduler passes the Execution Observer all User and System Context Stores for the proposed mapping and requests a runtime estimate. If the Execution Observer cannot create a runtime estimate for the Transformation [2], it informs the Scheduler to consider the runtime estimate invalid.

4. **Check Forward Eligibility**

   For every Transformation under consideration with a valid runtime estimate, the Scheduler checks that the mapping will remain eligible for the duration of the runtime estimate. The Scheduler removes from consideration any Transformation that will not remain eligible for the Transformation's makespan.

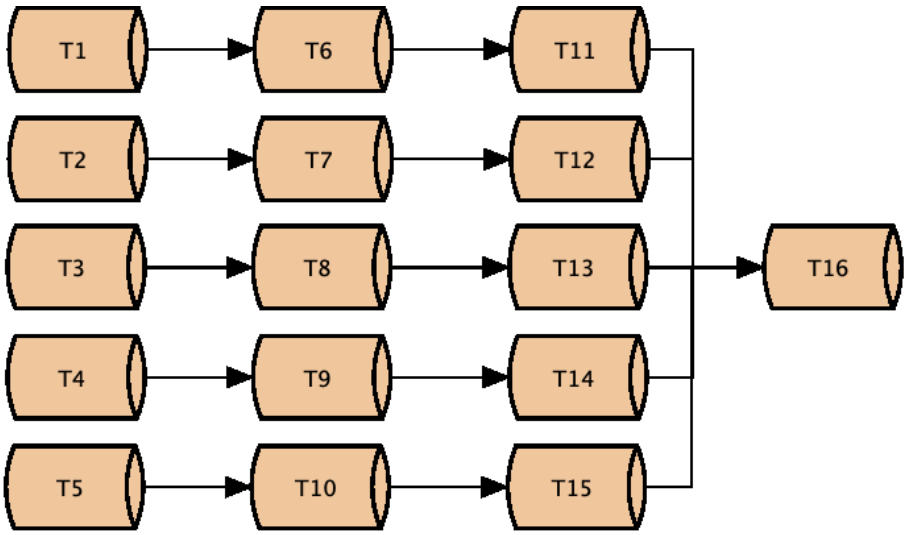
5. **Candidate Selection**

   In order to promote progress, the Scheduler prefers Transformation with the longest runtime estimate that it predicts will complete without violating policy. The Scheduler will schedule Transformations with invalid runtime estimates only when no Transformations with forward eligibility exist. When Scheduling Transformations without valid runtime estimate, the Scheduler will first select the Transformations with the fewest number of preemptions for policy violations, then the Transformation with the largest number of dependent Transformations.

[2]Most likely due to insufficient *a priori*

Chapter 11: Data Locality

When attempting to schedule Tasks and Transformations across different machines, Structant must address the issue of data availability. Consider the pipeline shown in Figure 11.1. In this pipeline, Structant must schedule the download-



Arrows in this diagram represent the data dependencies between operations. Arrows face from the operation producing a file a output to the operation needing the file as input.

Figure 11.1: Data Dependent Pipeline

ing of five different files from a remote file store (Transformations T1 through T5), extract each of the downloaded files, (Transformations T6 through T10), preprocess each extracted file (Transformations T11 through T15), and combine all the preprocessed files (Transformation T16). Transformations T6 through Trans-

formation T16 all contain virtual processing requirements because of their data dependencies. In this chapter, we discuss how Structant facilitates data flow in different Computational Environments to satisfy data dependence.

11.1 Shared Storage

Consider running the pipeline in a simple Computational Environment where all the Nodes running the Transformations have access to shared storage [1]. In this scenario, Structant does not need to facilitate data movement because all Nodes have access to shared storage. In other words, all Nodes in the cluster can immediately use the outputs of Transformation 6 through Transformation 16 via the shared storage. The Scheduler can select any Node in the cluster to run any Transformation in the pipeline without considering the availability of the data or the post Transformation movement of the data.

11.2 Local Storage

Consider the same pipeline running in a more complex scenario where each Node has its own local storage and no availability of shared storage. Scheduling Transformations in this pipeline becomes more complex because data for certain Transformations might not be locally available. For example, if Transformation T1 has run on Node A, the result resides on Node A's local storage. In order to schedule Transformation T6 on Node B, Node B must have access to the output of Trans-

[1] Storage shared by one or more Nodes over a high-throughput link

formation T1. Because of this requirement, some form of data movement must take place in order to schedule Transformation T6 on a Node other than Node A.

11.3 Grouping Tasks

The ability to group Transformations into Tasks allows the operator to specify certain properties about a group of Transformations. The first property is that a group of Transformations should be considered atomic. Defining a group of Transformations as atomic means that the output of any Transformation in the Task cannot be accessed by Transformations in another Task until all Transformations in the first Task have completed successfully. The second property is that a group of Transformations should be run on the same Node. While a user may group Transformations however he or she chooses (provided the pipeline is correct), grouping Transformations may become an arduous task for large pipelines. This raises the question of automatic Task grouping.

One might argue that Structant should group Tasks to minimize data movement. From this example, the following grouping may seem obvious: (T1,T6,T11), (T2,T7,T12), (T3,T8,T13), (T4,T9,T15), (T5,T10,T15), (T6). While this grouping should minimize data transfer in this case, it may also slow or inhibit progress. The following two examples explain why Structant does not perform automatic Task grouping.

11.3.1 Policy Based Pipeline Blockage

The problem statement dictates that Structant should attempt to achieve reasonable efficiency. As such, Structant tries make progress whenever possible. Each Transformation or Task may have Local or Global Policies that apply to it. Policies may dictate which Nodes may run the Task or Transformation. As we discussed in Chapter 8, these policies can have complex temporal relations. Suppose we modified this example slightly and Transformations (T1,T2,T3,T4,T5) could only run on a cluster with an IP address that falls within a certain range (for example on a University network) in order to download the data, while Transformations (T6,T7,T8,T9,T10) could only run on Nodes with Graphic Processing capability. If Structant were to automatically group the Transformations into the groups (T1,T6,T11), (T2,T7,T12), (T3,T8,T13), (T4,T9,T15), (T5,T10,T15), (T6), then the Scheduler could not find a mapping unless a Graphic Processing capable Node existing in the specific IP range. Since this processing requirement did not exist in the original pipeline, automatically grouping the Tasks resulted in a new (and possibly unsatisfiable) processing requirement.

One could additionally make the argument that Transformations could only be grouped if they had identical policies. This approach has a similar issue. Suppose that Transformation 1 through Transformation 15 make external requests across an egress link that charges per the amount of data accessed over a certain amount (i.e., the first 10Gb per IP address are free). A user could install a policy with PreCompute and PostCompute functions to ensure that the Scheduler

only permit Transformations to run on a Node if the aggregate data downloaded would be less than the 10Gb limit. If all fifteen Transformations required a 10Gb download, then the pipeline could take three days to complete because a Node scheduled to run the Task with Transformations (T1,T6,T11) would have to wait 48 hours for the rate limit to expire. This can occur even if all Transformations have identical policies.

11.4 PreStaging and PostStaging

In order to address the issue of data locality, Structant uses an automatic feature call PreStage and PostStage. PreStaging and PostStaging ensure that Nodes have access to data at runtime. Structant provides PreStaging and PostStaging as optional features that the user can disable if he or she has advanced knowledge of the Computational Environment.

11.4.1 PreStaging Overview

When a Task or Transformation requires PreStaging, it means that the Node running the Task might not have a local copy of the Transformation's inputs. In order to complete the Task or Transformation, the Node may need to obtain a copy of the input files. The Transformation specification may include the address of the Datastore Node that has a copy of the data. When the user does not explicitly define the Datastore Node, the Transformation will use the default Datastore Node. If user has not specified a default Datastore Node, the Dispatcher serves the de-

fault Datastore Node. When Structant assigns a Transformation to a Node, the Scheduler checks to determine if the Node has an up-to-date copy of the data readily available. If the Node ran a parent or sibling of the current Transformation, the Node may still have the files available. If the Node does not have a copy of the input data, the Scheduler instructs the Node to PreStage. When a Node PreStages data, it looks up the Datastore Node that has a copy of the Dataset, and downloads a copy of the Dataset to local storage.

11.4.2 PostStaging Overview

When a Task or Transformation requires PostStaging, it means that the Node running the Task or Transformation needs to publish a copy of the output data so that Nodes running antecedents of the current Task or Transformation will have access to the Dataset as inputs. The Task or Transformation may include an address for the PostStage Datastore Node where the Node should copy the output. When the user does not explicitly define a Datastore Node, the Transformation will use the default Datastore Node. If the user does not specify a default Datastore Node, the Dispatcher serves as the default Datastore Node.

11.5 Designing for Efficiency

PreStaging and PostStaging can cause unnecessary loads on Structant resources. While Structant uses techniques to reduce system load, such as preferring to schedule Tasks on Nodes which do not require PreStaging, the user can take

steps to promote efficiency in the system. The following three examples illustrate different ways that the Structant will behave with different specifications of the pipeline described above.

- **Example 1: The user defines sixteen ungrouped Transformations**: Structant will recognize the dependencies specified in the pipeline and schedule the Transformations in the correct order. Structant will PreStage and PostStage all Transformations to the default Datastore on the Dispatcher. Structant will schedule Transformations on any available Node, preferring Nodes with available slots that do not need to PreStage.

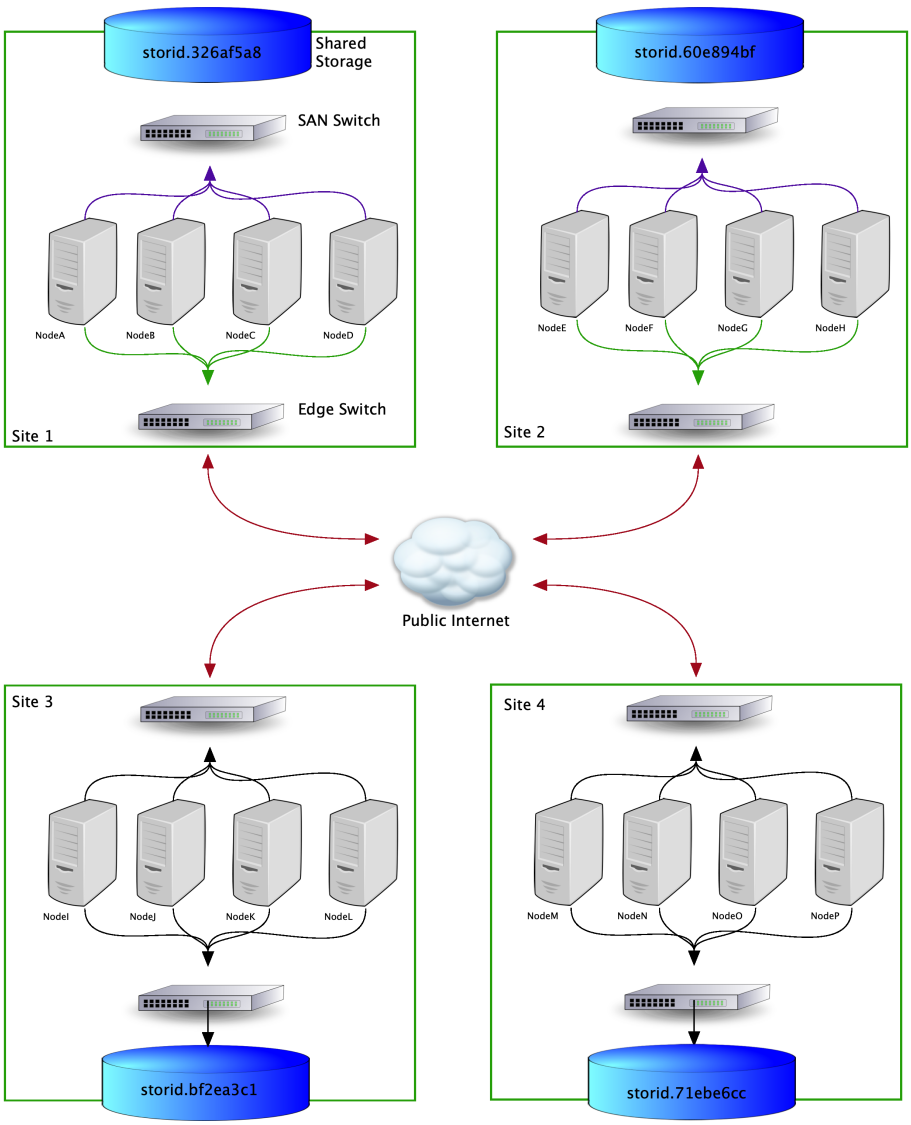
- **Example 2: The user specifies the groups (T1,T6,T11), (T2,T7,T12), (T3,T8,T13), (T4,T9,T15), (T5,T10,T15), (T6)**: Structant will recognize the dependencies specified in the pipeline and schedule Transformations and Tasks in the correct order. Structant will PreStage and PostStage after each Task to the default Datastore on the Dispatcher. Structant will schedule the Tasks on any available Node, preferring Nodes with available slots that do not need to PreStage.

- **Example 3: The user defines sixteen ungrouped Transformations, creates a User Policy to run all sixteen on the same cluster, and turns off staging**: Structant will recognize the dependences specified in the pipeline and schedule all Transformations in the correct order. Structant will schedule Transformations on any available Node in the same cluster, preferring Nodes with the lowest load. Structant will not PreStage or PostStage data since the user has specified that the data will be available to any Node that satisfies User Policy.

11.6 Structant Virtual File System

Structant tracks the location of files in the Computational Environment through the use of the Structant Virtual File System (SVFS). The SVFS uses a technique similar to database sharding [104]. The Fabric Model maintains a model of each Node's storage locality as well as the Datasets stored at each locality. Each Node can have either private storage directly connected to it, or it can share storage with other Nodes. All storage locations in Structant have a globally unique StorID that the Fabric Model uses to track the Datasets local to each Node in the Computational Environment. Transformations running on a Node can access Datasets stored on local storage without explicitly copying the file into working storage. When the Scheduler considers mappings during Schedule time, it consults the Fabric Model to determine the list of Nodes that have access to a StorID.

Consider the Computational Environment show in Figure 11.2. In this example, Nodes NodeA, NodeB, NodeC, and NodeD all have access to a file system mounted on a Storage Area Network (SAN), uniquely identified by *storid.326af5a8*. These Nodes can access Datasets on this file system without explicitly PreStaging data. Any other Node in the Computational Environment needing access to a Dataset on *storid.326af5a8* would need to PreStage data from NodeA, NodeB, NodeC, or NodeD.



The StorID uniquely identifies a Node's local storage. When a Node needs a file stored on a StorID other than its local storage, the Node will need to PreStage the data.

Figure 11.2: Storage Locality

11.7 PreStaging

When Structant parses a Transformation or Task description, it checks to see if the user has explicitly disabled PreStaging. If the user has explicitly disabled PreStaging, Structant sets the PreStage attribute in the Ontological representation of the

Transformation or task to "FALSE" so that the Transformation will not PreStage. If the user has explicitly enabled PreStaging, Structant sets the PreStage attribute in the Ontological representation of the Transformation or task to "TRUE" so that the Transformation or Task can PreStage [2]. If the user has not explicitly enabled or disabled PreStaging and has left the parameter blank or not specified it at all, then Structant will use the system default value for Transformation or Task PreStaging. By default, Structant will automatically PreStage both Transformations and Tasks if the user leaves the option blank. Like most behavior in Structant, the user can change this by setting a variable in the global configuration file. We chose this behavior for ease of operator use because it prevents the user from having to explicitly recognize data dependence. A user may wish to disable PreStaging a single Transformation for any number of reasons. If a Transformation will run on a Node where the operator knows that the data will exist in advance, then the operator may wish to disable PreStaging to speed up processing and reduce network bandwidth. This may occur when running Structant on a single machine or a cluster with shared storage. In this case, no Transformation would require PreStaging. This may also occur when Transformations are part of the same Task. In this case, if all the inputs of a Transformation come from antecedent Transformations in the same Task, the Transformations would have no need to PreStage since Datasets would already exist in local storage.

[2]The Transformation will not PreStage if the Node already has a local copy of the Data

11.7.1 Scheduling the PreStage

When a Transformation needs PreStaging, the PreStage operation occurs as a CopyTransformation. Before scheduling a PreStage, the Scheduler consults the Fabric Model to determine if a Node already has a copy of the inputs for the Transformation or Task. If the Node already has a copy of the input, then the Scheduler ignore the PreStage. The Dispatcher creates the CopyTransformation and dispatches it to the Workernode without updating the Ontology with this CopyTransformation. This design decision facilitates correctness. A CopyTransformation specified in the Ontology will always occur regardless of whether or not the files exist on the Node or the file store. The Node will always overwrite files in the destination of a CopyTransformation. PreStaging should not occur if the Workernode running the Transformation already has an up-to-date copy of the data. By not inserting an additional Transformation into the Ontology, the Dispatcher can dynamically decide whether the Workernode should PreStage data. Additionally, the decision of whether the data needs PreStaging can be more accurately represented and measured by the Execution Observer and used by the Scheduler.

11.8 Post-Staging

PostStaging happens similarly to PreStaging, but in the reverse. When Structant parses a Transformation or Task description, it checks to see if the user has explicitly disabled PostStaging. If the user has explicitly disabled PostStaging, Struc-

tant sets the PostStage attribute in the Ontological representation of the Transformation or Task to "FALSE" so that the Transformation will not PostStage. If the user has explicitly enabled PostStaging, Structant sets the PostStage attribute in the Ontological representation of the Transformations or Task to "TRUE" so that the Transformation will PostStage. If the user has not explicitly enabled or disabled PostStaging and has left the parameter blank or not specified it at all, then Structant will use the default value for Transformation or Task PostStaging. By default, Structant will automatically PostStage both Transformations and Tasks if the user either leaves the option blank or does not include it in the Task or Transformation description. Like PreStaging, the user can change the default PostStaging behavior by setting a variable in the global configuration file. By default, Structant will attempt to PostStage all Transformations and Tasks that have explicit outputs given. While this is probably unnecessary, it protects the user against forgetting to include relevant outputs in the Task description.

A user may wish to disable PostStaging on a single Transformation for any number of reasons. If a user knows that the outputs of a Transformation will only be read by Transformations running on the same Node, then the user may wish to disable PostStaging the Transformation. One plausible example of this occurring is when outputs from a Transformation are only used by other Transformations in the same Task. Since by definition, all Transformations within a Task will run on the same Workernode, Structant does not need to PostStage intermediate results to facilitate correct operation. A user may wish to disable PostStaging for the final Transformation or Task in a pipeline. If a user specified that the last Task

in a pipeline should not PostStage, then the user can check the logs to find the machine the Task finished on, and view the output file on that machine.

11.8.1 Scheduling the PostStage

When a Transformation needs PostStaging, Structant executes the PostStage as a CopyTransformation similarly to a PreStage. Like a PreStage operation, the Dispatcher creates the CopyTransformation and never enters it into to the Ontology. This design decision helps facilitate scheduling. Unlike the PreStage operation, a PostStage, if specified, will always occur because Structant assumes that Transformations always modify their outputs. Another dissimilarity with the PreStage operation is that the PostStage operation cannot be accurately estimated ahead of time. Structant has no method of determining the size of a Transformation's output in advance. From the available models, Structant has no way of knowing whether the Transformation will produce a Dataset larger than the input, about the same size as the input, or a single word answer. For this reason, Structant assumes that a PostStage will take around the same amount of time as a PreStage. In other words, Structant assumes that the inputs and outputs of a Transformation have identical sizes.

If the user has some advanced knowledge of the data pipeline, the user can specify a IO Ratio parameter in the Transformation or Task. This hint helps the Execution Observer calculate the expected cost of copying back the output to the shared Datastore Node. If the Transformation has no inputs, the use can specify

the expected size (in MB) of the output, in the IO Ratio parameter. As proposed expansion of Structant, a user may wish to include estimating this input to output ration automatically based on historical performance.

As a final consideration for PostStaging, the user should select a Datastore Node that has connectivity from all Nodes that the Scheduler may select to run dependent Tasks or Transformations. At this point, Structant assumes that all Nodes have connectivity to a Datastore Node. During scheduling, Structant does not check if a Node has access to the Datastore. In environment where the Nodes may not all have connectivity to a Datastore Node, the user should either manually specify data movement with CopyTransformations or PostStage to the Dispatcher.

Chapter 12: Node Interaction

The inability of users to interact with applications on Big Data stacks has been cited as a major hindrance preventing organizations from adopting large scale processing solutions [16]. Allowing the user to interact with the applications during execution has two major advantages. First, it aids with error correction and handling. Prior work has explored the possibility of rerunning parts of a directed acyclic graph given process failure [72, 75]. However, re-running operations may not correct certain types of error such as not finding a needed library or needing user confirmation before proceeding. Structant allows the user to decide how best to address an error at the point of failure. If the user can correct the error, then execution can continue and Structant does not need to reschedule the Transformation. If the user cannot correct the error, then the user can impose his or her knowledge on the situation by taking the appropriate corrective action (e.g., re-running, running on a different Node). Second, it allows the user to make decisions at key points in the program execution. Designers of cloud applications have begun to see the need for this ability [43, 66]. In iterative algorithms, the user can interact with the console on the running application to indicate when the program should stop running and to inspect intermediate results [50]. Maintaining

these connections allows the user to interact with Transformations and to serve as a key decision maker at critical points during execution (see Section 1.1.8).

12.1 Technical Overview

Structant facilitates interaction with running Transformations through the use of Ruby's POpen3 Library [1]. This section illustrates the inter-process communication techniques that Structant uses to multiplex console communication with Transformations running under multiple Workernodes. The following steps illustrate the procedure for communicating with a process running on a Workernode.

- **Bind the process's STDOUT and STDIN**: Using Ruby's POpen3 pipelining library, the Workernode executable starts the executable specified in the COMMAND attribute of the Transformation. The POpen3 library creates two pipes when starting the process - the first pipe is for STDIN to the child process, and the second pipe is for the child process's STDOUT. The Workernodes maintain two data structures to manage Transformation input and output. The first data structure stores references from the STDIN file descriptors to the Transformation associated with the file descriptor, and from the Transformation to the associated file descriptor [2]. The second data structure stores references from the STDOUT file descriptor to the Transformation associated with the file descriptor, and from the Transformation to the associated file descriptor [3].

[1] https://www.rubydoc.info/stdlib/open3/Open3.popen3
[2] The Ontology also stores a pointer to STDIN file descriptor
[3] The Ontology also stores a pointer to STDOUT file descriptor

- **Check if process is sleeping / stalled**: Periodically, the Workernode checks to see if any of its running Transformations are sleeping. The check is performed using Ruby's Thread library which provides the ability to check if a thread is either sleeping or waiting for input from the user. The Workernode then adds all Transformations with processes waiting for user input into a list of Transformations potentially needing user input.

- **Poll STDIN and STDOUT descriptors using IO.select**: Whenever the Workernode finds at least one of Transformations with a stuck process, it queries the Ontology to find the file descriptors for STDOUT associated with the sleeping Transformations. The Workernode polls the set of STDOUT sockets for sockets that are readable using Ruby's IO.select [4]. The Workernode records the output read from STDOUT separately for each Transformation.

- **Notify Dispatcher of sleeping / stalled process on the Workernode**: The Workernode then informs the Dispatcher of the stuck Transformations, and transmits associated STDOUT information.

- **Dispatcher's console informs the user that there is a sleeping / stalled process**: The Dispatcher then displays a message to the user, informing the user which Transformations on which Nodes need intervention. With each notification, the Dispatcher includes the name of the Workernode and the TransformationID. The user needs to utilize both of these attributes in order for Structant to route the response back to the correct process on the correct Node.

[4]https://ruby-doc.org/core-2.0.0/IO.html

Figure 12.1 illustrates the procedure for interacting with a process running on a Workernode.

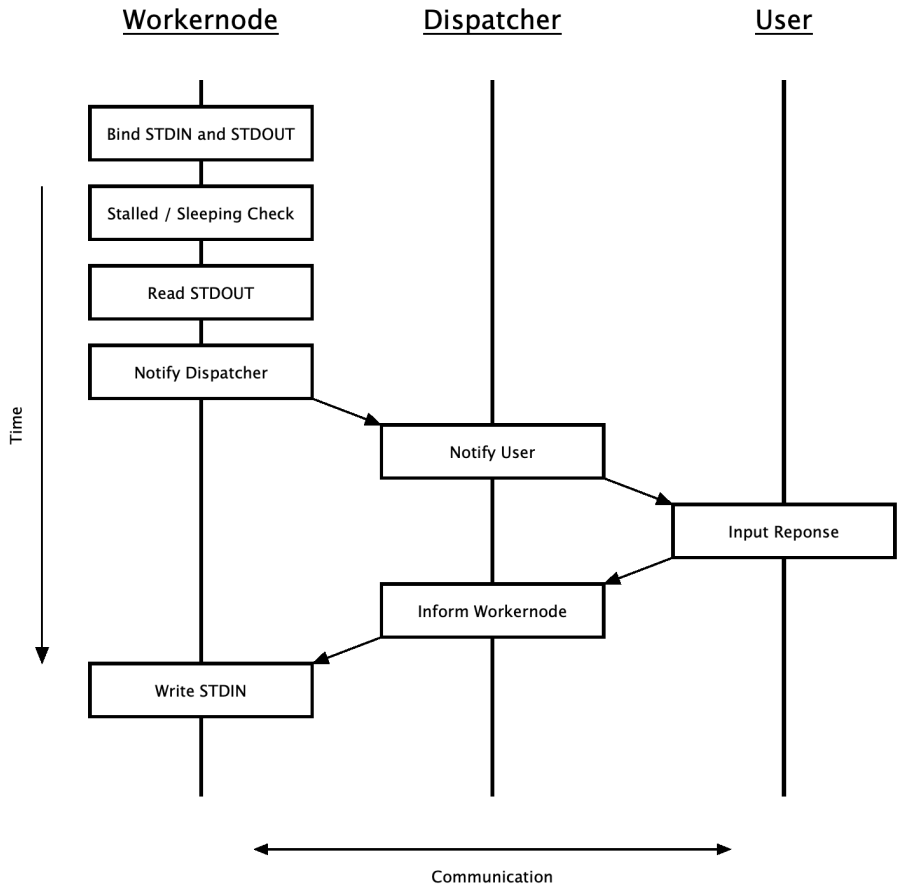


Figure 12.1: The user is able to read from STDOUT and write to STDIN on any process running on a Node in the Computational Environment via the Dispatcher

12.2 Error Correction

Pipelines running in production can often fail because of unsatisfied preconditions or post conditions of an executable. As a simple example, the Unix utility "unzip" [65] will stall, waiting for user input when attempting to extract a file if the destination file already exists. Unnzip does provide a command line option

to deal with this issue [5] by automatically updating or ignoring the exiting files. In order for this option to work, the user must remember to specify it and the user account running the program must have permission to delete or overwrite the existing file. Similarly, a program may need intervention if the executable cannot find a required library at runtime. While such a program may have an option for the user to manually specify the location of the library at runtime, a user may not think to enable such a feature if the program did not require it in the past. This situation could occur if a recent system update had rendered an existing library incompatible.

In some cases, allowing a pipeline to fail because of an error in one of the steps may be acceptable, but in other cases it may not. Some work has recognized this fact and allowed for mechanisms that re-run only selected portions of the pipeline [72, 75]. Time critical jobs such as image and data processing after natural disasters do not afford the user the flexibility to re-run an entire pipeline (or even part of it). Some steps in the pipeline can take upwards of six hours. If one such Task were to fail toward the end of the pipeline, marking the step as failed, and re-running it after fixing the preconditions or post conditions that caused the job to fail may still incur a delay of six hours in processing the data. After six hours, the data may be too stale for use and need to be recollected.

Structant takes a more active approach to dealing with pipeline steps that require user input. When a Workernode detects a stuck process (e.g., blocked waiting for user input), the Workernode sends a control message to alert the Dis-

[5] Specifying -u or -f will handle existing data

patcher to the presence of a stuck Transformation. In this control message, the Workernode includes the TransformationID of the stuck Transformation as well as the STDOUT and STDERR of the sub-process. The Dispatcher then gives an alert to the user indicating the presence of a stuck Transformation on a particular Workernode. The user then issues a command to the Dispatcher telling the Dispatcher to tell the Workernode to tell the Transformation how to proceed.

Handling errors in this way has several important benefits. The first benefit is that the missing information can be supplied to the running Transformation immediately. Structant's detection mechanism typically detects the presence of a stuck Transformation within a few seconds. The user can supply information to the Transformation to correct the problem, and then receive immediate feedback on the result. This eliminates the wait time of having to run the Transformation again with different parameters and then check to see if the error occurs again.

12.3 Decision Points

Structant affords users a high degree of flexibility by allowing the user to interact with programs at arbitrary steps during execution. Because the Dispatcher collects and presents information from each running Transformation separately, the user can interact with Transformations running across different Workernodes simultaneously. The Dispatcher identifies the Workernode and TransformationID associated with each STDOUT message. Giving the user this flexibility allows

Structant to introduce advanced features into traditional pipelines with minimal modification.

Structant provides the following method to interact with Transformations running on Workernodes:

- TELL <*NodeID*> TELL <*TransformationID*> <*msg*>: The first instance of the TELL command instructs the Dispatcher to deliver a message to the Workernode identified by NodeID. The Dispatcher will then deliver everything after NodeID to the node identified by NodeID. The second instance of the TELL command is executed on the Workernode. This command instructs the Workernode to deliver the message msg to the Transformation identified by TransformationID. The Workernode then writes the String msg to the file descriptor for the appropriate Transformation's STDIN file descriptor.

12.3.1 Hybrid Pipelines

Allowing the user to interact with running Transformations enables Structant to run "hybrid pipelines." Hybrid pipelines contain automated processing, but allow user input at certain steps. Such a pipeline may be desirable when a program requires a tremendous amount of preprocessing on input data or post-processing on output data. Creating such a pipeline in Structant requires no special configuration or changes to existing programs. When a user specifies executables in the pipeline that require user interaction in one or more Transformations, Structant will automatically detect that the running Transformations require user input and

inform the user. Hybrid pipelines are particularly useful in cases where the user needs to do some manual manipulation of data using some program in a way that cannot be coded, or when a user wishes to inspect preliminary results.

12.3.2 Controlling Execution

Although the current version of Structant does not allow the user to make specific decisions about the execution path and does not provide decision steps in the Ontology, the user can control the execution path in a primitive way by interacting with the Workernodes through the Dispatcher. Because the Transformations instantiate arbitrary executables, a user can create a Transformation that requires user input that will stage executables or modify the execution path downstream on the pipeline. Structant's Ontology lazily loads [6] only the name of the executable during instantiation. The Workernode never explicitly runs the executables, but passes the commands to the operating system's execution environment (see Section 6.2). This method of controlling execution demonstrates Structant's robustness by showing its ability to modify itself during runtime.

[6] Lazy loading is a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed

Chapter 13: Scenario 1

In this experiment, we demonstrate how Structant can facilitate and coordinate the collection, preprocessing, combining, and modeling of data obtained from disparate datasets by recognizing virtual processing requirements in the form of software and data dependence. This motivating example is based on a project in which we demonstrated that by applying well-known data-mining, machine learning, and prediction techniques to the domain of public health, we could perform forward predictions of drug-related death trends in specific communities. We have included this example to demonstrate Structant's ability to coordinate the collection and processing of data with environment specific requirements, while making minimal changes to the processing environment.

13.1 Problem Statement and Overview

We conducted this work as a preliminary investigation into the possibility of using machine learning techniques to make near real-time predictions of future drug epidemics as well as identifying communities and individuals at risk for drug use related injuries. The goal project was to assist public health, law enforcement, and medical professionals in identifying both geographic regions and

individual patients at risk of falling victim to the current opioid pandemic without modifying existing practices of law enforcement or medical professionals.

To achieve this goal, we framed the issue of opioid abuse identification as a data-mining problem and machine learning problem, and presented techniques for addressing the problem at a national and community level. The success of this work relied on the ability to collect, process, and join datasets from disparate sources. We relied on the experience of medical doctors, EMTs, and public health researchers to help give the data a basis for analysis by understanding the circumstances under which the data are collected, including legal requirements, political motivations, law-enforcement policy, and hospital procedures. In addition, this project relied heavily on data sources which were constantly ingressed from multiple locations in different formats.

The goal of this project was three-fold. First, we wished to apply current data-mining and machine learning techniques within the domain of public health. Second, we wished to build a framework for toxicovigilance by establishing an understanding of the relationships between existing datasets, identifying strengths and weaknesses of these datasets, and establishing techniques and procedures for combining disparate datasets in meaningful ways. Third, we wished to use this framework to build a system that augments existing medical and law-enforcement practices in order to help identify persons and communities in need of intervention as early as possible without changing any existing medical or law enforcement practices. This project focused primarily on the first and second goals at the national level and was able to demonstrate preliminary results.

13.1.1 Data Collection

In order to address the issues associated with a purely Big Data approach and create a more robust system that did not require manual annotation, we collected data from multiple, disparate data sources. In this work, we demonstrated a method to aggregate and use data from two different sources: the Centers for Disease Control (CDC) and the Federal Bureau of Investigation (FBI).

13.1.2 CDC Data Collection

We began by examining the data available through the CDC's Multiple Cause of Death (MCD) database. The CDC's MCD provides county-level data for mortality and population in the United States. The data in the CDC's MCD is based on death certificates for U.S. residents. Each death certificate contains a single underlying cause of death, up to twenty additional multiple causes of death, as well as demographic data about the deceased. The dataset offers information about the number of deaths, crude death rates and age-adjusted rate, as well as age-group, race, Hispanic ethnicity, gender, year and month of death, weekday of death, place of death, whether an autopsy was performed, and the four digit ICD-10 codes, for United States National, state, and county level. The underlying cause-of-death is defined by the World Health Organization (WHO) as "the disease or injury which initiated the train of events leading directly to death, or the circumstances of the accident or violence which produced the injury." Since May 23, 2011, all data queries for regions other than national data which contain be-

tween zero and nine deaths are suppressed in accordance with the CDC's privacy policy.

We used a guide published by the CDC [3] that defines common causes of death by drugs according to the ICD-10 underlying and contributing causes of death. The State Health Department defines the following categories of drug related deaths: Illicit Drug Poisoning, Pharmaceutical Poisoning, Prescription Opioid Poisoning, Other Pharmaceutical Poisoning, Illicit Opioid Poisoning and All Opioid Poisoning (illicit and prescription). For each category, the guide lists both the ICD-10 codes for the underlying and contributing causes under which the death may have been coded. Tables B.4, B.1, B.2, and B.3 show these categories.

To expedite the process of obtaining data as well as to reduce human error, we wrote an automated scraper that sent queries to the CDC's MCD database and processed the results. We scraped data for of all the categories mentioned above for each year between 1999 and 2015. In addition to scraping by year, we also scraped for quarterly data. We tried querying by month, but there was not enough data and too many of the rows were suppressed [1]. For each category and year, we obtained the results for each county in the database. In addition to the categories above, we also scraped data for each pair-wise data for each pair of underlying and contributing cause of death for all underlying and contributing ICD-10 code related to drug-related deaths.

[1] They were suppressed because of the CDC's Privacy Policy which suppresses regions with fewer than 10 deaths

13.1.3 FBI Data Collection

In addition to the CDC's MCD data, we collected data from the FBI's Uniform Crime Reporting (UCR) database [77]. The FBI's UCR is the FBI's attempt to create a uniform set of crime data for comparison across states and jurisdictions. The UCR program counts one arrest for each separate instance in which a person is arrested, cited, or summoned for an offense. The UCR program collects arrest data on twenty-eight different well-defined offenses. According to the FBI, the UCR offenses are divided into two separate groups, Part I and Part II. The Part I offenses include: criminal homicide, forcible rape, robbery, aggravated assault, burglary, larceny-theft, motor-vehicle theft, and arson. Participating law enforcement agencies are supposed to report information on the number of Part I offenses that become known to them; age, sex, and race are reported for Part I offenses. The Part II offenses include: simple assault, forgery and counterfeiting, fraud, embezzlement, stolen property, vandalism, weapons, prostitution and commercialized vice, sex-offenses, drug-abuse, gambling, offenses against family and children, driving under the influence, liquor laws, drunkenness, disorderly conduct, vagrancy, suspicion, curfew and loitering, and all other offenses. For Part II offenses, only arrest data is reported to the FBI as Part II offenses are considered less serious crimes. Because a person may have multiple arrest during a year, the UCR arrest figures do not reflect the number of persons arrested but rather the total number of arrests for a given area.

Because of variations in definitions of crimes across states as well as variations in local policy and reporting, we used the UCR as an *estimate* of the approximate crime rates of each state by year. Certain activities may be illegal in one state, but not in another. Similarly, an action might fall into one category in one state but a different category in another. To address this problem, the FBI publishes a handbook for state and local law enforcement to use when classifying entries to the UCR. Obtaining the FBI's UCR reports was simply a matter of automating the download of the various files from the FBI.gov website. The reports were probably hand generated, and were not consistently formatted across years; we had to write a script that would sanitize the reports and provide a consistent format.

13.1.4 Features Intro

After obtaining the data from CDC and the UCR, our goal was to combine and organize the data in such a way that we could use machine learning techniques to train models and make predictions on the data. We chose to predict the number of narcotics-related deaths in a given city for a given year. To accomplish this, we created a feature vector for each city. Each feature vector represented the data points for that city including the number of drug-related deaths for each year, the population for each year, and the number of arrests for each offense for each year. The decision to make predictions based on cities rather than counties was because most social media platform users list location by city, state rather than

county, and we planned to expand this work to include social media as a data source.

13.1.5 Feature Generation

The CDC reports statistics according to state/county while the FBI's UCR reports are based on jurisdiction. In addition, the CDC reports statistics per 100,000 persons, while the FBI reports raw counts. In order to simultaneously use data from multiple data sources, we needed to combine the data in a way that instances in each database represented the same type. Rather than try to map databases to each other, we generated a list of geographic areas we were interested in, and mapped all datasets to this list. Our list comprised of all cities in the US where the population exceeded 100,000 as of 2015.

Aggregating the data into bins based on city / state was not a trivial task. To the best of our knowledge, there was no existing work that discussed how to map counties to cities in a way that captured all the information we wanted. In order to demonstrate why this task was non-trivial and why it plays such a crucial role in our analysis, we present two straw man methods, which we immediately refute.

Straw man 1

Method: use a database to determine the county in which a city is located.

Problem: this method wastes a tremendous amount of data. Using this method,

the data points for counties surrounding a city (the suburbs of that city) are lost. We wanted our analysis to include not only the immediate city, but also the surrounding areas as that provides a more robust dataset.

Straw man 2

Method: use a map to manually classify counties as belonging to a city.

Problem: this method is time-consuming. There are over 3,000 counties in the CDC database alone and manually classifying these would take a tremendous amount of time. Additionally, this method is in-exact; it relies on personal judgment to make distinctions which may not be reproducible or consistent. Finally, this method does not permit change; if we wanted to change our algorithms or mapping definitions, all counties would require manual reclassification.

13.1.6 Geo-Mapping

Since the CDC aggregated data by county, we needed a way to map this to US city data. At first, we tried using a dataset from the United State Postal Service (USPS) to look up which city each county was in, and vice-versa. This technique presented several issues which made it unsuitable for our purposes. The first issue was the naming convention; the USPS data is based on the US Census data. Some cities were represented by multiple names (e.g., Pittsburgh and Pittsburgh Proper, and Pittsburgh Outlying Suburbs). We found that the MaxMind US City

database [1] was much simpler to work with and seemed to provide more accurate results.

The second issue was that several of the counties did not map directly to a city in our list. In order to address this and also to provide a more robust picture, we decided to map counties to the nearest city using three different distance metrics. We justify this by arguing that this will capture the features of not only the city, but of the nearby suburbs as well, which were of interest.

Mapping Counties to Cities

Our procedure is as follows: ssing the MaxMind US Cities database, we decomposed each city into a collection of ZIP codes and each county into a collection of zip-codes. For each ZIP code, we use the USPS database to lookup the latitude and longitude of the centroid. We then computed the distance between each city and county by calculating the average pairwise distance using the haversine formula between all pairs of ZIP codes in the county and the city. Our method returns three different metrics: the shortest distance between the county and the city (0 if they share a ZIP code), the longest distance between the county and the city, and the average distance between the city and the county. When used together, these three metrics capture the amount of overlap and approximate shape of the two regions.

In Equations 13.1, 13.2, and 13.3, $dist(Z_i, Z_o)$ refers to the haversine distance between the centroids of Z_o and Z_i as defined by the USPS.

$$\frac{\sum_{Z_o \in County} \sum_{Z_i \in City} dist(Z_i, Z_o)}{|County| \times |City|} \tag{13.1}$$

$$\min_{\forall Z_o \in County, \forall Z_i \in City} dist(Z_i, Z_o) \tag{13.2}$$

$$\max_{\forall Z_o \in County, \forall Z_i \in City} dist(Z_i, Z_o) \tag{13.3}$$

13.1.7 Sanity Checking

To the best of our knowledge, the literature does not contain prior work on geomapping counties to cities in this fashion. Since this methodology is largely untested, we decided to hand-test the results of our classifier using two different ground truths. While our method differs from the ground truth in both cases, the methodology of both ground truths account for the variances between our predictions and the ground truth.

Checking Against Hand-Coded Data

We obtained a list of CDC counties that geographers had mapped to Atlanta, GA for performing a similar analysis. Using this hand-compiled list as ground truth, we examined all the counties that our method mapped to Atlanta, GA and compared the results. Our method performed exceptionally well against the hand-

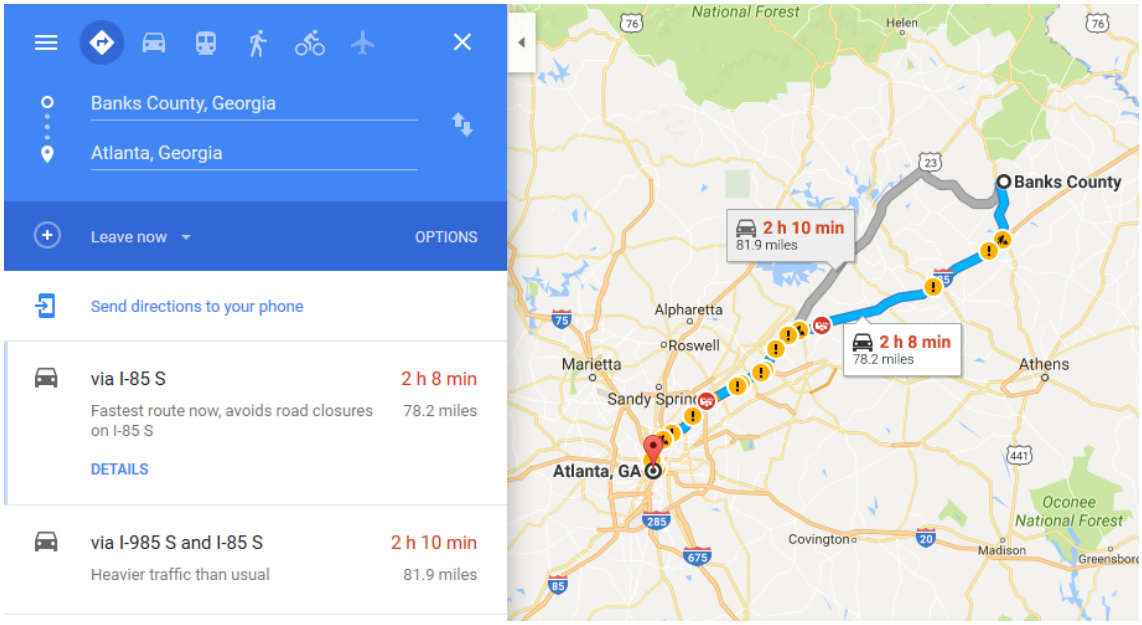
coded values. Our model of the geographic mapping contained more points than the hand-coded values produced by the geographers. The set of locations used by the geographers did not include the cities of Athens, GA; Macon, GA; Huntsville, GA; or Columbus, GA. In each case where our method mapped a city to Athens, Macon, Huntsville, or Columbus, we manually verified that the county was much closer to the mapped city than it was to Atlanta, GA. Table B.5 shows the comparison of our automatic geomapping algorithm with the hand-coded ground truth.

Checking Against Google Maps

In addition to checking against the hand-coded values, we checked results against Google Maps by selecting several county to city mappings and selecting a route in Google Maps. In general, our results tended to be about 10% to 15% shorter than Google Maps. Most importantly, our results were consistently shorter and by approximately the same scale factor. The most obvious explanation for this is because our algorithm reports a haversine distance between the centroids, while Google Maps accounts for roads. This can be seen in Figure 13.1.

13.1.8 Data Smoothing

In order to make the data compatible, we began by normalizing the data. The FBI UCR gave raw counts for each offense type, so we normalized this by the population for the given year to achieve offenses per 100,000 individuals. The CDC



Google Maps showing an indirect route from Banks County, GA to Atlanta, GA

Figure 13.1: Google Maps with an Indirect Route

MCD database included a normalized measure denoted by crude rate, which represented the number of deaths per 100,000 population. When we initially tried machine learning techniques on these values, the values were much higher than the values in the MCD. As a result, the machine learning techniques were ignoring the MCD values. To combat this, we scaled the MCD values by a factor of 100. MCD values were reported as deaths per 1000 individuals, and UCR values were reported as arrests per 100,000 individuals.

13.1.9 Results

Since we were interested in predicting early indicators of drug epidemics, we selected various classes from the dataset that we believed could be accurate indicators of drug activity in a given region and attempted to make predictions of

those class variables. From the FBI UCR data set, we selected: Drug Narcotic Violations, Drug Narcotic Offenses, Drug Equipment Violations. From the CDC Database, we selected the following classes: Illicit Drug Deaths, Prescription Opioid Deaths, Non-Prescription Opioid Deaths. In our dataset, each class was represented multiple times, once for each year 2011 through 2015. When attempting to make predictions for a class value of a given year (Y), we only used the class variables associated with the years up to Y-1. We fabricated this limitation in order to simulate and test our ability to make future predictions based on *a priori* data, rather than our ability to make retrospective predictions. In our terminology, a class variable is a specific property of a city, for example, "2015 Opioid Deaths according to the CDC"; a feature is a class variable used to predict another class variable in a given model. In order to demonstrate a specific property, we attempted some predictions using concurrent data (i.e., other classes from the same year). We used Weka [2] for both visualization and machine learning on the data.

One of the largest problems we faced was due to the size of the dataset. Of the cities that we were interested in, only about thirty-three had data from both the CDC's MCD and the FBI's UCR. This dataset was smaller than we would have liked.

13.1.10 Feature Selection

In addition to hand selecting features to use for prediction models, we also utilized Weka's automatic feature selection toolkit. In Weka, we used BestFit search method combined with CfsSubSetEval to select the feature set with the most predictive power. So as not to overfit the data, we limited the feature selection to ten features per model. In all cases, we hand-selected three features, and then used the top seven features from the feature selection as the remainder of the features. We had over 300 features to choose from, and for each class we attempted to predict, different features performed better. For the sake of brevity, the highlights of our findings are described below:

- **Recent Features**: Selecting features corresponding to years closer to the class value we were trying to predict and eliminating features corresponding to earlier year generally gave better results. This held true even in the combined dataset as the selection algorithms tended to prefer more recent crime statistics over older CDC MCD data.

- **Stolen Property**: By augmenting our features with crimes relating to theft and stolen property, we were able to improve our results.

- **Concurrent Data**: By using concurrent features (features from the same year as the class we are trying to predict), we were able to achieve the best results.

13.1.11 Predictions Baseline

In order to establish a baseline for our predictions and demonstrate the efficacy of including additional data sources, we first attempted to predict the number of deaths in 2015 for each drug related category for each reason using only historical information from the CDC. We chose to first make predictions without using the FBI data in order to evaluate whether using multiple data sources provides for more accurate predictions. We used Weka's linear regression tool with 10-fold cross validation to make predictions and evaluate the accuracy of the models. We attempted to predict Illicit Drug, Pharmaceutical, and Prescription Opioid poisoning for 2015 using the Illicit Drug, Pharmaceutical, and Prescription Opioid poisonings death values from 2011 through 2014. In all cases, we were able to build models that had high correlation, but also had high RAE, and MAE. One initial finding that we observed consistently across datasets, models, and predictions was that more recent data tended to have much more predictive power than older data.

Prescription Opioid 2015

Using Linear Regression with 10 fold cross validation, we obtained a correlation coefficient of 0.8124 using historical (2011 through 2014) data. We achieved MAE of 0.0047, which translates to predicting the actual number of deaths to within 4.7 deaths per 1,000 population.

Multifeature Prediction

When augmented with the FBI's UCR dataset, the number of features in the dataset grew to over 300. This was large compared to the number of observations we had for both the CDC and the UCR. As a result, Linear Regression performed poorly. Performing the same Linear Regression with 10 fold cross validation, we achieved a correlation coefficient of 0.44 with an MAE of 0.0086.

Perceptron Network

We chose to use a multi-layer neural network with fifty nodes and three hidden layers when making predictions on datasets with features from the FBI's UCR report. Due to the size of the dataset, it was simply not feasible to make linear regression perform meaningfully, as the algorithm constantly overfit the data. The neural network performed well on the dataset; we were able to achieve a correlation coefficient of 0.92 and MAE of 1.7 deaths per 1,000. For a baseline comparison, Linear Regression achieved a correlation coefficient of 0.88 with MAE of 2.2 deaths per 1,000 for the combined dataset after feature selection. These both show an improvement over the CDC only dataset. Due to the size of these data-sets, future work should include larger and more meaningful datasets.

13.1.12 Experiment Limitations

Despite the feasibility demonstrated by this initial work, there were several limitations that this work uncovered, as well as some surprising results. One unex-

pected result was that having more recent data greatly increases the accuracy of predictions. Using features from the same year as the class we were trying to predict improved the quality of the predictions. As can be seen in Figures 13.2, 13.3, and 13.4, there is a near perfect correlation between Pharmaceutical Poisonings and Prescription Opioid Poisonings from 2013 through 2015. Naturally, this suggests that more recent data will result in predictions that are more accurate and more precise.

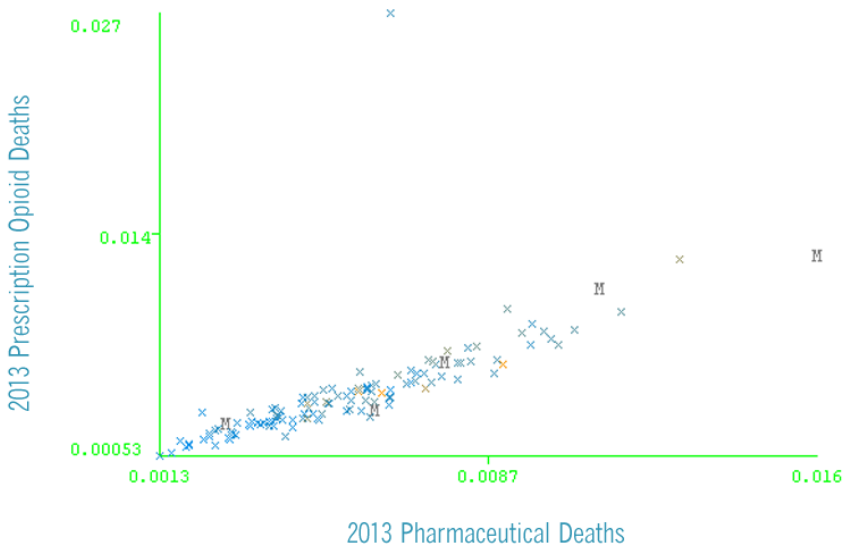


Figure 13.2: 2013 Pharmaceutical Deaths vs Prescription Opioid Deaths

One of the most enlightening observations from conducting this experiment was discovering how quickly the complexity of the pipeline for ingesting, preprocessing, and modeling the data grew. In what should seem like a simple experiment, the pipeline for processing the two datasets that we combined consisted of over ninety separate shell scripts, that all needed to be run in the correct order. Due to the vast differences in the types of data that we processed as well as the

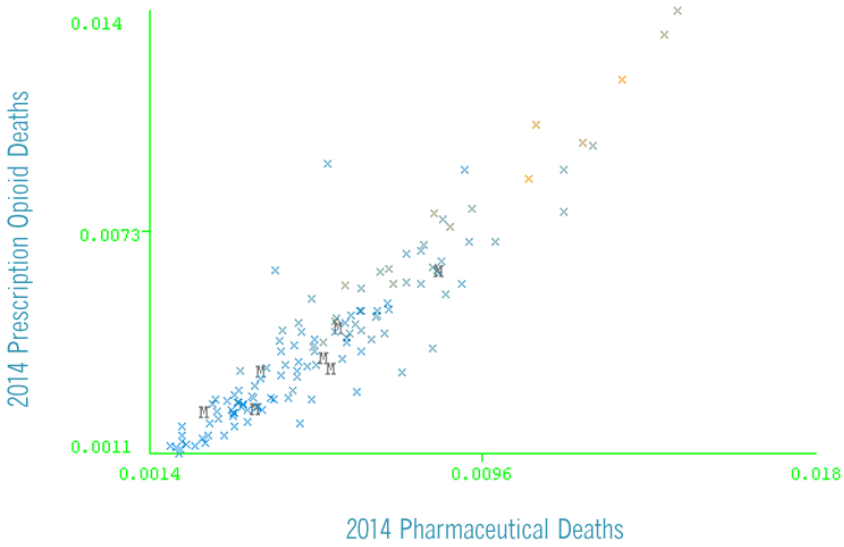


Figure 13.3: 2014 Pharmaceutical Deaths vs Prescription Opioid Deaths

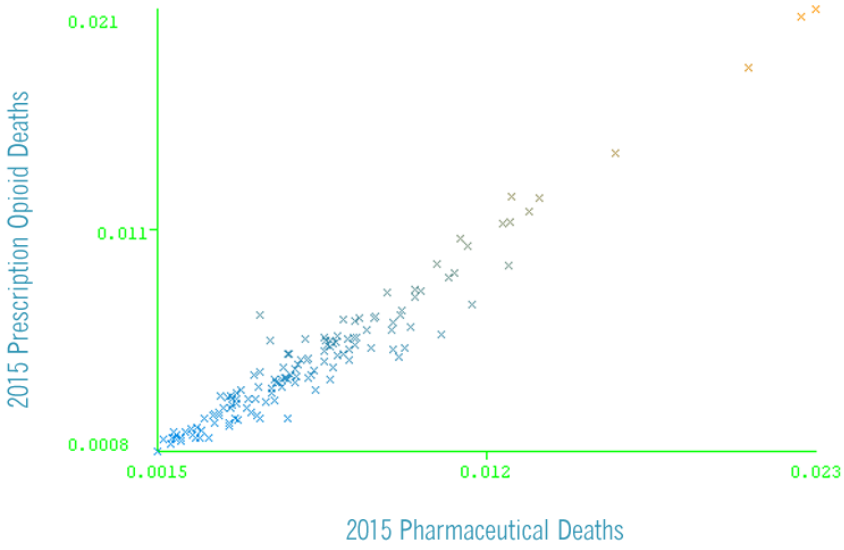


Figure 13.4: 2015 Pharmaceutical Deaths vs Prescription Opioid Deaths

computational goals at each step, the pipeline was written in four different programming languages: Ruby, Python, Javascript, and bash. Certain tools (such as the geolocation tools) were only available in certain languages, and certain steps required specific versions of languages. This often led to copying data between

machines. The original scope of the work included plans for ingesting, preprocessing and modeling data from Twitter. However, modifying the pipeline or even running the pipeline quickly became too time-consuming, so that portion of the experiment was dropped.

13.2 Methodology

One reason that we were able to conduct this experiment was due to the large number of open source gems and libraries available to Ruby and Python, respectively. Ruby provided a gem for working with Microsoft Excel documents (which is how the FBI UCR reports are published), and Ruby's mechanize gem proved indispensable for creating the scraping tool for the CDC MCD database. Similarly, Python provided tools for working with the MaxMind databases. These libraries and gems created dependency issues, however. The mechanize gem required an older version of Ruby that was not compatible with the gem we used for working with Microsoft Excel documents. Similarly, the Python version we needed to run the MaxMind database lookups required a newer version of Python than RHEL had in its package manager. As a result of this, we processed CDC MCD scraping on one machine, and geomapping on another. Whenever we needed to adjust the parameters we queried on the CDC's website, we had to run the pipeline again manually, and copy the data between the machines. While this may seem like a trivial inconvenience because we could have simply written a program to copy the data between the machines and run remote com-

mands, this problem is representative of a larger problem present in data processing. Data processing pipelines can quickly become complex to architect and maintain which consequently makes them costly to operate [16]. Although we developed only simple processing steps, our pipeline quickly became complex due to the processing requirements and data locality issues, and we had scripts executing commands on remote machines.

13.2.1 Using Contextual Information

Every processing step in the original pipeline had some set of requirements associated with it. These requirements comprised contextual information, because they were relevant attributes for the machines and operations that determined how they should interact with each other [4]. Facilitating correct pipeline operation could be done in two ways. The first technique was to embed the pipeline's logic at the processing layer [2]. In this technique, each step in the pipeline is responsible for calling the next step, and transferring data to the correct location for processing. The problem with technique is that the application is (implicitly) responsible for inferring the contextual information and the program is mixing contextual inference with the application logic. The second technique is to separate the contextual inference from the application logic [52]. In this technique, contextual information is stored in a well-defined and consistent manner, inferred by an oracle, and used to schedule processing. Prior work has shown that this tech-

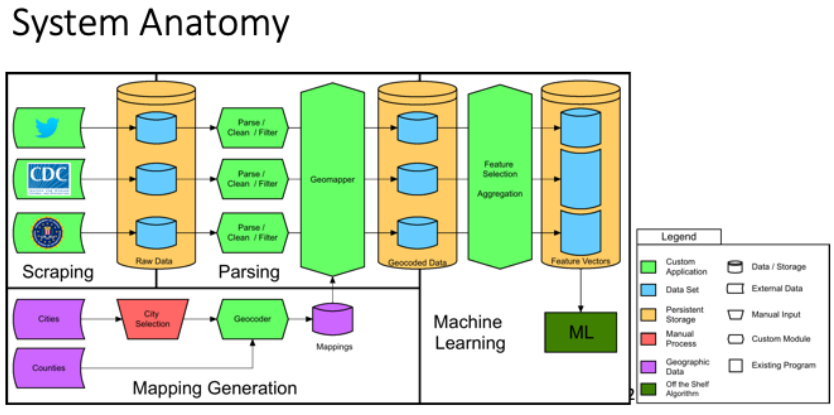
[2]Our original pipeline used this technique

nique is preferable because changes to the contextual information do not require modification of the pipeline itself.

Structant, as a context-aware system, employs the second technique. All dependencies and requirements are called out ahead of time (as contextual information), and stored separately from the application logic. Changes to requirements for a specific application or changes to the resources of a computational device [3] do not require any modification to the underlying pipeline.

13.2.2 The Original Pipeline

In the original pipeline, each step in the pipeline had one or more shell scripts associated with it. Data movement and task precedence made the pipeline difficult to fully automate, so we ran several steps manually. Figure 13.5 illustrates the steps in the original pipeline. In the figure, each step colored in green corre-



Overview of steps in our original data scraping and processing pipeline

Figure 13.5: Original Toxico Pipeline

[3] Updating the Ruby or Python version

sponds to a custom application that we wrote to scrape, preprocess, combine, and analyze data. The Twitter scraper used Python's Twitter streaming API Tweepy [4], the CDC Scraper was a custom application we wrote that used Ruby's mechanize library, and the FBI scraper was simply a collection of wget scripts. Each year of the FBI's UCR data had a separate parsing script associated with it to adjust for the fact that each year of the FBI's UCR reports differed slightly. The Geocoder was a custom Python script that we wrote using the MaxMind library, which read a copy of the MaxMind dataset, a list of the regions in the FBI UCR, a list of the counties in the CDC's MCD, and a list of cities of interest, then created a database mapping the FBI's regions and the CDC's Counties to a list from our cities of interest. The Feature Selection and aggregation step was a Ruby script that combined the geocoded FBI UCR and CDC MCD data into a single dataset.

13.2.3 Making a Structant Pipeline

Since we were working with an existing pipeline, each operation already had well-defined inputs and outputs. We further simplified the creation of the pipeline by copying all of our scripts to each Node (even though Nodes would not be able to run all the scripts). Our setup consisted of two Nodes, connected by a high speed Local Area Network link. We created the following User Context entries for each Node in the corresponding Workernode's configuration file.

```
1
2 # User Context Store (Node A)
3 UC_Node[A] = {"PyVersion":"2.7"}
```

[4] http://www.tweepy.org/

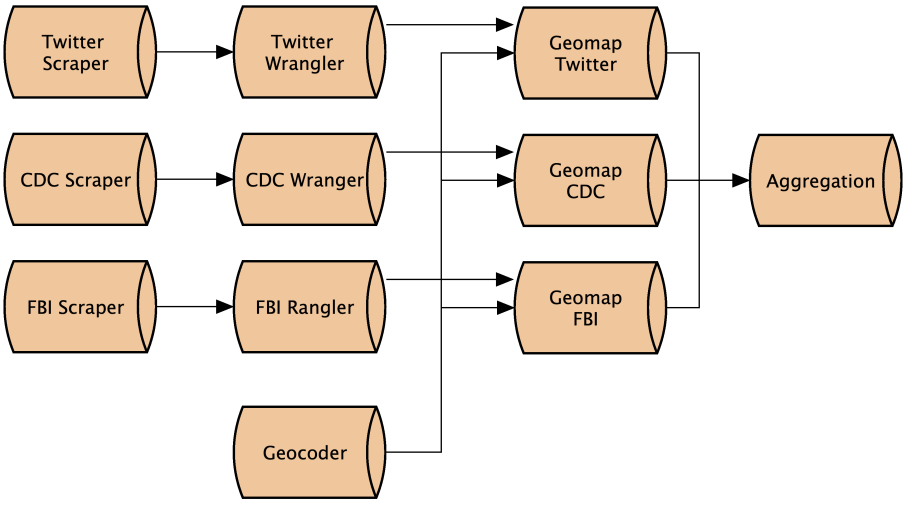
```
UC_Node[A] = {"RbVersion":"1.9.3"}

# User Context Store (Node B)
UC_Node[B] = {"PyVersion":"3.4"}
UC_Node[B] = {"RbVersion":"2.0.0"}
```

In order to correctly infer meaning from this contextual information, we created several local policies, which we attached to Transformations. These policies enabled Structant to interpret the "PyVersion" and "RbVersion" values in the Nodes' User Context stores. Correct interpretation was that the "PyVersion" and the "RbVersion" corresponded to the Python and Ruby versions installed on the Nodes, respectively. After creating policies, we specified a Task for each oper-

Policy	EntityType	Operator	Values	ContextLabel	FailOnMissing	Time
Py27	"NODE"	=	["2.7"]	"PyVersion"	TRUE	ALL
Rb19	"NODE"	=	["1.9.3"]	"RbVersion"	TRUE	ALL
Py34	"NODE"	=	["3.4"]	"PyVersion"	TRUE	ALL
Rb20	"NODE"	=	["2.0.0"]	"RbVersion"	TRUE	ALL

Table 13.1: These policies dictate which versions of Ruby and Python a Node must have to run a job.

ation in the pipeline. Since the machines had connection via a high-speed Local Area Network, and the Datasets were small, we were not concerned with data transfer, so we allowed each operation in the pipeline to be a separate Task with a single Transformation. Figure 13.6 shows the Tasks we created. For each Transformation, we defined the inputs and outputs, and allowed Structant to PreStage. We elected not to PostStage since we were not concerned with which machine had the final result. Lastly we attached policies to the Tasks as show in Table 13.2 It is worth noting that the policy "PyV27" was never used. We included this policy for consistency and to support future pipelines.



Overview of our original toxicovigilance pipeline as Structant Tasks with dependencies.

Figure 13.6: Toxico Pipeline in Structant Ontological Representation

13.3 Results

Using Structant reduced the complexity of the pipeline from over ninety independent bash scripts (some manually run across two different Nodes), to thirteen Transformations, four Local Policies, and two Workernode configuration files. Structant correctly recognized all the virtual process requirements and Task precedence, and scheduled Transformations on correct Nodes and in the correct order. No user interaction was required to operate the pipeline after we started the Dispatcher and Workernodes.

Table 13.3 shows an execution trace for the pipeline used in this experiment on one iteration. We allowed Structant to PreStage data in this pipeline, but as shown by the execution trace, Structant tried to avoid PreStaging as much as possible. Structant assigned the Twitter scraper to Node A randomly (see Section 10.3.1).

Task	Policies
Twitter Scraper	-
CDC Scraper	Rb19
FBI Scraper	-
Twitter Wrangler	-
FBI Wrangler	Rb20
CDC Wrangler	-
Geocoder	Py34
Geomap Twitter	-
Geomap CDC	-
Geomap FBI	-
Aggregation	-

Table 13.2

Time	Operation	Node
T1	Twitter Scraper	Node A
T1	CDC Scraper	Node A
T1	FBI Scraper	Node B
T4	Twitter Wrangler	Node A
T2	CDC Wrangler	Node A
T3	FBI Wrangler	Node B
T4	Geocoder	Node B
T6	Geomap Twitter	Node A
T6	Geomap CDC	Node A
T6	Geomap FBI	Node B
T7	Aggregation	Node A

Table 13.3

The Scheduler assigned the CDC Scraper to Node A due to policy "Rb19" and the FBI Scraper to Node B since Node B had available slots. Both Node A and Node B were compatible and eligible (see Section 10.2) to run the Twitter Wrangler and the CDC Wrangler, but Node A already had the data, so it would not have needed to PreStage. Node A was compatible with the FBI Wranger, but it was not eligible due to policy "Rb20", so the Scheduler assigned Node B the Task. Node A was also compatible with the Geocoder, but it was not eligible due to policy "Py34", so the Scheduler assigned Node B the Task. Node A and Node

B were both compatible with Geomap Twitter, Geomap FBI, and Geomap CDC; Node A was assigned Geomap Twitter, and Geomap Twitter because it did not need to PreStage, and Node B was assigned Geomap FBI because it did not need to PreStage. The Scheduler assigned Node A the Aggregation Task because the Twitter Dataset was local to Node A, and the cost of PreStaging the Twitter data was (slightly) more than the cost of PreStaging the FBI and CDC data to Node B.

13.4 Conclusion

This experiment demonstrates Structant's ability to schedule steps a data processing pipeline across multiple machines with respect to virtual processing requirements. Using Structant greatly simplified deploying and running the pipeline, as it replaced the task of manually managing data flow and program execution with the task of entering a small amount of contextual information and three simple policies. In addition to simplifying the user experience, Structant also automatically parallelized the processing of the data. Through Task dependency recognition, Structant allowed non-dependent operations to run concurrently. By correctly accounting for data locality through the use of PreStaging and PostStaging operations, Structant maintained data locality automatically, and transferred data between Computational Resources only when necessary.

Chapter 14: Scenario 2

In this experiment, we demonstrate that by creating simple policies and entering a small amount of contextual information in User Context stores, Structant has enough information to satisfy organizational processing requirement and use runtime prediction and forward policy check to avoid preemption. This experiment highlights Structant's Execution Observer, Policy Broker, and Scheduler.

14.1 Problem Statement and Overview

We conducted this experiment with data, information, and organization polices from Telligent Masonry LLC, a local small business in Rockville, MD. Telligent agreed to participate in this experiment by allowing us to use their network, corporate policies, computational goals, and at-rest data statistics. Their corporate network comprises on-premise file servers that store corporate data including financial records, HR files, architectural drawings, 3D models, and job-site photos. We designed this experiment to test if Structant could orchestrate an off-site backup of their data, as quickly as possible, without interfering with day-to-day user operations. Due to the physical location of their office, the only Internet Service Provider (ISP) available offered a maximum upload speed of 25 Mb/s. The

file server contained nearly 1500 GB of files of varying sizes that they needed to copy to an off-site backup as part of a disaster response plan. Due to the limitations on upload bandwidth, running the backups during business hours, even at a throttled speed, impacted the performance of the network for users working remotely via Virtual Private Network (VPN). Figure 14.1 illustrates the Computational Environment of the experiment.

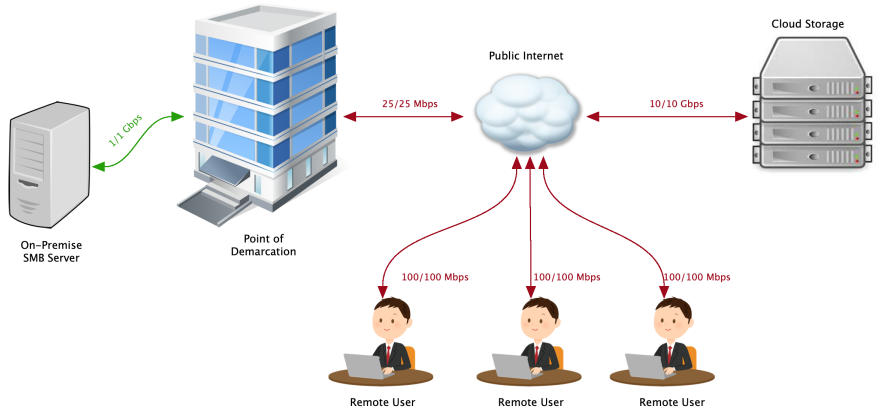


Figure 14.1: Remote users and backup operations sharing a single link from the on-premise file server to the public Internet

14.1.1 Methodology

Due to the length of time it would take to run the backup on-site, we did not conduct the entire experiment on premises. Instead, we profiled the Computational Environment and ran a smaller version of the experiment in the Navy Research Lab's CORE emulator [12].

14.1.2 File Size Distribution

In order to create a realistic distribution of file sizes for the experiment, we ran a script on the file server to obtain the full name of each file, including the path, and size in KB. We chose a realistic distribution because we wanted to test the efficacy of Structant on real data rather than a contrived Dataset. We surmised that, with an abundance of small files, Structant would not perform well since the risk associated with violating a policy is small, the link utilization is minimally affected since re-tries are not expensive, and Structant's profiling and model synchronization does induce additional computational overhead. Similarly, a truly random distribution would likely give Structant an unfair advantage since Structant could neatly pack the file sizes into the available time slots.

From the distribution, we grouped the file sizes into bins by 100 MB increments. To obtain the number and sizes of files for the experiment distribution, we multiplied the count of files in each bin by the ratio of the distribution we were trying to create, and created a number of random files equal to the result, over a uniformly even distribution within the range of the bucket. For example, to create a 5% sample size experiment data, we multiplied the count of each bucket by 0.05, and then generated a number of files equal to the resulting size. The random files were generated using OpenSSL on Red Hat Enterprise Linux (RHEL) 7.0.

14.1.3 Context

We copied the files using scp. We chose this method to demonstrate Structant's ability to schedule and profile operations without the use of the Link Speed parameter of the entities in the Network Model. All Transformations consisted of a single call to scp on the command line. Although the file servers at Telligent Masonry ran Windows, we conducted the experiment on Unix. We considered this reasonable because we found a compiled version of scp that ran on Windows.

We created each Transformation as an "EXEC" Transformation. For each Transformation, we defined pieces of context ("TRANSTYPE" $\rightarrow$ "SCP1"), and ("FILESIZE" $\rightarrow$ "FSIZE"), where file size corresponded to the file size (in MB), in each Transformation's User Context. To simplify this procedure, we used the scripting language we developed during testing. Since we used "EXEC" Transformations, the "FILESIZE" Contextual Identifier had no explicit meaning associated with it. Throughout execution, the Scheduler, Policy Broker, and Execution Observer inferred meaning from this contextual information by modeling how these values affected runtime, and scheduling Transformations based on this prediction.

14.1.4 Special Considerations

We had to remove files from the distribution that would not copy within a sixteen-hour time frame. These files interfered with the two naive scheduling algorithms. We discuss this in the conclusion of this experiment.

14.1.5 Naive Scheduler 1

Naive Scheduler 1 was a Longest Running Job First (LRJF) scheduler. Since the naive schedulers did not have access to runtime predictions, we implemented the LRJS scheduler to select the largest remaining file at each iteration. If the copy operation had not completed at the start of business hours, our scheduler preempted the copy any recorded its progress.

14.1.6 Naive Scheduler 2

Naive Scheduler 2 was a random scheduler. At each iteration, the Random scheduler randomly selected a remaining file and scheduled the copy. If the copy operation had not completed at the start of business hours, our scheduler preempted the copy and recorded its progress.

14.1.7 Structant Policies

We attached a single policy to the Node running the Transformations that forbid the execution of an "SCP1" Transformation during normal business hours (08:00 to 17:00).

14.1.8 Tracing File Copy

In order to conduct a reproducible experiment, the schedulers did not execute the copy operations directly. Instead, we created execution traces by executing all the copy operations in the network emulator. For each copy operation, we denoted

the file size, and the actual time that the emulator took to copy the file (accounting for delay and bandwidth). To simulate each copy operation in our experiment, we replaced each copy operation with an executable that would sleep for the amount of time that the corresponding copy operation had taken to complete in the emulator. This allowed us to compensate for variances in file transfer time between experiments since each scheduler was scheduling operations that took the exact same amount of time with the same amount of information about the operations.

14.2 Results

In order to better understand the behavior of Structant's profiling and scheduling, we conducted the experiment with three different distributions. The first distribution was a manually designed distribution with numerous small files. The second distribution was the distribution that we sampled from Telligent's file server. The third distribution had file sizes chosen uniformly at random. We found that Structant's Scheduler achieves more efficient orderings when expected runtimes have greater variance.

14.2.1 Worst Case Distribution

We chose the distribution shown in Figure 14.2 in order to demonstrate that Structant's algorithms will not perform worse than either naive algorithm even with a difficult case. This distribution is hard for Structant's algorithms to optimize for

two reasons. First, the uniformity of the file sizes does not allow much flexibility when scheduling. Second, the files are small so there is less of a chance that an individual copy is preempted, and preemptions have a much smaller impact on the channel utilization since the retransmission amount is small. In the uniform

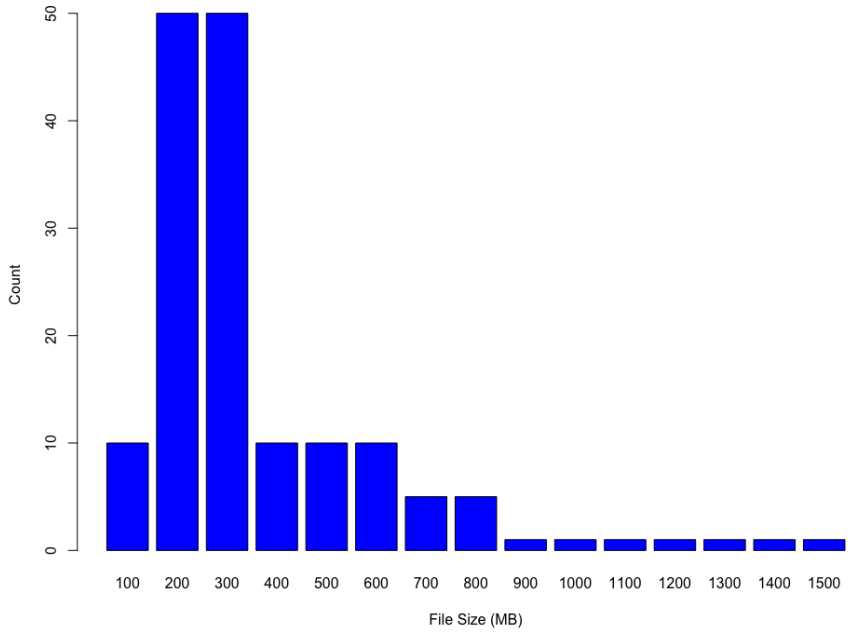


Figure 14.2: Worst Case File Distribution

small file distribution, Structant managed to achieve better channel utilization than either the Longest First or the Random scheduler. Interestingly, Structant's algorithm resulted in a longer overall runtime than the Random scheduler, even though the Random scheduler was preempted six times whereas Structant was never preempted. This is an artifact of Structant's forward policy checks (see Section 10.3.1). When Structant performs a forward policy check, the Scheduler automatically adds a buffer of 10% to the runtime prediction. This safety feature

accounts for the inherent imprecision in runtime prediction and tries to prevent preemption of long-running jobs by not scheduling too close to a policy which would cause preemption. In this experiment, the Random scheduler achieved slightly faster runtime by scheduling more aggressively, however, Structant used the channel more efficiently. Table 14.1 summarizes the results of this experiment.

Metric	**Longest First**	**Random**	**Structant**
Total Files Copied	157	157	157
Scheduler Preemptions	6	6	0
Wasted Time	6 Min	4 Min	0 Min
Progress Time	265 Min	265 Min	265 Min
Total Time	391 Min	388 Min	390
Channel Efficiency	97.57%	97.86%	100%

Table 14.1: Results summary for worst case data copy **Note**: Total Time includes copying, retransmissions and waiting for policy to allow copying.

14.2.2 Telligent Distribution

The file distribution obtained from Telligent was close to the worst case distribution that we manually created. Structant achieved 3% better bandwidth utilization than the longest job first scheduler and 6% better bandwidth utilization than the random scheduler. Out of the 1085 files in our experiment, Structant's copies were only preempted for policy violation two times. File copies scheduled by the LRJF and Random schedulers needed preemption 34 and 35 times respectively.

Metric	Longest First	Random	Structant
Total Files Copied	1085	1085	1085
Scheduler Preemptions	34	35	2
Wasted Time	38 Min	67.5 Min	2 Min
Progress Time	1338 Min	1338 Min	1338 Min
Total Time	2056 Min	2105 Min	2046 Min
Channel Efficiency	97.23%	95.18%	99.85%

Table 14.2: Results summary for data copy with Telligent's file size distribution **Note**: Total Time includes copying, retransmissions and waiting for policy to allow copying.

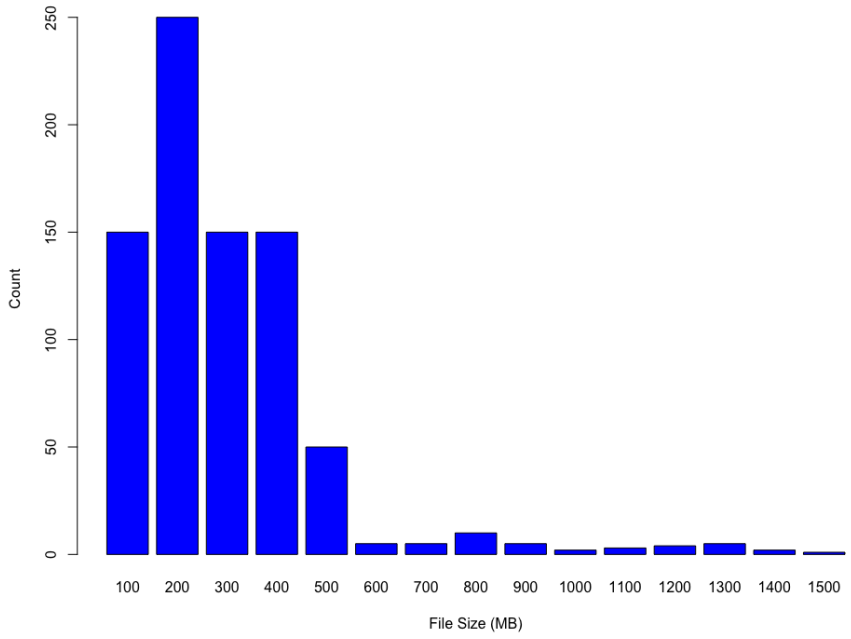


Figure 14.3: Telligent File Distribution

14.2.3 Random Distribution

Structant performed exceptionally well on the random distribution, achieving more than 11% improvement over both naive schedulers. Both naive schedulers wasted more than one hour copying files that resulted in retransmissions, whereas Structant spent no additional time copying files that it would later have to retransmit.

Metric	Longest First	Random	Structant
Total Files Copied	100	100	100
Scheduler Preemptions	16	16	0
Wasted Time	78 Min	82 Min	0 Min
Progress Time	597 Min	597 Min	597 Min
Total Time	995 Min	999 Min	911 Min
Channel Efficiency	88.33%	87.81%	100.00%

Table 14.3: Results summary for data copy with a random file size distribution **Note**: Total Time includes copying, retransmissions and waiting for policy to allow copying.

14.3 Conclusion

This experiment demonstrates that Structant's contextual inference techniques provide it with enough information to satisfy organization processing requirements correctly. While Structant's Scheduler incurred far fewer preemptions than either naive scheduler, we do not make the argument that Structant provides a better scheduling algorithm for the generic case. Rather, we make the argument that Structant's contextual inference help it to facilitate Task scheduling in a way that correctly recognizes organizational processing requirements. Both naive schedulers completed the same Task, but caused far more violations of organization policies.

By correctly inferring contextual information, Structant could, in this case, improve channel utilization and copy speed. For all three file distributions, Structant achieved higher channel utilization than either of the naive schedulers. In the worst case, Structant completed two minutes after the Random scheduler, but used bandwidth more efficiently. This type of scheduling behavior has advantages in environments with a monetary cost associated with usage of the Compu-

tational Resources. By scheduling more aggressively, the Random scheduler took more chances scheduling close to the deadline, and in some cases those risks paid off. The Random scheduler did, however, waste more resources to achieve this speed. Structant's scheduler could emulate this behavior by removing the 10% buffer during forward policy checks.

When copying the random file distribution, Structant performed exceptionally well over the two naive schedulers. We offer two explanations for this occurrence. Intuitively, the naive schedulers have more "bad" options available to them. For example, assume that the times required to copy the files have a uniform distribution between one second and the entire allowed copy time; When trying to select a file to copy halfway through the allowed time period, the LRJF scheduler will always select a file copy that will result in preemption. Similarly, the Random scheduler will have a 50% chance of choosing a file copy that will cause preemption. In contrast, Structant's selection of a file to copy, will only result in preemption when the Execution Observer makes an incorrect runtime estimate.

As mentioned earlier, we had to remove files that would take too long to copy in the allowed window. These files presented tremendous problems for the naive schedulers. The LRJF scheduler made no progress, trying repeatedly at every iteration to copy a file that it could never successfully copy. The Random scheduler would randomly select one of the files that could not copy; when it failed, it chose another file. As the number of remaining files decreased, the Random scheduler chose these files more often, adding a tremendous amount

of overhead to the copy due to the repeated failed attempts. When Structant encountered the large files, the Dispatcher warned the user that the policies prevented the pipeline from making progress.

The results of this experiment illustrate that by using only a small amount of contextual information, Structant can automatically recognize the interrelationships between User Policies, and the Performance of Transformations on Nodes.

Chapter 15: Scenario 3

In this experiment, we demonstrate Structant's ability to obtain contextual information from an external source, infer meaning from this contextual information, and use this information to adapt the Computational Environment. Environmental attributes are also an important source of context for the Computational Environment. Government and regulatory agencies publish guidelines that dictate the physical properties of a server room that an organization must maintain if they wish to remain compliant.

15.1 Problem Statement and Overview

The National Credit Union Administration has mandated that federally-insured credit unions take reasonable measures to protect member information from destruction due to "physical hazards and technological failures" [7]. Credit unions that wish to remain federally insured must undergo annual review by an NCUA examiner to verify that such measurements are in place. NCUA examiners have frequently interpreted this requirement to mean that credit unions should employ some form of environmental monitoring device to alert system administrators to server room operating conditions that fall outside of the acceptable parameters.

While current solutions exist, Structant provides a more robust solution to this requirement by granting system administrators the ability to create fine-grained and specific reaction sequences that occur when the properties of the physical Computational Environment fall outside of an acceptable range. In this experiment, we pose the issue of maintaining a safe Computational Environment as a context aware scheduling problem, and demonstrate how Structant can schedule operations to stabilize the Computational Environment.

15.2 Methodology

This experiment was conducted in a hybrid environment. For obvious reasons, we choose to use thresholds that were below the values considered damaging to hardware. To perform a controlled experiment, we artificially raised and lowered the reading on the ambient temperature sensor with an external heat source.

15.2.1 Physical Machine Configuration

This experiment consisted of four physical machines running the Structant Workernode executable, connected via a Layer 2 1Gb Ethernet switch. A fifth physical machine ran the Structant Dispatcher executable and was connected to the other four machines via the same switch. All machines ran CentOS 6.9 [83], except the machine running the Dispatcher which ran Red Hat Enterprise Linux (RHEL). All five machines had Private IPv4 Addresses. The machines had the following IP addresses:

Node	EntityType	Description	IP Address
Node A	“NODE”	=	[“10.11.1.111”]
Node B	“NODE”	=	[“10.11.1.112”]
Node C	“NODE”	=	[“10.11.1.113”]
Node D	“NODE”	=	[“10.11.1.114”]

Table 15.1

We tested connectivity between machines by pinging the Dispatcher machine from each of the Workernode machines with 100 pings, and checking for 0% loss and consistent round trip time.

15.2.2 Environment Monitor

The physical properties of the environment were measured using a server room environment monitor developed by TechNet Enterprises [97]. The server room environment monitor consisted of a network-enabled Raspberry Pi [44] connected to five remote sensors. The remote sensors included one water presence detection sensor, one air pressure / quality sensor, and five temperature probes. The temperature probes connected to the unit via 25’ cables, allowing us to place them strategically around the room. Figure 15.1 shows the configuration of the physical environment. In order to gather the useful contextual information about the Computational Environment, we placed one temperature probe in front of the physical machines to measure ambient temperature, and each of the remaining four sensors behind the exhaust fans on the four Workernode machines. This configuration allowed Structant to gain additional contextual information by encoding information about how much heat each server produces. We did not use

the water presence detection or air pressure / quality sensors in this experiment.

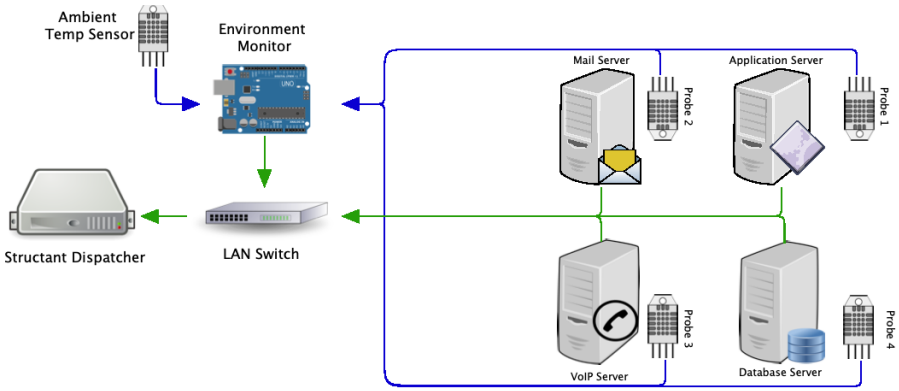


Figure 15.1: An environmental sensor with five probes was used to monitor external context of the server room. One sensor monitored ambient temperature and the remaining sensors monitored server exhaust temperatures.

15.2.3 Calculating Temperatures

The Raspberry Pi provided an API that the Structant Global Context Broker could query. In order to provide an accurate measurement of the server room temperature and avoid erratic responses, the environmental sensor computes the average temperature of the server room over the past ten minutes using an Exponentially Weighted Moving Average, where λ controls the depth of the memory of the EWMA, and T_t is the Temperature observed from a particular probe at time t.

$$\text{EWMA}_t = \lambda T_t + (1 - \lambda)\text{EWMA}_{(t-1)} \tag{15.1}$$

The environmental monitor polls the sensors every sixty seconds and computes the EWMA with $\lambda = 0.10$. The API for the Raspberry Pi's API returns a JSON

object, giving a sensor identifier and the EMWA at the last time the sensor was polled.

15.2.4 Contextual Models

In order to create a realistic Computational Environment, we added two pieces of contextual information for each Node in the Network Model. We assigned each Node a unique name; this was necessary because Structant needed a way to associate sensor data with a Node model. The second piece of contextual information that we added to each Node was a classification. This contextual information modeled how "important" the Computational Operations running on the Computational Resource were to the network. In this experiment, we simulated a Mail Server, VoIP Server, Application Server, and Database Server. In a realistic environment, Computational Resources may provide different services to the users. Users may wish to indicate that some servers should be shut down before others. For example, a user may wish to shut down a legacy application server that employees infrequently use to access archived data before shutting down a mail server providing incoming and outgoing email service to the entire organization. Similarly, a user might find it preferable to shut down a database server that provides data to applications running on the local network before shutting down a VoIP server that is providing phone access to the entire organization. Table 15.2 shows the User Context that we created for each Node.

MachineID	MachineName	Classification
Machine1	"ApplicationServer"	1
Machine2	"MailServer"	10
Machine3	"VoIPServer"	5
Machine4	"DatabaseServer"	5

Table 15.2: User Context entered into the Network Model for the four machines in the experiment.

15.2.5 Desired Behavior

In order to create a pipeline for Structant to execute, we started by defining the desired response behavior. We concluded that the system should behave as follows:

1. If the EWMA of the Computational Environment's ambient temperature rose above 90°F, Structant should take corrective action;

2. Structant should shutdown less critical servers first;

3. Structant should shut down servers forty-five minutes apart to leave time for the temperature to stabilize; and

4. The failure of one server to shutdown should not prevent Structant from attempting to shut down the other servers.

15.2.6 Policies

Given the desired operations, we created several policies that Structant would use to control execution.

PolicyID	EntityType	Operator	Values	ContextLabel	FailOnMissing	Time
Policy1	"NODE"	"ATMOST"	1	"Classification"	TRUE	ALWAYS
Policy2	"NODE"	"ATMOST"	5	"Classification"	TRUE	ALWAYS

Table 15.3: Policies created to facilitate correct shutdown ordering of the Computational Resources.

Policy1 dictates that the Transformation to which it is attached may only run on a Node that has a Classification value of at most one. This policy will prevent Structant from scheduling a given Transformation on Machine1, Machine3, or Machine4. Policy2 dictates that the Transformation to which it is attached may only be run on a Node that has a Classification values of at most five. This policy will prevent Structant from scheduling a given Transformation on Machine1.

We choose to create policies for machines somewhat arbitrarily. In practice, we could have achieved the exact same behavior by creating policies for the Nodes and dictating which shutdown operations they could run. We attached policies to the Transformations because it required less context creation.

15.2.7 Obtaining External Context

We defined a User Policy to query the Raspberry Pi's API. We created an Evaluate function that, when called by the scheduler, queried the Raspberry Pi, parsed the JSON Object, and checked to see if the value returned for the ambient temperature was above 90°F. If the temperature was above 90°F, the policy returned true, otherwise it returned false. The policy was named "PolCheckTemp1". Since this policy only needed to check the ambient temperature, it did not modify any of the context associated with the Computational Resources.

15.2.8 Pipeline Creation

Given the policies and the desired operation, we next created the pipeline. Pipeline creation required only specifying the commands, order, and policies. The pipeline comprised five steps that Structant needed to schedule in the correct order and at the correct times.

Shutting Down the First Node

The intention was to shut down the lowest priority Node first. The first step in the pipeline was a Task containing a single Transformation that the Workernode would run on the Computational Resource causing it to shut down. This Task had no dependence because its execution was controlled directly by the policy PolCheckTemp1. Policy1 was attached to the Task to force Structant to bind it to the lowest priority Node.

Waiting

After shutting down the first Node, we wanted Structant to wait forty-five minutes before shutting down the second Node. To achieve this, we created a Task with a single Transformation that slept for forty-five minutes before exiting. Because any Node could run this Task, we attached no policy to it. Task listed the first shutdown Task as an antecedent so that Structant would not schedule it until after scheduling the first Node to shut down.

Shutting Down the Remaining Nodes

Since we wanted Structant to shut down either of the remaining lower priority servers next, we created another Task with a single Transformation with the same shutdown command as the previous Task. We attached policies PolCheckTemp1 and Policy2 and listed the wait Task as an antecedent. We configured another wait Task and another shutdown task to shut down the final Node, if the temperature had not stabilized. Figure 15.2 shows the Tasks we created to control the shutdown sequence.

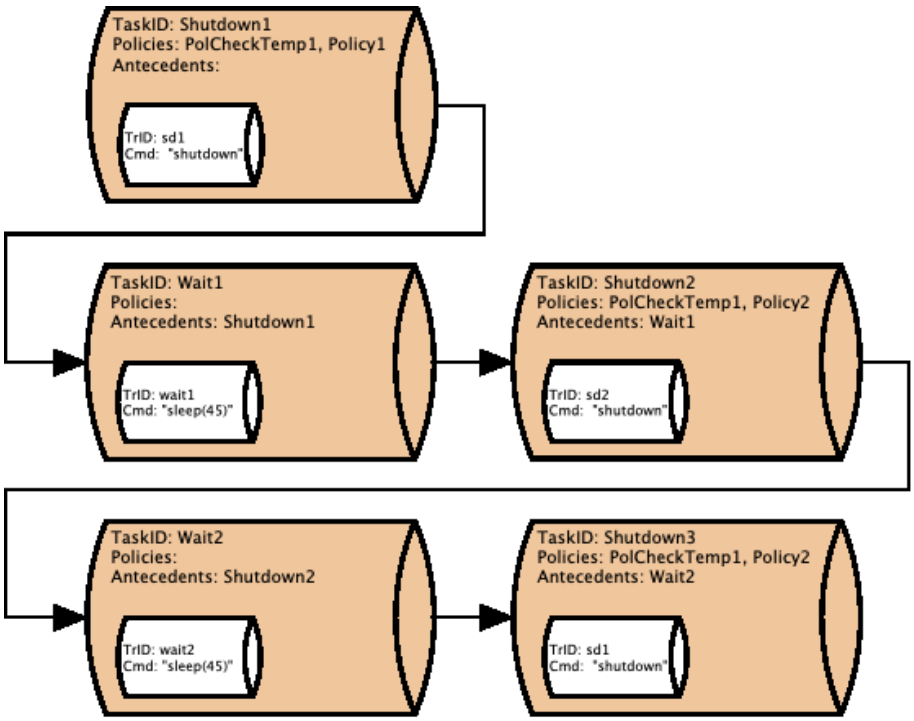


Figure 15.2: Policies and NO-OP sleep Transformations and Local Policies defined the correct shutdown sequence for a collection of servers.

15.3 Results

Structant obtained external contextual information from the environmental sensors and used this information to shut down the correct servers in accordance with the policies and pipeline. The policies installed in the pipeline correctly prohibited a third Node from being shutdown after we removed the source of heat from the sensor and allowed the readings to return to a normal range. Figure 15.3 shows the readings from the external temperature sensors during the experiment.

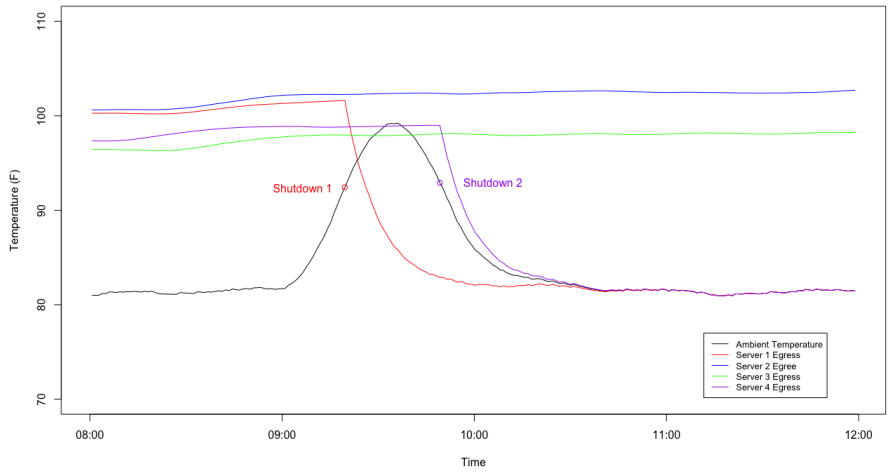


Figure 15.3

Figure 15.4 shows the sequence of events that Structant executed during the experiment.

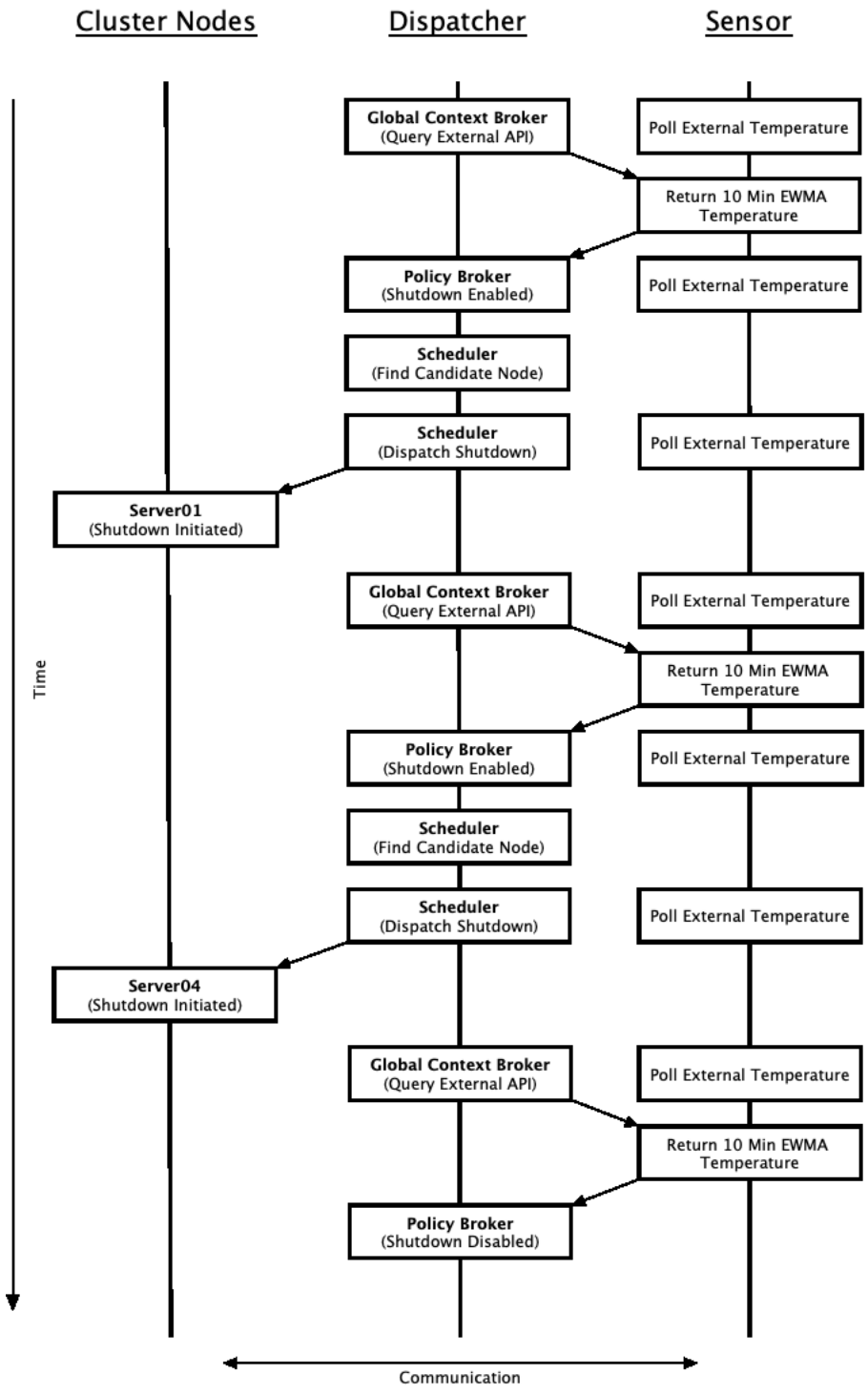

Cluster Nodes
Dispatcher
Sensor
Global Context Broker
(Query External API)
Poll External Temperature
Return 10 Min EWMA
Temperature
Policy Broker
(Shutdown Enabled)
Poll External Temperature
Scheduler
(Find Candidate Node)
Scheduler
(Dispatch Shutdown)
Poll External Temperature
Server01
(Shutdown Initiated)
Global Context Broker
(Query External API)
Poll External Temperature
Return 10 Min EWMA
Temperature
Policy Broker
(Shutdown Enabled)
Poll External Temperature
Scheduler
(Find Candidate Node)
Scheduler
(Dispatch Shutdown)
Poll External Temperature
Server04
(Shutdown Initiated)
Global Context Broker
(Query External API)
Poll External Temperature
Return 10 Min EWMA
Temperature
Policy Broker
(Shutdown Disabled)
Time
Communication

Figure 15.4

15.3.1 Running the Shutdown

During this experiment, the shutdown command proved problematic for the Workernode to run. This occurred because the Workernode is unable to reap the exit status of a Transformation's command until after the process running the command has terminated. The problem with this logic was that the shutdown commands never exits and returns a status, because on the completion of the shutdown process, the machine powers off. Without this return, the Workernode never informs the Dispatcher that the shutdown was successful, and the pipeline becomes stuck. While this presented an issue for this particular scenario, we opted not to change any of Structant's functionality to remedy this.

Arguably, Structant's logic is correct in this case. Structant should not schedule the next Transformation until it has received confirmation that the first Transformation was successful. Because the Node running the shutdown will have powered itself off, there is no way for the Worker to inform the Dispatcher of the success. To remedy this, we made a wrapper process for the shutdown command. The wrapper program simply forked, and checked if the fork was successful. The parent returned zero immediately, and the child process exec'd itself with a shutdown. This technique allowed the Workernode to inform the Dispatcher that it had successfully scheduled the shutdown. The code for the shutdown wrapper is included here for reference.

```
1 int main(int argc, char ** argv){
```

```
2
3     pid = fork();
4     /* child code */
5     if(!pid){
6        char * args[] = {"shutdown", "-h", "60", NULL};
7        char * env[] = {NULL};
8
9        execve("/sbin/shutdown", args, env);
10       perror("execve");
11       exit(1);
12
13    /* parent code */
14    } else {
15       exit(0);
16    }
17 }
```

It should be noted that Structant could automatically PreStage the required executable, eliminating the step of copying the executable to each Node.

15.4 Conclusion

This experiment demonstrated Structant's ability to schedule Transformations with respect to inputs from the Computational Environment. Allowing the Scheduler to use this type of contextual information to map operations to resources has implications and uses beyond distributed Node management. For one, a user could employ a similar pipeline to automatically update computational costs for resources on a cloud computing platform. Given a well-defined API [1], user-defined scheduling functions could automatically query an external source to obtain a resources pricing model, and update the User Context of the Node associated with that computational resource with the new pricing information. Lo-

[1]The user can employ any scraping technique that can retrieve the contextual information and store it as a (Key $\rightarrow$ Value)

cal Policies could prohibit certain Transformations from running on Nodes if the computational cost exceeded a specific threshold. Prior work has explored this possibility, but only with a limited set of predefined resource and cost parameters [22]. Another use for external context is to determine machine specific load. Currently, Structant's ability to determine the load of a machine is limited to the resource slots configured by the user. By querying an available resource monitor, Structant could obtain fine-grained machine load information and incorporate this additional contextual information in its Scheduling procedures.

Chapter 16: Concluding Remarks

In this dissertation, we have introduced Structant, and demonstrated the complex scheduling operations that it can achieve by combining contextual information about computational operations with contextual information about the underlying computation environment. We have shown that by making a task scheduling framework context-aware, we can schedule operations with respect to realistic physical, virtual, and organizational processing requirements and not just a set of predefined criterion.

Structant objectively satisfies the goal of the problem statement (see Section 1.1). When the user enters Computational Operations as interdependent Tasks and Transformations, Structant's Scheduler obeys all processing requirements, and uses historical information from the User and System Context stores to perform forward runtime estimation in a way that allows the pipeline to proceed with reasonable efficiency. The user remains a key decision maker throughout operation as Structant will warn the user and ask for intervention if the Scheduler cannot schedule a job cannot or an operation fails.

By satisfying this problem statement, Structant has solved what we described as the Goldilocks Problem (see Section 1.3.1). Structant provides a realistic

task management framework for organizations with large amounts of disparate data ingested from multiple sources, and processed across multiple machines with different processing capabilities that do not have the budget or technical expertise to adopt cloud-based Big Data processing stacks. By using the contextual information surrounding the machines, operations, and computational operations, Structant profiled execution, used this information to create runtime estimates, and scheduled with respect to organizational policy (see Chapter 14). We also demonstrated how a user can rapidly adapt Structant to an existing pipeline and how Structant can manage operations based on a set of user supplied requirements for a jobs and capabilities of Computational Resources (see Chapter 13). We demonstrated how Structant can manage the pipeline and parallelize execution without modification to any of the existing code or specialized applications provided by the user. We described how the user can make decisions during the pipeline execution (rather than allowing a pipeline to have a hard failure) (see Chapter 12). Finally, we showed how Structant can model and respond to information describing the Computational Environment in a way that allows the organizations to process information in a Computational Environment that maintains regulatory compliance (see Chapter 15).

16.1 The Future of Structant

Due to the success demonstrated during our experiments, we believe that Structant will serve as a valuable research and management tool.

16.1.1 Future Experiments

We plan to continue using Structant to facilitate data collection, movement, and processing for a variety of domains. We intend to use Structant in the domains of disaster response, to facilitate rapid pipeline deployment, automatic data flow, and task management for processing data collected from drones during hurricane damage assessment. Additionally, we plan to use Structant as the basis for building a smart-network monitoring solution. Using Structant's pipelines and policies, we plan to build distributed Node management tools that automatically react to external information including natural disasters, environmental conditions, and security threats.

16.1.2 Future Improvements

Based on our experience using Structant, we propose additions to improve the functionality and accuracy of Structant's modeling components.

Resource Modeling

Structant treats all information in the context store identically. We suggest extending Structant's modeling capabilities to model and differentiate information in a Node that describe the processing characteristics of the Node. Automatic feature selection may help identify contextual information particularly suited to describing the processing capabilities of a Node, but this information should be augmented with in-depth hardware profiling.

We mentioned in Section 9.1 how traditional techniques of modeling resources simply as numerical amounts can lead to inaccurate models of performance. Runtime prediction rarely considers advanced hardware properties such as Bus Speed, CPU Cache, real cores vs logical cores, memory bandwidth, memory clock speed, and the latent variables describing the interrelationship between them.

Machine Learning

Structant's Execution Observer does not use nominal values from context stores in the model that it builds for runtime prediction (see Section 9.2.2). While this limitation facilitates correct operation of the logistic regression models used by the Execution Observer, it presents a limitation because it may exclude valuable information from runtime prediction. Another limitation with Structant's machine learning is that Structant reconstructs the models for each scheduling operation. As the context stores grow larger and the size of the pipelines increase, this begins to impose a non-trivial computational overhead. Based on these limitations, there we propose two future improvements.

The first improvement is model caching. We believe that, in pipelines with large numbers of similar tasks, it would be more efficient to pre-compute models based on automatic feature selection and cross-fold validation, and then select the best model for a mapping based on the Context Identifiers in the mapping. Not

only would this reduce computational overhead, it would help to reduce under and over-fitting of the models.

The second improvement would be on the algorithm itself. Logistic regression, as demonstrated by our experiments, performs nicely with a small amount of contextual information. Problems such as under-fitting or over-fitting start to arise when context stores start to fill up. Logistic regression lacks the ability to model intricate relations between features; other models, such as neural networks, capture these interrelationships much more accurately. We believe that a more robust machine learning algorithm that incorporates the non-numeric features would provide more accurate runtime estimates.

16.2 Contributions Summary

In addition to being a functional tool that users can employ to manage tasks in heterogeneous computational environments, the algorithms and techniques that Structant uses contribute to the greater understanding of context-aware computing, runtime estimating, and task scheduling. The techniques used in Structant provide the corporate and research communities with a new way of viewing Computational Environments — as a collection of Nodes, Edges, Tasks, Datasets, and *Users*, all with relevant (and irrelevant) contextual information that defines their interactions. By viewing the Computational Environment in this fashion, system designers can separate the inference of contextual information from the application logic. Designers can build schedulers, profilers, policy mechanisms,

and task managers to use not only a greater variety of attributes and information about their environments but also latent attributes that may require user defined functions to interpret. Not only does this design technique allow for the creation of systems and algorithms that user richer and more meaningful information as input, it allows these systems to model and run in a greater variety of computational environments and solve a greater variety of problems, because the application logic depends less on specific attributes of the system.

16.3 Final Thoughts

We have seen the utility of Structant both in our experiments and in our discussion with academic and commercial organizations. By taking a generalized, context-aware approach to runtime estimation, pipeline scheduling, and processing requirement inference, Structant not only solves problems with existing solutions, but uncovers a collection of new research challenges. We provide Structant not only as a tool solve processing challenges faced by organizations today, but as a foundation and road map for future work to consider the Computational Environment, user, and system owners as critical decision makers and sources of information about how, when, and where tasks will execute and data will flow. In this respect, Structant is not only a Context-Aware schedule; it is a new way of

understanding the interaction between data, machines, the environment, and the user.

"Any sufficiently advanced technology is indistinguishable from magic."

– Arthur C. Clark

Chapter A: Appendix A

Transformation System Context

Context Identifier	Configurable	Description
"Transformation ID"	YES	A unique identifier for the Transformation within Structant
"Status"	NO	This stores the status of the Transformation.
"Antecedents"	YES	This is a list of the Transformation IDs on which this Transformation depends. All Transformations identified by the Transformations IDs within this list must their Status value set to "COMPLETED" before this Transformation is eligible to be scheduled.
"Posteriors"	YES	This is a list of the Transformation IDs which depend on this Transformation. This information is currently not used but was included to facilitate future optimization and feature implementation. It will be used to quickly find new runnable tasks and to facilitate the ability to re-run selections of a pipeline in the event of failure.
"Command"	YES	The command and arguments used to specify the execution that should take place in this Transformation.
"Input"	YES	A list of files and directories that this Transformation needs in order to be executed.
"Output"	YES	A list of files and directories that this Transformation produces as output.

“IORatio”	YES	An estimate of the size of the input compared to the size of the output.
“ParentTask”	YES	The Task ID that uniquely identifies the Task with which this Transformation is associated.
“ParentObject”	NO	A pointer to the memory location in the Ontology of the Task object representing this Transformation’s parent Task.
“TransformationType”	NO	Stores the type of the current Transformation.
“Thread”	NO	This context is used only by an instance of the Workernode. The Thread is used to reference the process which is running the execution specified by “Command” after the system EXECs the process.
“STDIN”	NO	This context is used only by an instance of the Workernode Stores a reference to the file descriptor for STDIN of the process returned after the system EXECs the command specified in “Command.” This is used to facilitate interaction with a stuck or blocked process
“STDOUT”	NO	This context is used only by an instance of the Workernode Stores a reference to the file descriptor for STDOUT of the process returned after the system EXECs the command specified in “Command” This is used to facilitate interaction with a stuck or blocked process
“AssignedNode”	NO	Stores a reference that uniquely identifies the running instance of the Workernode to which this Transformation has been assigned.
“Policies”	YES	Stores a list of User Policies the the user has attached to this Transformation
“PreStage”	YES	Indicates whether this Transformation should PreStage its data

"PostStage"	YES	Indicates whether this Transformation should PostStage its data
"PreStageDatastore"	YES	The storage identifier that uniquely identifies where this Transformation's inputs should be PreStaged from
"PostStageDatastore"	YES	The storage identifier that uniquely identifies where this Transformation's outputs should be PostStaged to
"UserContext"	YES	A pointer to the memory location of the User Context associated with this Transformation

Table A.1: This table describes the System Context that Structant stores for each Transformation in the Ontology. The user can specify Contextual Values for Contextual Identifiers indicated as Configurable, during setup. The user defines Transformations via the User Interface or on configuration files at the Dispatcher.

Task System Context

Context Identifier	Configurable	Description
"Task ID"	YES	A unique identifier for the Task within Structant
"Status"	NO	This stores the status of the Task
"Antecedents"	YES	This is a list of the Task IDs on which this Task depends All Tasks identified by the Task IDs within this list must their Status value set to "COMPLETED" before this Task is eligible to be scheduled.
"Posteriors"	YES	This is a list of the Tasks IDs which depend on this Task. This information is currently not used but was included to facilitate future optimization and feature implementation. It will be used to quickly find new runnable tasks and to facilitate the ability to re-run selections of a pipeline in the event of failure.
"Transformations"	YES	This is a list of the Transformation IDs which are part of this Task. All Transformations identified by the Transformation IDs in this list must have statuses equal to "COMPLETED" before this Task can be considered "COMPLETED"
"Input"	YES	A list of files and directories that this Task needs in order to be executed
"Output"	YES	A list of files and directories that this Task produces as output
"Assigned Node"	NO	Stores a reference that uniquely identifies the running instance of the Workernode to which this Task has been assigned.
"Policies"	YES	Stores a list of User Policies the user has attached to this Task
"PreStage"	YES	Indicates whether this Task should PreStage its data

"PostStage"	YES	Indicates whether this Task should PostStage its data
"PreStage Datastore"	YES	The storage identifier that uniquely identifies where this Task's inputs should be PreStaged from
"PostStage Datastore"	YES	The storage identifier that uniquely identifies where this Task's outputs should be PostStaged to
"User Context"	YES	A pointer to the memory location of the User Context associated with this Task

Table A.2: This table describes the System Context that Structant stores for each Task in the Ontology. The user can specify Contextual Values for Contextual Identifiers indicated as Configurable, during setup. The user defines Tasks via the User Interface or on configuration files at the Dispatcher.

Node System Context

Context Identifier	Configurable	Description
EntityType	YES	"NODE" - a String that identifies this entity as a Node.
"NodeID"	YES	A globally unique identifier for the Node. Randomly generated if not provided by the user
"MaxTasks"	YES	The maximum number of Tasks that the Node can run concurrently
"DispatcherPort"	YES	The port number to connect to the Dispatcher on
"DispatcherIP"	YES	The IPv4 address to connect to the Dispatcher on
"MaxTrans"	YES	The maximum number of Transformations that the Link can run concurrently
"MaxTasks"	YES	The maximum number of Tasks that the Link can run concurrently
"Policies"	YES	Stores a list of User Policies the user has attached to this Node
"UserContext"	YES	A pointer to the memory location of the User Context associated with this Node
"Edges"	YES	A list of incoming and outgoing Edges from this Node
AssignedTasks	NO	A list of Tasks assigned to this Node
AssignedTrans	NO	A list of Transformations assigned to this Node
FinishedTasks	NO	A list of Tasks assigned to this Node, finished, and not yet unbound
FinishedTrans	NO	A list of Transformations assigned to this Node, finished, and not yet unbound
HELLO	NO	Indication of whether or not the Node has informed the Dispatcher of its information on startup

Table A.3: This table describes the System Context that Structant stores for each Node in the Fabric Model. The user can specify Contextual Values for Contextual Identifiers indicated as Configurable, during setup. The user defines Node context on configuration files at the Workernode.

Edge System Context

Context Identifier	Configurable	Description
"Link ID"	YES	A unique identifier for the Link within Structant
EntityType	YES	"EDGE"; a String that identifies this entity as a Node.
"NodeID"	YES	A globally unique identifier for the Link. Randomly generated if not provided by the user
"MaxTasks"	YES	The maximum number of Tasks that the Link can run concurrently
"MaxTrans"	YES	The maximum number of Transformations that the Link can run concurrently
"SourceNode"	YES	The NodeID of the Node that can send traffic over this Edge
"StatedCapacity"	YES	The throughput that the user has defined for this Edge
"ActualCapacity"	YES	The measured throughput of the Edge based on prior observations
"Policies"	YES	Stores a list of User Policies the user has attached to this Edge
"UserContext"	YES	A pointer to the memory location of the User Context associated with this Edge
AssignedTasks	NO	A list of Tasks assigned to this Edge
AssignedTrans	NO	A list of Transformations assigned to this Edge
AssignedTasks	NO	A list of Tasks assigned to this Edge
Alive	YES	Whether or not the Edge was found to be up based on the last check

Table A.4: This table describes the System Context that Structant stores for each Edge in the Fabric Model. The user can specify Contextual Values for Contextual Identifiers indicated as Configurable, during setup. The user defines Edges via the User Interface or on configuration files at the Dispatcher.

Heartbeat Information

Context Identifier	Configurable	Description
"HeartbeatID"	NO	Unique identifier for the Heartbeat within Structant
"HBTime"	YES	The local time on the Node when it generated the Heartbeat message
"MaxTrans"	YES	The maximum number of Transformations that the Node can run concurrently
"MaxTasks"	YES	The maximum number of Tasks that the Node can run concurrently
AssignedTasks	NO	A list of Tasks assigned to this Node
AssignedTrans	NO	A list of Transformations assigned to this Node
FinishedTasks	NO	A list of Tasks assigned to this Node, finished, and not yet unbound
FinishedTrans	NO	A list of Transformations assigned to this Node, finished, and not yet unbound

Table A.5: This table describes the information contained in Heartbeat messages from the Workernodes to the Dispatcher. The Workernodes automatically send the Dispatcher a Heartbeat message every fifteen seconds, or automatically after completing a Transformation. The information contained in this Heartbeat message corresponds to the Workernode's local context stores at the time of message creation. The Dispatcher never explicitly responds to Heartbeat messages. Rather, the Dispatcher updates it local models of the Workernode based on the information in the Heartbeat and issues subsequent directives in accordance with its updated models.

Chapter B: Appendix B

ICD10 Code	Description
	Poisoning by ...
T36	Systemic antibiotics
T37	Systemic anti-infectives and antiparasitics
T38	Hormones and their synthetic substitutes and antagonists, not elsewhere defined
T39	Nonopioid analgestics, antipyretics, and antiheumatics
T40	Narcotics and psychodysleptics [hallucinogens]
T41	Anesthetics and therapeutic gases
T42	Antiepileptic, sedative-hypnotic, and antiparkonsonism drugs
T43	Psychotropic drugs, not elsewhere defined
T44	Drugs primarily affecting the autonomic nervous system
T45	Primarily systemic and haematological agents, not elsewhere defined
T46	Agents primarily affecting the cardiovascular system
T47	Agents primarily affecting the gastrointestinal system
T48	Agent primarily acting on smooth and skeletal muscles and the respiratory system
T49	Topical agents primarily affecting skin mucous membrane and by the ophthalmological, otorhinolaryngological and dental drugs
T50	Diuretics and other unspecified drugs, medicaments and biological substances

Table B.1: ICD10 Codes: Contributing

ICD10 Code	Description
	Poisoning by ...
T40.0	Opium
T40.1	Heroin
T40.2	Other Opioids
T40.3	Methadone
T40.4	Other Synthetic Narcotics
T40.5	Cocaine
T40.6	Other unspecified narcotics
T40.7	Cannabis (derivatives)
T40.8	Lysergide [LSD]
T40.9	Other and unspecified psychodysleptics [hallucinogens]
T43.0	Tricyclic and tetracyclic antidepressants
T43.1	Monoamine-oxidase-inhibitor antidepressants
T43.2	Other and unspecified anti-depressants
T43.3	Phenothiazine antipsychotics and neuroleptics
T43.4	Butyrophenone and thioxanthene neuroleptics
T43.5	Other and unspecified antipsychotics and neuroleptics
T43.6	Psychosimulants with abuse potential
T43.8	Other psychotropic drugs, not elsewhere classified
T43.9	Psychotropic drug, unspecified
T50.0	Mineralocorticoids and their antagonists
T50.1	Loop [high-ceiling] diuretics
T50.2	Carbonic-anhydrase inhibitos, benzothiadiazides and other diuretics
T50.3	Electrolytic, caloric and water-balance agents
T50.4	Drugs affecting uric acid metabolism
T50.5	Appetite depressants
T50.6	Antidotes and chelating agents, not elsewhere defined
T50.7	Analeptics and opioid receptor antagonists
T50.8	Diagnostics agents
T50.9	Other and unspecified drugs, medicaments and biological substances

Table B.2: ICD10 Codes Contributing Cont'

ICD10 Code	Description
	Accidental poisoning by and exposure to...
X40	nonopioid analgesics, antipyretics and antirheumatics
X41	antiepileptic, sedative-hypnotic, antiparkinsonism and psychotropic drugs, not elsewhere classified
X42	narcotics and psychodysleptics [hallucinogens], not elsewhere classified
X43	other drugs acting on the autonomic nervous system
X44	other and unspecified drugs, medicaments and biological substances
	Intentional self-poisoning by and exposure to...
X60	nonopioid analgesics, antipyretics and antirheumatics
X61	antiepileptic, sedative-hypnotic, antiparkinsonism and psychotropic drugs, not elsewhere classified
X62	narcotics and psychodysleptics [hallucinogens], not elsewhere classified
X63	other drugs acting on the autonomic nervous system
X64	other and unspecified drugs, medicaments and biological substances
X85	Assault by drugs, medicaments and biological substances
	Event of undetermined intent
Y10	nonopioid analgesics, antipyretics and antirheumatics
Y11	antiepileptic, sedative-hypnotic, antiparkinsonism and psychotropic drugs, not elsewhere classified
Y12	narcotics and psychodysleptics [hallucinogens], not elsewhere classified
Y13	other drugs acting on the autonomic nervous system
Y14	other and unspecified drugs, medicaments and biological substances

Table B.3: ICD10 Codes: Underlying

Category	Underlying Cause	Contributing Cause
Illicit Drug Poisoning	X40 X41 X42 X43 X44 X60 X61 X63 X64 X85 Y10 Y11 Y12 Y13 Y14	T40.1 T40.5 T40.7 T40.8 T40.9 T43.6
Pharmaceutical Poisoning	X40 X41 X42 X43 X44 X60 X61 X62 X63 X64 X85 Y10 Y11 Y12 Y14	T36 T37 T38 T39 T40.2 T40.3 T40.4 T41 T42 T43.0 T43.1 T43.2 T43.2 T43.3 T43.4 T43.5 T43.8 T43.9 T44 T45 T46 T47 T48 T48 T50.0 T50.1 T50.2 T50.3 T50.4 T50.5 T50.6 T50.7 T50.8
Prescription opioid poisoning	X40 X41 X42 X43 X44 X60 X61 X62 X63 X64 X85 Y10 Y11 Y12 Y13 Y14	T40.2 T40.3 T40.4
Other pharmaceutical poisoning	X40 X41 X42 X43 X44 X60 X61 X62 X63 X64 X85 Y10 Y11 Y12 Y13 Y14	T36 T37 T38 T39 T41 T42 T43.0 T43.1 T43.2 T43.3 T43.4 T43.5 T43.8 T43.9 T44 T45 T46 T47 T48 T49 T50.0 T50.1 T50.2 T50.3 T50.4 T50.5 T50.6 T50.7 T50.8
Illicit opioid poisoning	X40 X41 X43 X44 X60 X61 X62 X63 X64 X85 Y10 Y11 Y12 Y13 Y14	T40.0 T40.1
All opioid poisoning (illicit and prescription)	X40 X41 X42 X43 X44 X60 X61 X62 X63 X64 X85 Y10 Y11 Y12 Y13 Y14	T40.0 T40.1 T40.2 T40.3 T40.4

Table B.4

Agree	County	Code	Notes
-	Barrow	13013	Athens
+	Bartow	13015	
-	Butts	13035	Macon
+	Carroll	13045	
+	Cherokee	13057	
+	Clayton	13063	
+	Cobb	13067	
+	Coweta	13077	
-	Dawson	13085	Athens
+	DeKalb	13089	
+	Douglas	13097	
+	Fayette	13113	
-	Forsyth	13117	Athens
+	Fulton	13121	
-	Gwinnett	13135	Athens
+	Haralson	13143	
-	Heard	13149	Heard
+	Henry	13151	
-	Jasper	13159	Macon
-	Lamar	13171	Macon
-	Meriwether	13199	Columbus
-	Morgan	13211	Huntsville
-	Newton	13217	Athens
+	Paulding	13223	Athens
+	Pickens	13227	
-	Pike	13231	Macon
-	Rockdale	13247	Athens
-	Spalding	13255	Macon
-	Walton	13297	Athens

Table B.5: Results of our automatic geomapping algorithm versus hand-coded ground truth

Bibliography

[1] Maxmind geolite2. https://www.maxmind.com/en/open-source-data-and-api-for-ip-geolocation, publisher=MaxMind.

[2] Weka 3: Data mining software in java. http://www.cs.waikato.ac.nz/ml/weka/, journal=Weka 3 - Data Mining with Open Source Machine Learning Software in Java.

[3] Guide to icd-9-cd and icd-10 codes related to pain and poisoning, Aug 2013.

[4] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In Hans-W. Gellersen, editor, Handheld and Ubiquitous Computing, pages 304–307. Springer Berlin Heidelberg.

[5] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. Commun. ACM, 17(12):685–690, December 1974.

[6] Federal Drug Administration. Drugsfda: Fda approved drug products. https://www.accessdata.fda.gov/scripts/cder/daf/index.cfm?event=overview.process&varApplNo=022272, 2019.

[7] National Credit Union Administration. 12 cfr part 748 and appendix a guidelines for safeguarding member information. https://www.ffiec.gov/exam/infobase/documents/02-ncu-12_cfr_748_app_a_safeguard_info-010100.pdf, 2001.

[8] Popescu Adrian Daniel, Balmin Andrey, Ercegovac Vuk, and Ailamaki Anastasia. Predict: towards predicting the runtime of large scale iterative analytics. Proc. VLDB Endow., 6(14):1678–1689, 2013. 2556553.

[9] Sameer Agarwal, Srikanth Kandula, Nico Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Reoptimizing data parallel computing. In Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pages 281–294, San Jose, CA, 2012. USENIX.

[10] I. Ahmad and Y. K. Yu-Kwong Kwok. A new approach to scheduling parallel programs using task duplication. In 1994 Internatonal Conference on Parallel Processing Vol. 2, volume 2, pages 47–51, Aug 1994.

[11] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. IEEE Transactions on Parallel and Distributed Systems, 9(9):872–892, Sep. 1998.

[12] Jeff Ahrenholz. Comparison of core network emulation platforms. In 2010-Milcom 2010 Military Communications Conference, pages 166–171. IEEE, 2010.

[13] L. B. de Almeida, E. C. de Almeida, J. Murphy, R. E. De Grande, and A. Ventresque. Bigdatanetsim: A simulator for data and process placement in large big data platforms. In 2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT), pages 1–10.

[14] and B. Shirazi and J. Marquis. Mapping of parallel tasks to multiprocessors with duplication. In Proceedings of the Thirty-First Hawaii International Conference on System Sciences, volume 7, pages 96–105 vol.7, Jan 1998.

[15] and S. K. Tripathi. On performance prediction of parallel computations with precedent constraints. IEEE Transactions on Parallel and Distributed Systems, 11(5):491–508, May 2000.

[16] Marcos D. Assunção, Rodrigo N. Calheiros, Silvia Bianchi, Marco A. S. Netto, and Rajkumar Buyya. Big data computing and clouds: Trends and future directions. Journal of Parallel and Distributed Computing, 79-80:3–15, 2015.

[17] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 23(2):187–198, 2011.

[18] Francoise Balmas. Displaying dependence graphs: a hierarchical approach. Journal of Software Maintenance and Evolution: Research and Practice, 16(3):151–185, 2004.

[19] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. Jmt: Performance engineering tools for system modeling. SIGMETRICS Perform. Eval. Rev., 36(4):10–15, March 2009.

[20] C. Bolchini, F.A. Schreiber, and L. Tanca. A methodology for a very small data base design. Information Systems, 32(1):61 – 82, 2007.

[21] D. Bozdag, U. Catalyurek, and F. Ozguner. A task duplication based bottom-up scheduling algorithm for heterogeneous environments. In Proceedings 20th IEEE International Parallel Distributed Processing Symposium, pages 12 pp.–, April 2006.

[22] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. In Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, volume 1, pages 283–289. IEEE, 2000.

[23] HEE EON BYUN and KEITH CHEVERST. Utilizing context history to provide dynamic adaptations. Applied Artificial Intelligence, 18(6):533–548, 2004.

[24] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software: Practice and experience, 41(1):23–50, 2011.

[25] Fred Canter. V7m 2.1 spd. http://bitsavers.org/pdf/dec/pdp11/ultrix-11/Unix_V7M_Release_2.1_Software_Description_Sep81.pdf.

[26] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: Easy and efficient parallel processing of massive data sets. Proc. VLDB Endow., 1(2):1265–1276, August 2008.

[27] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. Tenzing a sql implementation on the mapreduce framework. In Proceedings of VLDB, pages 1318–1327, 2011.

[28] Harry Chen, T. I. M. Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. The Knowledge Engineering Review, 18(3):197–207, 2004.

[29] Q. Chen, J. Yao, B. Li, and Z. Xiao. Pisces: Optimizing multi-job application execution in mapreduce. IEEE Transactions on Cloud Computing, 7(1):273–286, 2019.

[30] Yuan Chen, Subu Iyer, Xue Liu, Dejan Milojicic, and Akhil Sahai. Translating service level objectives to lower level policies for multi-tier services. Cluster Computing, 11(3):299–311, 2008.

[31] Giovanni Chiola. A software package for the analysis of generalized stochastic petri net models. In International Workshop on Timed Petri Nets, pages 136–143, Washington, DC, USA, 1985. IEEE Computer Society.

[32] Microsoft Corporation. What is hybrid cloud computing. https://azure.microsoft.com/en-us/overview/what-is-hybrid-cloud-computing/, 2019.

[33] R. C. Correa, A. Ferreira, and P. Rebreyend. Scheduling multiprocessor tasks with genetic algorithms. IEEE Transactions on Parallel and Distributed Systems, 10(8):825–837, Aug 1999.

[34] PCI Security Standards Council. Secure softwarerequirements and assessment procedures, 2019.

[35] Ardagna Danilo, Barbierato Enrico, Evangelinou Athanasia, Gianniti Eugenio, Gribaudo Marco, B. M. Pinto lio, Guimar Anna, es, Silva Ana Paula Couto da, and M. Almeida Jussara. Performance prediction of cloud-based big data applications, 2018. 3184420 192-199.

[36] Sekhar Darbha and Dharma P. Agrawal. A task duplication based scalable scheduling algorithm for distributed memory systems. Journal of Parallel and Distributed Computing, 46(1):15 – 27, 1997.

[37] J. D. Day and H. Zimmermann. The osi reference model. Proceedings of the IEEE, 71(12):1334–1340, Dec 1983.

[38] John Day. The (un)revised osi reference model. SIGCOMM Comput. Commun. Rev., 25(5):39–55, October 1995.

[39] Anind K. Dey. Understanding and using context. Personal Ubiquitous Comput., 5(1):4–7, January 2001.

[40] Goran Lj. Djordjevic and Milorad B. T. A heuristic for scheduling task graphs with communication delays onto multiprocessors. Parallel Computing, 22(9):1197 – 1214, 1996.

[41] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel processing letters, 21(02):173–193, 2011.

[42] R. Eliasi, T. Elperin, and A. Bar-Cohen. Monte carlo thermal optimization of populated printed circuit board. IEEE Transactions on Components, Hybrids, and Manufacturing Technology, 13(4):953–960, Dec 1990.

[43] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with big data analytics. interactions, 19(3):50–59, May 2012.

[44] Raspberry Pi Foundation. Raspberry pi - teach, learn, and make with raspberry pi. https://www.raspberrypi.org/.

[45] The Apache Software Foundation. Apache hadoop. http://hadoop.apache.org, 2018.

[46] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. Walking in facebook: A case study of unbiased sampling of osns. In 2010 Proceedings IEEE Infocom, pages 1–9. Ieee, 2010.

[47] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In 2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15), pages 459–471, 2015.

[48] Google. Gcp pricing — google cloud, 2019.

[49] Network Working Group. Ip network address translator (nat) terminology and considerations. https://tools.ietf.org/html/rfc2663, 1999.

[50] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis: the control project. Computer, 32(8):51–59, Aug 1999.

[51] K. Henricksen and J. Indulska. A software engineering framework for context-aware pervasive computing. In Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. Proceedings of the, pages 77–86.

[52] Karen Henricksen, Jadwiga Indulska, and Ted McFadden. Modelling context information with orm. On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops, pages 626–635. Springer Berlin Heidelberg.

[53] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling context information in pervasive computing systems. Pervasive Computing, pages 167–180. Springer Berlin Heidelberg.

[54] Jong-yi Hong, Eui-ho Suh, and Sung-Jin Kim. Context-aware systems: A literature review and classification. Expert Systems with Applications, 36(4):8509–8522, 2009.

[55] E. S. H. Hou and N. Ansari and. A genetic algorithm for multiprocessor scheduling. IEEE Transactions on Parallel and Distributed Systems, 5(2):113–120, Feb 1994.

[56] Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. J. ACM, 24(2):280–289, April 1977.

[57] IBM. Ibm solutions grid for business partners: Helping ibm business partners to grid enable applications for the next phase of e business on demand.

[58] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 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.

[59] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 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.

[60] Information technology – security techniques – information security management systems – overview and vocabulary. Standard, International Organization for Standardization, Berlin, DE, March 2018.

[61] Aubin Jarry, Henri Casanova, Francine Berman, et al. Dagsim: A simulator for dag scheduling algorithms. 2000.

[62] Jo, Coutaz lle, L. Crowley James, Dobson Simon, and Garlan David. Context is key. Commun. ACM, 48(3):49–53, 2005. 1047703.

[63] Oh-Han Kang and Si-Gwan Kim. A task duplication based scheduling algorithm for shared memory multiprocessors. Parallel Computing, 29(1):161 – 166, 2003.

[64] Michael Kerrisk. free(1) - linux manual page. http://man7.org/linux/man-pages/man1/free.1.html, 2019.

[65] Michael Kerrisk. unzip(1) - linux manual page. http://man7.org/linux/man-pages/man1/unzip, 2019.

[66] Eric A. King. How to buy data mining ; a framework for avoiding costly project pitfalls in predictive analytics. DM Review, 15(10):38, 2005.

[67] Yu-Kwong Kwok. On exploiting heterogeneity for cluster based parallel multithreading using task duplication. The Journal of Supercomputing, 25(1):63–72, May 2003.

[68] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv., 31(4):406–471, December 1999.

[69] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. ACM Trans. Program. Lang. Syst., 4(3):382–401, July 1982.

[70] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 631–636. ACM, 2006.

[71] H. Liu. Big data drives cloud adoption in enterprise. IEEE Internet Computing, 17(4):68–71, July 2013.

[72] Erik Maehle and Franz-J. Markus. Fault-Tolerant Dynamic Task Scheduling Based on Dataflow Graphs, pages 357–371. Springer US, Boston, MA, 1998.

[73] V. W. Mak and S. F. Lundstrom. Predicting performance of parallel computations. IEEE Transactions on Parallel and Distributed Systems, 1(3):257–270, July 1990.

[74] Tillenius Martin, Larsson Elisabeth, M. Badia Rosa, and Martorell Xavier. Resource-aware task scheduling. ACM Trans. Embed. Comput. Syst., 14(1):1–25, 2015. 2638554.

[75] Kurt Mehmet Can, Krishnamoorthy Sriram, Agrawal Kunal, and Agrawal Gagan. Fault-tolerant dynamic task graph scheduling, 2014. 2683672 719-730.

[76] R. Nelson and A. N. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. IEEE Transactions on Computers, 37(6):739–743, June 1988.

[77] U.S. Department of Justice. Uniform crime report (ucr) program - fbi. https://www.fbi.gov/services/cjis/ucr, 2019.

[78] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[79] Chan-Ik Park and Tae-Young Choe. An optimal scheduling algorithm based on task duplication. In Proceedings. Eighth International Conference on Parallel and Distributed Systems. ICPADS 2001, pages 9–14, June 2001.

[80] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In 2008 IEEE International Conference on Cluster Computing, pages 142–151. IEEE, 2008.

[81] C. L. Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. Information Sciences, 275:314–347, 2014.

[82] STELLA C.S. PORTO and CELSO C. RIBEIRO. A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. International Journal of High Speed Computing, 07(01):45–71, 1995.

[83] The CentOS Project. Centos project. https://www.centos.org/.

[84] J. Xu J Zhou P.S. Yu J. Wang, H. Xiong Y. Ishikawa. On mining big data. In Lecture Notes in Computer Science, volume 7923. Springer-Verlag, 2013.

[85] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. IEEE Transactions on Computers, 38(8):1110–1123, 1989.

[86] S. Ranaweera and D. P. Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000, pages 445–450, May 2000.

[87] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, pages 1357–1369, New York, NY, USA, 2015. ACM.

[88] Thomas Sandholm and Kevin Lai. Mapreduce optimization using regulated dynamic prioritization. In Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '09, pages 299–310, New York, NY, USA, 2009. ACM.

[89] A. Sangroya and R. Singhal. Performance assurance model for hiveql on large data volume. In 2015 IEEE 22nd International Conference on High Performance Computing Workshops, pages 26–33.

[90] N. Sannomiya. A tabu search with a new neighborhood search technique applied to flow shop scheduling problems. In Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No.00CH37187), volume 5, pages 4606–4611 vol.5, Dec 2000.

[91] Matthew A. Scholl, Kevin M. Stine, Joan Hash, Pauline Bowen, L. Arnold Johnson, Carla Dancy Smith, and Daniel I. Steinberg. Sp 800-66 rev. 1. an introductory resource guide for implementing the health insurance portability and accountability act (hipaa) security rule. Technical report, Gaithersburg, MD, United States, 2008.

[92] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. Proc. VLDB Endow., 5(12):1736–1747, August 2012.

[93] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system.

[94] PassMark Software. Passmar software. https://www.cpubenchmark.net.

[95] Clemens Szyperski, Martin Petitclerc, and Roger Barga. Three experts on big data engineering. IEEE Software, 33(2):68–72, 2016.

[96] Andrew Tanenbaum. Computer Networks. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[97] LLC TechNet Enterprises. Technet enterprises. https://www.technetenterprises.com/commercial.

[98] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. Proc. VLDB Endow., 2(2):1626–1629, August 2009.

[99] H. Topcuoglu, S. Hariri, and Wu Min-You. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems, 13(3):260–274, 2002.

[100] Tatsuhiro Tsuchiya, Tetsuya Osada, and Tohru Kikuno. Genetics-based multiprocessor scheduling using task duplication. Microprocessors and Microsystems, 22(3):197 – 207, 1998.

[101] Elizabeth Varki and Lawrence W. Dowdy. Analysis of balanced fork-join queueing networks. SIGMETRICS Perform. Eval. Rev., 24(1):232–241, May 1996.

[102] K. Wang and M. M. H. Khan. Performance prediction for apache spark platform. In 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, pages 166–173.

[103] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale. Modeling interference for apache spark jobs. In 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pages 423–431.

[104] Weinan Wang, Joseph E Magerramov, Maxym Kharchenko, Min Zhu, Aaron D Kujat, Alessandro Gherardi, and Jason C Jenks. Facilitating data redistribution in database sharding, April 23 2013. US Patent 8,429,162.

[105] Joel Wolf, Andrey Balmin, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Sujay Parekh, Kun-Lung Wu, and Rares Vernica. On the optimization of schedules for mapreduce workloads in the presence of shared scans. The VLDB Journal, 21(5):589–609, October 2012.

[106] Wensheng Yao, Jinyuan You, and Baiyan Li. Main sequences genetic scheduling for multiprocessor systems using task duplication. Microprocessors and Microsystems, 28(2):85 – 94, 2004.

[107] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In Proc. of EuroSys, pages 265–278. ACM, 2010.

[108] F. Zhu, M. W. Mutka, and L. M. Ni. Service discovery in pervasive computing environments. IEEE Pervasive Computing, 4(4):81–90, Oct 2005.