UNIVERSITY OF OSLO
Department of Informatics

Exploring Patterns for Scalability of Network Administration with Topology Constraints

Master thesis

Matthew S. Disney
Oslo University College

May 23, 2007

Abstract

This thesis considers the impact of distributed network communication patterns on the scalability of dynamic systems configuration and monitoring using the software cfengine. Decentralized patterns are partially implemented as cfengine policy on topologies with node degree constraints. Experiments investigate total processing latency of patterns. Results show they provide a balanced approach to parallelization and scalability. The study of patterns on a chain topology reveals the challenge of phasing delay in deep tree structures. A time buffering method for reducing total processing latency is tested and found to be effective. Included are suggestions on new cfengine functionality and syntax to support patterns integration. As a whole, this thesis offers new perspectives in on-going patterns research as well as identifying challenges and solutions for bringing patterns to cfengine.

Acknowledgements

First, I am grateful for the teaching, patience, guidance, and support of my advisor, Professor Mark Burgess. His work and research in systems administration is instrumental in the effort to illuminate the technical field with analytical science and I am very proud to have been his student and advisee. I have learned much from the instructors in this program and I thank them for their excellent efforts. I am also thankful to Professor Rolf Stadler at the Royal Institute of Technology (KTH) in Stockholm and his students for their pioneering development of network navigation patterns, for generously hosting me to learn more about their work, and for the numerous helpful discussions.

I am honored to be in the good company of my fellow classmates and alumni in this program; they have been excellent comrades throughout our demanding classes. Alex Andersen (first, as always), Kyrre Begnum, and Edson Ochoa have been my closest companions and colleagues at school here and I am grateful for their friendship, good will, and collaboration.

The support of my amazing friends and family was nothing less than critical in my will and ability to move to Norway to pursue my master's degree. It is humbling and I will never forget their encouragement. My parents have given me a life of learning, comfort, and opportunity even beyond their own. If there is a spirit of achievement in me, it has been passed down from my mother and father.

Finally, I want to thank my wife, Hannah, who has been my constant companion for these two years. She has supported me unconditionally in every way and at every turn, often making sacrifices to do so. Her responsibility for this new success in my life cannot be overstated; I would never have given any serious consideration to taking the bold step of moving to Norway for this program without her encouragement. And I am certainly glad we did.

Oslo, May 2007

Matt Disney

Preface

Developed in conjunction with this thesis, Professor Mark Burgess and I have cowritten a paper entitled "Understanding Scalability in Network Aggregation with Continuous Monitoring," which has been submitted to the 18th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2007) conference. We have also collaborated with Professor Rolf Stadler at the Royal Institute of Technology (KTH), Stockholm on a paper entitled "Using Patterns in Cfengine for Robustly Scaling Network Administration," which has been submitted to the 21st USENIX Large Installation System Administration conference. As of May 2007, both papers are awaiting acceptance.

This work is supported by the EC IST-EMANICS Network of Excellence (#26854).

Contents

List of Figures

Chapter 1

Introduction

1.1 Dynamic systems configuration

Configuration management is a moniker often applied to system administration of many hosts. This term has also been used in the context of *software configuration management*, which may be similar in some respects but should be considered a separate field unrelated to dynamic system configuration (DSC) for the purposes of this thesis. Other popular labels for configuration management include those similar to *system configuration* [1], *large scale system configuration* [2], and *infrastructure management* [3]. For clarification, this thesis will rely on the term *system configuration*.

System configuration includes all aspects of managing the service of a host or group of hosts. This can range greatly from fundamental tasks such as installing operating system software and controlling access to advanced tasks such as automatically scaling network services or programmatically responding to failures. Such advanced concepts can be considered closely related to computer autonomics [4], described in [5] as the creation of systems that are self-configuring, self-healing, self-optimizing, and self-protecting.

In 1997, Evard [1] described a history of system configuration that highlighted an early system cloning method [6], the emergence of NIS and NFS [7], and the introduction of the *Tivoli Systems Management Environment* [8]. System configuration tools were at first primarily *low-level* [9] [10], requiring the system administrator to specify exactly what the desired outcome would be on each host and how to get there. Evard noted a significant expansion in 1994 with

four ambitious new configuration systems designed for centrally managing networked hosts: LCFG [11], GeNUAdmin [12], OMNICONF [13],and Config [14]; Cfengine [15] was created during the same period. The emergence of these tools represents a major step toward DSC and gave system administrators access to high-level tools specifically created for the management of many hosts. This thinking, *dynamic* system configuration (DSC), focuses on the challenge of requirements that change over time and has developed jointly with high-level system configuration ideas that allow for increasingly abstract specifications [10]. DSC stands in contrast to other system configuration ideas that represent an ad hoc approach or do not take configuration drift from the baseline into account. Some similar form of this kind of policy-based management, in which can be seen strong similarities to configuration programming work described in [16] as early as 1987, is a necessary component of autonomic systems.

Recent years have seen not only an expansion of the system configuration tool market (evident from the selection of technologies in [9]) but also increasing attention to the theory involved in managing systems. This work continues in that vein of research to provide an analytically sound basis for challenges facing contemporary system administrators with a large number of hosts as well as difficulties many system or network administrators will encounter in a future of pervasive network computing.

The approaches described in this thesis are motivated by a desire to explore and analyze the decentralization capabilities of a DSC tool, in this case cfengine. Inspiration for how to approach decentralization is provided by research in network navigation patterns, which include algorithms for distributing monitoring-related resource demands throughout a network in a controlled manner (as described in Section 2.1).

1.2 The centralization question

A centralized interface to configuration policy is an important aspect in the simplification of system configuration. That begs the question: Why is the issue of decentralization important in the context of DSC, when centralization is the key to a scalable DSC interface?

Policy interface in system configuration is something that benefits from centralization: a single policy change can cause numerous configuration actions. However, these two aspects (*policy interface* and *configuration action*,including

communication) can be decoupled and optimized independently.

The problem with centralized configuration action involves the limitations of a single system to either execute those actions itself or to communicate those actions to other systems for execution. As an example, it is useful to enumerate the parallel computation resource constraints internal to a computer system:

CPU The number of processors available constrains the number of operations that can happen simultaneously. The processor architecture and speed also affect how many operations can happen within a given time period.

Memory Each running process is loaded into memory, which means that the number of running processes is restricted by the memory as well as the size of those processes.

Storage Many programs require long-term storage for data that must reside outside of memory. This storage typically consists of hard drives but can easily be other forms of media; however, regardless of the media type, a common characteristic is that the media is architecturally further away from the CPU and has slower read and write performance than the main memory and CPU cache. This storage can also provide low-performance expanded (virtual or swap) memory space used to house data.

Network The network hardware serves to limit the amount and rates of data that can be communicated with other systems.

Operating System The operating system itself contains a number of constraints, as its architecture directly affects the performance of the hardware to which it is an interface. Different operating systems will take advantage of certain resources (e.g. memory, networking, process scheduling, etc...) better than others.

If any of these resources needed by a system configuration program is completely consumed, the ability of the centralized configuration action server to execute its policy in a timely fashion, or at all, is crippled. In other words, unlike policy interface, centralizing configuration action introduces the concern of being a performance or scalability bottleneck due to resource constraints.

This notion of *controlled* methods for decentralization is foreseeably important to the userbase of Cfengine; the raison d'être for a DSC utility is to manage a set of nodes. Pursuing a modification to a DSC utility that might offer decentralization benefits but decreases its capability to manage the subject

networked nodes is generally undesirable. Network navigation patterns that have recently been considered are generally applied on tree topologies. This structure provides centralized policy interface and decentralized policy action; ergo network navigation patterns offer a suitable direction for experimenting with decentralization in Cfengine.

Note: Any reference to "decentralization" for the remainder of this thesis is intended to represent "decentralizing configuration action," not configuration policy interface.

1.3 Motivation and research questions

A combination of cfengine and patterns would be desirable in at least two general scenarios:

- Existing cfengine implementations that are facing limits to their central cfengine server resources or that would generally benefit from shifting the centralized workload throughout the network, and

- Decentralized networks, particularly mobile ad-hoc networks (e.g. sensor networks) but also including wide area networks and any topology that can benefit from proxied cfengine usage (e.g. firewalled enterprise local area networks).

However, in order to be useful in these contexts, a number of questions must be answered about cfengine and patterns, which can be decomposed into the following topics and questions.

1.3.1 Echo pattern

Can a true parallel echo pattern be configured in cfengine? If so, how does this compare with serial and parallel stars? Does it meet expectations of time to aggregation values between that of serial and parallel stars? What is the workload comparison for star and echo patterns? What is the parallel speedup for parallel star and echo patterns compared to a serial star?

1.3.2 Aggregating chain

Can an aggregating chain be implemented in cfengine using the *copy* and *editfiles* actions? If so, does the chain perform as expected, with a TTA that corresponds directly to the chain length? If it performs worse or better, to what extent and why? Can this performance be predicted? Can it be tuned and optimal variable values determined? Can a chain be implemented in cfengine using voluntary RPC methods? If so, how does the TTA for a voluntary RPC implementation compare to a copy chain?

1.3.3 Aggregating tree

Can an aggregating tree be implemented in cfengine using the *copy* and *editfiles* actions? If so, how does tree performance compare to chain performance for a similar number of nodes? Can the performance be predicted? Can it be tuned and optimal variable values determined? Can a tree be implemented in cfengine using voluntary RPC methods? If so, how does the TTA for a voluntary RPC implementation compare to a copy tree?

1.3.4 General analysis

Will modeling these patterns in promise theory explain observed behavior? Is there a symbolic expression for determining cost of using these patterns? If so, are there optimal values? How well does the cfengine syntax for these patterns scale? Is there functionality that could be added to cfengine to better support patterns?

Chapter 2

Background and theory

2.1 Navigation patterns

Navigation patterns are defined by the order and direction of inter-node communication in a logical network topology. Specifically, they are characterized by the messaging flow imposed by the combination of an overlay network and communication policies as well as the usage of computing power throughout the network to act on data (as opposed to all calculation occurring in a single node). The product of navigation patterns is a capability to perform management operations on a network *without shared memory* and have both the link usage and the computational intensity distributed throughout the network rather than focused at a single node.

Definition 1 (Navigation pattern)**.** *A graph traversal algorithm executed on a network of programmable nodes.*

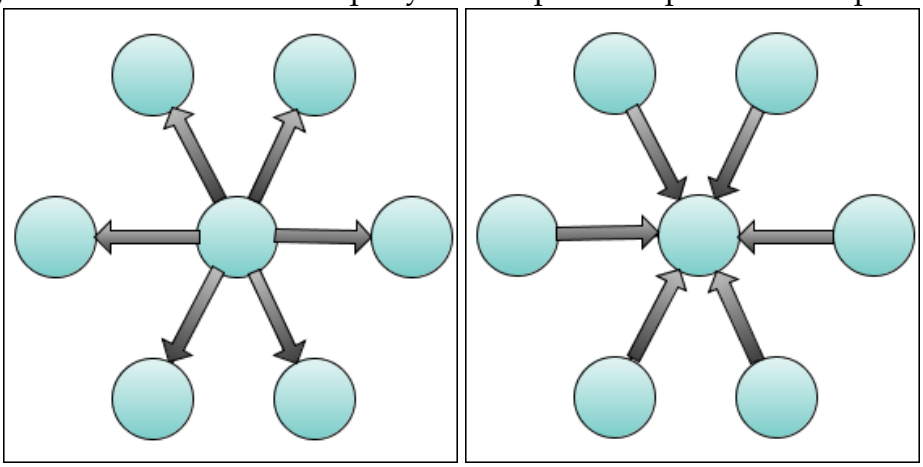
A description of the star pattern as a reference and a brief summary of recently defined navigation patterns follows.

2.1.1 Star pattern

The *star* pattern is a straight-forward example that serves as a good reference for newer patterns because it is not decentralized. The star pattern relies on a

star topology, i.e. a one-level tree. All clients interact directly with the server, in a hub-and-spoke fashion. It is characterized by that central control as well as bi-directional communication between the server and clients. This is illustrated in Figure 2.1 as the server directly queries the clients and they respond directly.

Figure 2.1: The centralized query and response steps in the star pattern.



In particular, the star pattern can be subdivided into two types: serial and parallel. This is remarkable because `cfrun` works in serial by default. This kind of serial pattern can serve to protect a central resource from overloading due to concurrent processes, but the time complexity is very high (as can be seen in experimental results later). However, a parallel star incurs a very high workload on central resources. This will be discussed in more detail later in the thesis.

Patterns should not be confused with the topologies on which they rely. Discussing the star pattern as an example might confuse readers into believing a certain pattern and topology are always related. This is not the case. For example, the patterns described in the following sections could be employed on a star topology as well as in trees. However, the experiments contained herein focus on patterns with specific topology constraints, principally the number of children per node. This will be explained in more detail in the following sections.

2.1.2 Echo pattern

The simplest example of a navigation pattern designed for decentralization is the *echo* pattern. The fundamental topology of the echo pattern is a spanning

tree, with a very central node in the network as the root of the tree. The pattern has two phases of communication: *expansion* and *contraction*. During the expansion phase, the root node issues a query to its children. Each node in the tree repeats this process. The contraction begins as the query reaches a leaf node. The leaf node answers the query, sending its response to its parent in the tree. The parent receives the response of its children, aggregates or calculates information for the query to the fullest extent possible, and then sends a single aggregate answer to its own parent. This process is repeated back to the root node, which finally receives and aggregates the messages from its children. The tree topology provides for parallelized execution while the aggregation of query responses during contraction reduces the amount of traffic that would otherwise be necessary [17].

Ref. [17] provides a formal expression for the time complexity of the echo pattern in terms of *message transition time*, *network delay*, and *processing time*. Equation (1) represents the time complexity for an execution graph $G' : C_{time} = C_{time}(v'_{root})$, where:

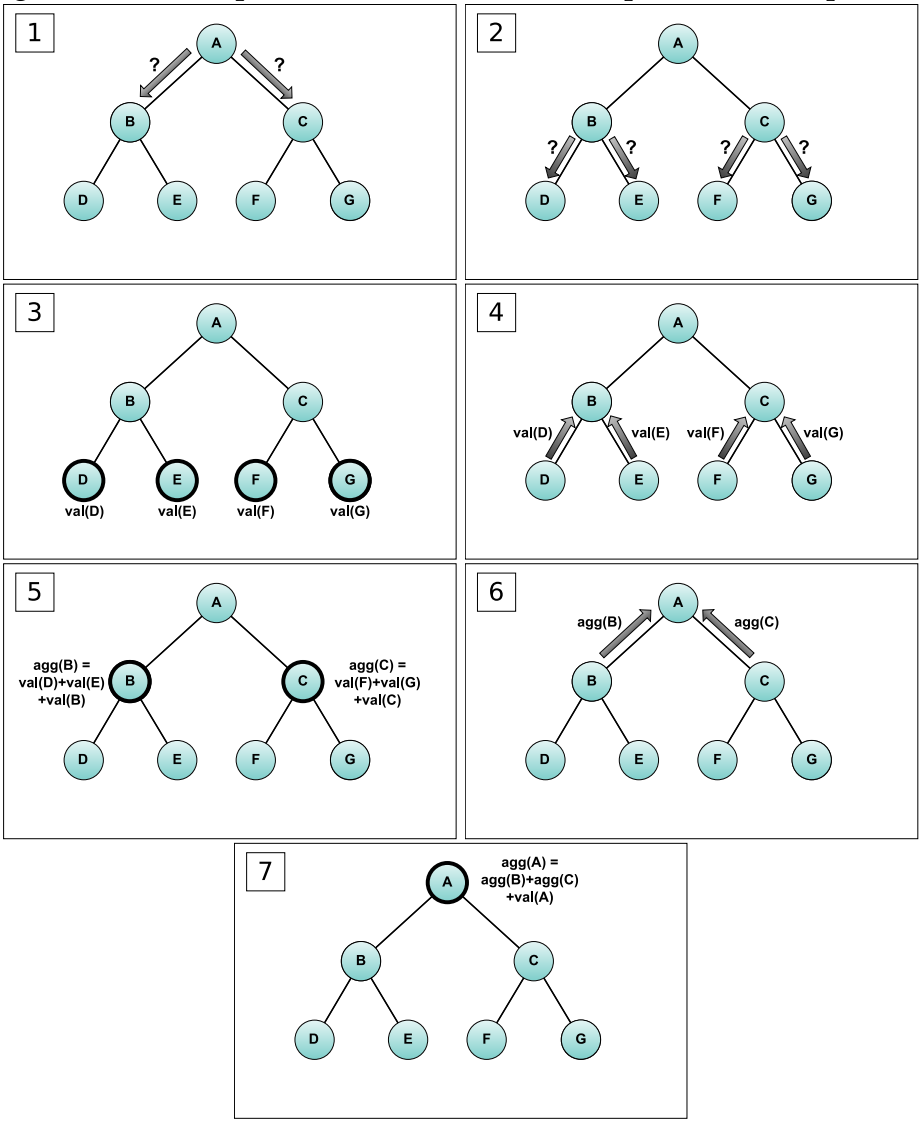
- v' is a vertex in the graph G',
- v'_{root} is the root vertex in the graph G',
- t_q is the total time required to transmit a query to a child (*message transition time*),
- t_r is the total time required to transmit a response to a parent (*message transition time*),
- t_n is the total time required for a message to traverse the path between two nodes (*network delay*),
- t_c is the total time required to compute the local query and aggregation from children (*processing time*).

Equation 1 (Echo pattern time complexity.)**.**

$$C_{time}(v') = \begin{cases} t_c + t_r & \text{if } childcount(v') = 0, \\ t_c + t_r + M(v') & \text{otherwise.} \end{cases}$$

$$M(v')_{1 \leq k \leq childcount(v')} = max\{kt_q + 2t_n(v', child_k(v')) + C_{time}(child_k(v'))$$

Figure 2.2: The expansion and contraction steps in the echo pattern.



Equation (1) shows that the time complexity of a node depends on the time complexity of its children. Because the operations for each child are executed in parallel, the time complexity is largely dependent on the max value of all the children. Therefore, severely imbalanced trees do not stand to make performance gains as substantial as that of well balanced trees.

2.1.3 Generic aggregation protocol

In contrast to the echo pattern in which the managing node queries other nodes in the network, the generic aggregation protocol (GAP) employs a strategy where the managing node is query-passive and receives continuous updates from other nodes in the network. GAP creates a self-stabilizing, breadth-first spanning tree that accommodates for network and node failures by maintaining neighborhood information at each node. The ways in which the neighborhood table at a node changes in response to changes in its neighbors can be modified by policy, which allows for general policy-level adjustment of how the network automatically forms and stabilizes [18].

TCA-GAP

An extension of GAP, TCA-GAP is concerned with aggregated threshold crossing alerts. This method allows for the reduction of total traffic complexity in a network by restricting the number of updates sent to the parents (in terms of the overlay network tree) of the region for which a threshold is defined. The local thresholds are derived from a global threshold and negotiated by neighbors in the overlay. This distributed approach stands in contrast to the typical current practice of monitoring thresholds at only the network edge and processing that information on a managing node, which is often dedicated to the task due to its computational intensity [19].

A-GAP

Another extension of GAP, A-GAP addresses the challenge of specifying a monitoring accuracy level while minimizing the necessary management overhead. Typical rate-control methods reduce traffic complexity but without regard to the impact on accuracy. A-GAP, on the other hand, allows for the specification of an accuracy threshold, below which local filters will prevent

updates from being pushed to the managing node. The global accuracy threshold is first specified on the managing node then distributed and re-computed throughout the network as local filters [20].

Recent and current work

Recent and current work on GAP derivatives has not been reviewed for this thesis but deserve mention because they have potential to be interesting for comparison or application in cfengine. G-GAP [21] provides for network-wide monitoring using epidemic communication strategies to provide robustness. Results indicate that tree-based approaches both perform better and are more robust, indicating that epidemic methods do not necessarily outperform more organized structures. M-GAP is designed to distribute total aggregate data throughout the network; this work is on-going and early discussions of the work with the Prof. Stadler at KTH indicate it will be relevant to cfengine.

2.1.4 Aggregating chain

It is instructive to study the extreme circumstances of any research topic. In the case of tree-based overlay centralization, there are two extremes. The first is the star topology, as described earlier in this thesis. The complete opposite of the star topology is a *chain*, provided the topology is one in which nodes can pass on messages to other nodes. The example of a pattern relying upon a chain topology is used here to facilitate the study of an extreme topology configuration.

Definition 2. *Node degree k: The number of children of a node.*

Definition 3 (Extreme node degrees in tree-based topologies)**.** *. The two extreme cases of node degrees as constraints in trees, excepting the case of 0 children, are chains ($k = 1$) and stars (configurable k). All trees can be considered a combination of chains and stars.*

In a chain, each node with the major exceptions of the root node and the leaf node have exactly one child and one parent. A chain maximizes depth, minimizes breadth, and equalizes workload among nodes (as much as possible

within the constraints of Amdahl's law, which will be discussed later in this thesis). In contrast, star maximizes breadth, minimizes depth, and focuses workload on the central resources.

Conventional wisdom suggests that tree depth corresponds directly to latency in terms of end-to-end communication; chains contain the maximum number of non-repeated hops in a topology and therefore the highest latency on messages passing from one end of the chain to the other. Chains are highly susceptible to failure due to the fact that any individual link or node failure can disrupt end-to-end communications; the closer the failure is to the root, the more substantial the loss.

2.2 Cfengine

Cfengine is an agent-based server/client DSC application as well as a vessel for researching various configuration and monitoring topics at Oslo University College. It is characterized by several features, quoted here from [22]:

- Centralized policy-based specification, using an operating system independent language, which conceals implementation details.
- Distributed agent-based action, in which every host node is responsible for its own maintenance.
- Convergent semantics encourage every transaction to bring the system closer to an 'ideal' average-state, like a ball rolling into a potential well.
- Once the system has converged, action by the agent desists, or more usually, does not even start at all, when convergence was assured on a previous run of the agent.

Cfengine consists of several major components:

cfagent This is the program that actually applies the operators to implement a given policy, which is specified in the files `update.conf` and `cfagent.conf`. `updated.conf` is intended to contain the policy necessary to ensure a running cfengine setup. The actual configuration policy is contained in `cfagent.conf`.

cfservd This is a daemon that serves two purposes. One is to listen for requests from other nodes to execute cfagent. The other purpose is to act as a file server for other nodes that wish to copy files from the node in question. Its configuration file, which includes access control operators, is `cfservd.conf`.

cfexecd This can either be run as a standalone daemon or in non-daemon mode. It is a wrapper for the execution of cfagent. It is preferred to simply running cfagent directly from a crontab because cfexecd can capture and handle output according to policy settings in `cfagent.conf`.

cfenvd This is a daemon used to collect certain statistical data about nodes on which it runs, including statistics about users, load, processes, and sockets.

cfrun This is a command used to initiate cfagent on other nodes. Without arguments, all the hosts in `cfrun.hosts` are contacted serially. However, classes or specific hosts can be specified as constraints.

Cfengine has the ability to effectively either push or pull a configuration from a server to a configuration client. Normally, cfagent is run on each configuration client, which then executes the operators in the `update.conf` before moving to the operators in `cfagent.conf`. This process can include pulling new copies of those files from a configuration server. However, the push is actually simulated; the command cfrun is used to send a request to the cfservd processes on the clients to execute cfagent, which operates in the pull manner described above. This implementation, where the push is actually achieved through a requested pull, is a result of the autonomy-centric *voluntary cooperation* model guiding the development of Cfengine [23].

2.3 Scalability

Generally speaking, scalability can be considered whether the growth of a system approaches some kind of limit that prohibits its function or viability. However, that explanation provides no dimensions so it is important to clarify the terms of its use and measurement for this work. Throughout this thesis, several different factors each provide some notion of scalability. In each case, scaling linearly is optimal.

2.3.1 Workload

First, scalability can be measured in terms of resource consumption on a given node. Low-level resource consumption monitoring method depends on the resource itself. However, in this case, a general characterization of the workload on each node is sufficient and attainable by prediction. The workload metric used for this thesis is defined below:

Definition 4 (Workload)**.** *The number of cfengine processes executing simultaneously on a node related to that node's participation in a pattern test.*

2.3.2 Time to aggregation

A metric specific to patterns, the total time to aggregation is defined below:

Definition 5. *Total Time to Aggregation (TTA): The total amount of time necessary to propagate aggregated data from the leaf of a tree or chain to the root. In the case of push-based protocols, this will include time spent in the expansion/query phase.*

While TTA implies that the entire network aggregate is complete, TTPA is the amount of time for a subsection of the tree or chain to aggregate. Such a sub-aggregate is called a *partial aggregate*.

Definition 6. *Time to Partial Aggregation (TTPA): The amount of time necessary to propagate aggregated data from the leaf of a tree or chain to a node in the network that is lower in the logical network than the root.*

2.3.3 Cost

Another potential limit to scalability is cost-prohibitiveness. Therefore a metric based on the indirect cost related to power is considered in Section 5.6.

2.3.4 Configuration syntax

Configuration syntax is the only aspect of interface scalability considered in this thesis. Syntax is considered to scale poorly if it must be changed as the network changes in terms of growth or failure and if it requires many lines to express the objective. A scalable syntax can be expressed in few lines and does not need to be modified as the network changes.

2.4 Promise theory

Promise theory is a high level description of constrained behavior in which ensembles of agents document the behaviors they promise to exhibit. Agents in promise theory are truly autonomous, i.e. they decide their own behavior, cannot be forced into behavior externally but can voluntarily cooperate with one another[24]. This approach has been used to create an implementation of remote procedure calls that preserves node autonomy, as described in Section 2.6.

Definition 7 (Promises)**.** *A promise is a directed edge* $a_1 \xrightarrow{b} a_2$ *that consists of a promiser* a_1 *(sender), a promisee* a_2 *(recipient) and a promise body* b*, which describes the nature of the promise.*

Promises made by agents fall into two basic categories, promises to provide something or offer a behavior b (written $a_1 \xrightarrow{+b} a_2$), and promises to accept something or make use of another's promise of behavior b (written $a_2 \xrightarrow{-b} a_1$). A successful transfer of the promised exchange involves both of these promises, as an agent can freely decline to be informed of the other's behavior or receive the service.

Promises can be made about any subject that relates to the behavior of the promising agent, but agents cannot make promises about each others' behaviors. The subject of a promise is represented by the promise body b. Finally, the *value* of a promise to any agent is a numerical function of the constraint e.g. $v_{a_1}(a_1 \xrightarrow{+b} a_2)$, and is determined and measured in a currency that is private to that agent. Any agent can form a valuation of any promise that it knows about.

The essential assumption of promise theory is that all nodes are independent agents, with only private knowledge (e.g. of time). No node can be forced to promise anything or behave in any way by an outside agent. Moreover, there are no common standards of knowledge (such as knowing the time of day) without explicit promises being made to yield this information from a source. This viewpoint is important in analyzing the collection of distributed information for measurement purposes.

2.5 Chain propagation delays

The following theory was developed in cooperation with Prof. Mark Burgess and is also discussed in [25].

Consider a sampling/aggregation process scheduled to run every P seconds along a chain, where P is the period (see Figure 2.3), so that the promise compliance frequency is P. In a periodic process, that which happens in one period

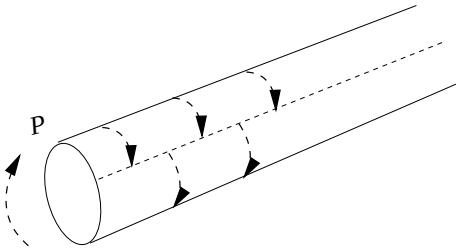


Figure 2.3: Visualizing the periodic sampling process.

is identical (for all intents and purposes) to that which happens in any period. Thus, describing one period is sufficient for describing all of them. This can be turned into a description of the system in terms of a new time variable τ which runs from time 0 to P.

Definition 8. *τ is a time offset, in seconds, of any period P such that $0 < \tau < P$.*

These are related by

Equation 2. $t = nP + \tau,\ n = 0, \pm 1, \pm 2, \ldots$

Equation 3. $\tau = t \bmod P$

The effects of scheduling become clearest when using modulo "clock" arithmetic τ. Let t_n be the real time at which agent a_n wakes up, samples its data and collects data from upstream agent a_{n-1}. The time to keep the promise $t(a_n \overset{-d}{\rightarrow} a_{n-1}) = t_n - t_{n_1}$. In order for promises to be kept transmitted along the chain in a single avalanche, we must have:

Equation 4. $t_n > t_{n-1}, \ \forall n > 1$

i.e. each successive agent must begin at a time that is strictly greater than that of its predecessor. This constraint is easily solved by choosing:

Equation 5. $t_n = t_0 + (n-1)t_s,$

for some offset t_s. The subtlety here is that this schedule has to be wound around the periodic time schedule of the agents. Since each agent is time-shifted by t_s relative to the last, the total time-span of a single schedule is $(N-1)t_s$. As long as this is less than a single period, there will be no phasing between the delays and the periodic activation of the agents. However, in long chains where $(N-1)t_s > P$ some agents will wrap around into the next activation of the schedule and possibly start out of sequence with respect to periodic time τ. Thus, even if Equations 4 and 5 are satisfied in real time, it is not necessarily true that $\tau_n > \tau_{n-1}$. Indeed, this condition will necessarily be violated for some of the agents as long as $(N-1)t_s > P$.

2.6 Voluntary cooperation

Burgess describes a type of policy and protocol designed to preserve the autonomy of nodes in networked cooperative environments [26]. It considers how nodes can interact without one subjugating the other, i.e. preserving the node's autonomy. The method, called *voluntary remote procedure calls (RPC)*, involves two nodes (a server and a client) communicating directly to first negotiate a service agreement. Afterwards, a third party (a *blackboard*) acts as a service broker between the two nodes, with the blackboard receiving requests

from the client and letting the server pull that information from the blackboard. This system essentially uses a publish/subscribe method to ensure that all parties are pulling information from each other rather than directives being pushed upon them by others. In that way, the method preserves the autonomy of each system.

In the case of voluntary RPC in cfengine [26], there are two notable departures from the original model. First, there is no initial negotiation phase in which the server and client communicate directly to make a service agreement; the service agreement is implied and trusted in the form of configuration files. Additionally, the cfengine implementation does not make use of a blackboard, therefore publication repositories on both the server and client serve in place of a blackboard.

The following steps describe the interaction in a cfengine voluntary RPC operation:

1. *Host A* has an RPC method defined to run on *Host B* in `cfagent.conf`. *Host A* publishes the method in a special outbound RPC directory (e.g. `/var/cfengine/rpc_out/`), designated for *Host B*.
2. When *Host B*'s cfagent runs, it checks in `/var/cfengine/rpc_out/` on its *MethodPeers* (as defined in `update.conf`) to find any RPC methods designated for *Host B*. In this case, there is an RPC method waiting on *Host A*. *Host B* downloads this RPC method into an inbound RPC directory (e.g. `/var/cfengine/rpc_in/` and executes it. The results, including return data, are stored in *Host B*'s outbound RPC directory and designated for *Host A*.
3. When *Host A*'s cfagent runs, it checks *Host B*'s `/var/cfengine/rpc_out/` and finds the response to the method. *Host A* downloads the result to its own */var/cfengine/rpc_in/* and processes it according to the policy in *cfagent.conf*.

In the steps above, both hosts are configured (using the *MethodPeers* parameter in `update.conf`) to check each other for RPC methods.

It is worth noting that a principal motivation for using patterns is to reduce the amount of traffic in a network necessary to perform a given monitoring action. However, the voluntary RPC model is shown to have a relatively higher overhead in this respect [26]. Therefore, this work does not aim to combine benefits of voluntary RPC and patterns with the intended result of reducing network traffic complexity.

2.7 Amdahl's law and speedup

Amdahl's Law [27] provides a way to analyze the experiment results and determine what amount of the workload benefits from parallelization. The premise is that any computational task has subcomponents that must occur sequentially and others that can be performed in parallel. Understanding how much of the task is serial is imperative in being able to determine the maximum speedup and vice-versa.

Equation 6 (Amdahl's Law: Observed speedup). $\Psi = \frac{T(1)}{T(p)}$,
where $T(p)$ is the time required to solve the problem with p processors [27].

Equation 7 (Amdahl's Law: Maximum speedup). $\Psi \leq \frac{1}{e+\frac{1-e}{p}}$,
where e is the fraction of the problem that cannot be parallelized and p is the number of processors [27].

Equation 8 (Karp-Flatt Metric: Serial fraction). $e = \frac{\frac{1}{\Psi}-\frac{1}{p}}{1-\frac{1}{p}}$,
where Ψ is the observed speedup and p is the number of processors [28].

Gustafson's Law [29] provides an alternative to the Amdahl speedup model that accounts for a problem size that scales with the number of processors. The analysis in this thesis uses a fixed problem size, which makes Amdahl's speedup more suitable.

Chapter 3

Methodology

3.1 Test network setup

These experiments are designed to test the TTA of different communication patterns. Therefore it was necessary to use a network of hosts sufficient to indicate some problems of scalability. Due to resource constraints, the decision in this case was to create a network of Xen virtual machines [30] on a small set of servers. The utility MLN [31] was used to easily create and maintain 20 Ubuntu Dapper Drake GNU/Linux [32] virtual machines across 5 servers with 4 virtual machines per physical host.

3.2 Measurement and automation of TTA and TTPA data processing

The output of a total or partial aggregate in these tests is a file containing a timestamp of each hop and when it arrived. The TTA can be determined by taking the difference between the timestamp of the root node and that of the oldest leaf node. Similarly the TTPA for any node is the difference between its own timestamp and that of its oldest leaf node. The script used for transforming copy chain output into something suitable as output for graphing is included in Appendix C.2.

3.3 Comparing parallel star and echo using `cfrun`

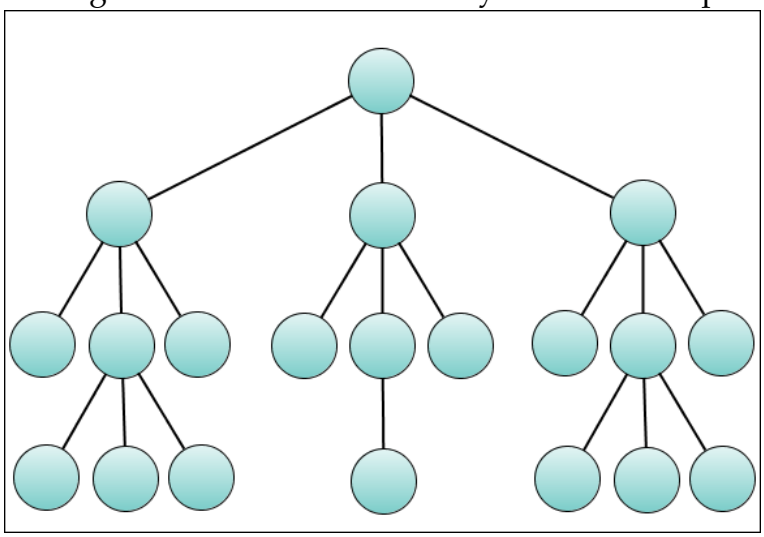
The first step in analyzing cfengine in light of patterns is considering cfengine's behavior that is analogous to the echo pattern, as it is the simplest of the decentralized navigation patterns. By default, `cfrun` works in a centralized manner similar to the star pattern but it runs in serial, not parallel. As such, the implied parallelism of that pattern is not sufficient in describing `cfrun`. For clarification, further references in this work will specify *parallel* or *serial* echo patterns.

The cfengine configuration used to create a serial star is listed in Appendix A.1 while the configuration used to create a parallel star is in A.2. The parallel star `cfagent.conf` issues an individual `cfrun` command to each client in the background. Additionally, the output from each of those commands is redirected to a file. When all the `cfrun` processes have finished, the output files are concatenated together and printed to the terminal so that the parallel and serial star tests both provide nearly identical terminal output. However, it should be noted that the parallel star approach, involving the use of a separate temporary file for each client, involves a great deal more file input and output operations than serial star.

The echo `cfagent.conf` draws from the same framework used for the parallel star, e.g. executing `cfrun` commands in the background with output redirected to files. In this case, a variable is defined for each host that has children. The variable contains a list of the node's children in the tree. If this variable is defined, `cfrun` is called for each child node. Therefore the tree, as illustrated in Figure 3.1, is statically defined. There is a later discussion in this thesis regarding the creation of overlay networks in the context of cfengine.

Each test must have some kind of monitoring or configuration operation. For this test, a hostname query is performed on each client. This is a simple and fast operation without network dependence. There are two important metrics for this test. The first is total elapsed time for the query to complete on the entire network of 20 hosts. The other is workload. A metric in the form of maximum number of simultaneous `cfrun` operations for any node is used to indicate workload. This serves as a very simple indicator of how much work the node is doing. A lower level representation of scalability (e.g. measuring system calls) might be more precise, but using a refined definition of the previously stated workload definition is nonetheless instructive and sufficient in the case of very large factors of difference between the methods.

Figure 3.1: Diagram of 20-node tree overlay used for echo pattern tests.



Definition 9 (Echo workload)**.** *The theoretical maximum number* `cfrun` *processes any node will be running in performance of a given test.*

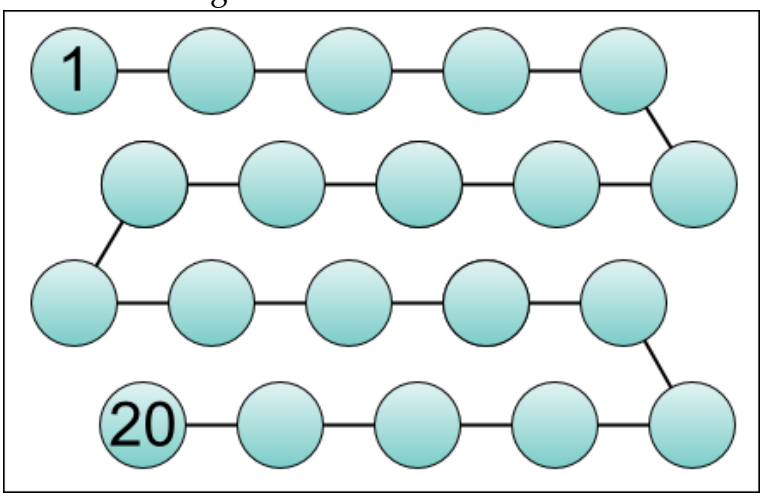
For an indication of uncertainty, each test is performed 50 times.

3.4 Copy chain

The next stage of experiments target a chain in which the nodes autonomously send data to the root at certain intervals, but there is no polling for data by the root node. And so it is more similar to GAP than echo, however without the benefit of topology recovery. The exclusion of topology recovery in this work is by design; implementing topology awareness in cfengine is a formidable task. The work in this thesis reveals certain foundational aspects of pattern behavior that further the understanding of how topology awareness and management could be implemented in cfengine. Considerations for this challenge will be discussed later in the thesis.

Using the same 20 nodes as the echo pattern tests, cfengine on each node is configured into a chain topology, i.e. as shown in Figure 3.2 and the cfagent.conf in Appendix A.4. Similar to the way variables were used to define the tree for the echo pattern tests, for the chain tests a variable containing the hostname of the node's child is defined for each node.

Figure 3.2: A 20-node chain.



The result of the completed aggregation for this test is a file on the root node containing each node's CPU load average as well as the time at which that information was collected. Each node uses the cfengine *copy* action to copy a partially aggregated file from its child. Then the node uses the cfengine *editfiles* action to append its own load data to the bottom of the file.

Because this is a concatenation, it is not strictly an aggregation as proposed by Stadler et al [17]. In such an aggregation, input at each node would be processed in a way that resulted in a single value being passed up the tree rather than a collection of values (e.g. operations such as finding the maximum, minimum, or sum). Such an aggregation was not practical for this scenario because the data collected from each node needed to include a timestamp which could then be used to measure the latency of the update at the root node. All nodes were configured to synchronize time using the network time protocol, which allowed for measurements accurate up to the second. The cfengine configuration ensured that each cfagent would be initialized every minute. Therefore, every node would simultaneously try to obtain an update from its child each minute. Cfengine uses locks to ensure that multiple copies of cfagent are not running at the same time. If cfagent is called and another cfagent is running, the new cfagent will exit.

The tests in the copy chain experiments run every minute but the timing can safely be abstracted to phasing in general (i.e. discussing latency in terms of cycles or steps), provided all the nodes remain relatively time-synchronized and execute cfagent on the same schedules. Two metrics are measured in this case, TTA and time to partial aggregation (*TTPA*).

The expectation of the tests were that the length of the chain would be identical to the number of copy phases. As will be shown later in this thesis, that turned out not to be true. The chain had significantly less latency than originally expected. This led to a further exploration of chains and considerations on their behavior and opportunities for improving chain performance.

Specifically, an *incremental sleep* is added to the copy chain cfengine configuration. For example, node20 is the leaf and will not incur a sleep. However, node19 is node20's parent and will sleep for 1 second before executing its aggregation actions. node18, which is node19's parent, will sleep for 2 seconds. And so the sleep value at the root node will be $N - 1$, where N is the number of nodes in the chain. Taking the tests a step further, a *sleep interval factor* is defined and the the incremental sleep value is multiplied by the sleep interval factor to observe the effect on the update latency of the chain. A higher sleep interval factor corresponds to each node having more time to complete its processing operation; the sleep time serves as a buffer that allows for a sequenced communication from the leaf to the node. Such a coordinated communication has the potential to significantly reduce chain latency. This is in contrast to the copy chain without sleep, in which the communication is not coordinated and therefore will only occur in sequence randomly or by clock shift.

Because a chain is simply a worst-case depth configuration of a tree, discoveries made about chains are also likely to apply to more standard patterns scenarios that rely on tree-based topologies.

3.5 Copy tree

Taking the next step from a copy chain, the experiment that followed concerns expanding the copy chain approach into a binary tree (i.e. a tree with a node degree of $k = 2$). As in the case of the copy chain, the copy tree more closely resembles GAP than echo pattern.

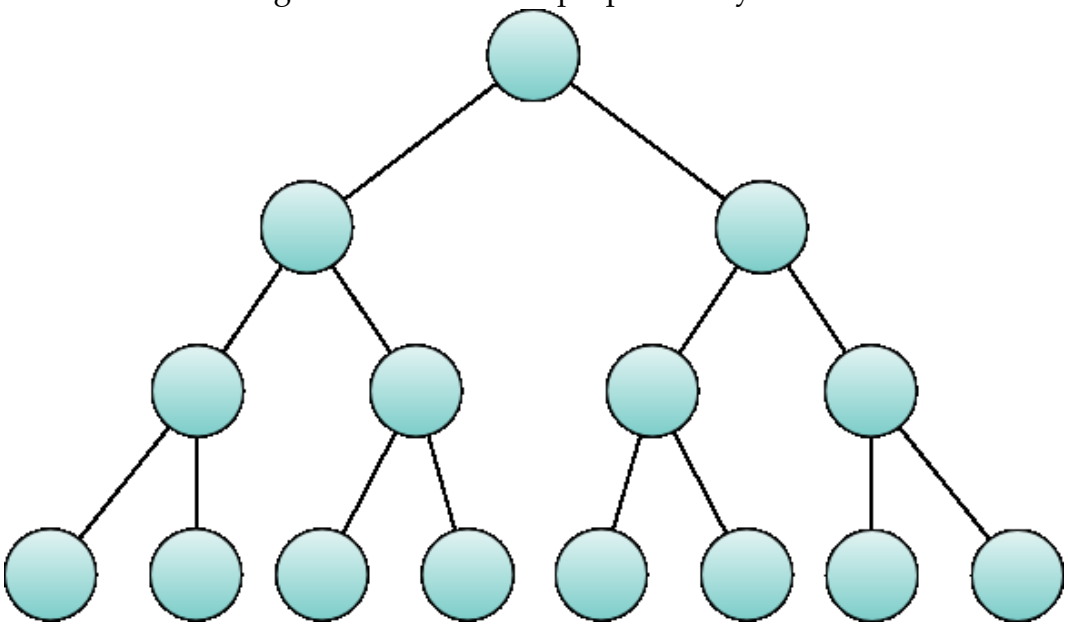
With a similar number of nodes, it is reasonable to expect the copy tree to perform much better than the copy chain. It is also expected to be less susceptible to high-level failures in that half the tree can continue updating the root node, even if the other half is not providing updates.

However, a challenge inherent in any attempt to implement a pattern on a non-chain topology that is not push-based. In current versions of cfengine, the *copy* action is unable to parallelize copies within a single running cfagent process.

Cfengine has no way to perform multiple copies at once. However, the copy tree is partially parallel because the separate nodes are performing their copies in parallel relative to each other.

To have a full binary tree, i.e. a tree where each node has exactly 0 or 2 children [33], 20 nodes could not be used. Therefore, the copy tree tests use 15 nodes, as indicated in Figure 3.3.

Figure 3.3: A 15-node proper binary tree.



3.6 Experimental error and remarks

3.6.1 Standard deviation

The equation used for computing the standard deviation of experimental data samples is listed below.

Equation 9. $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$

3.6.2 Time synchronization issues

On the initial echo pattern tests, unusual timing results were observed in the echo pattern but not the star pattern. Certain nodes appeared to be completing before their children in the same phase, which should not be possible. Looking at the issue in detail revealed that this was consistent to specific nodes and that the clocks on those nodes were lagging behind their children, which caused the aforementioned discrepancy. Because the error was systematic, it could be corrected but I chose to repeat the tests. The data presented for those tests are from the repeated tests, not the adjusted data from the initial results.

Even though the time synchronicity issue was generally solved, further investigation into the matter following the experimentation phase determined that the use of the network time protocol on the Xen virtual machines was suboptimal. Virtual machines, including Xen domains, cannot currently have a system clock with a sufficiently high response time for clock interrupts [34]. Similar future experiments, if using virtual machines, should install the network time protocol on the physical host and ensure that each virtual machine is getting its own time directly from the physical host. In consideration of this, it is likely that there were minor timing errors throughout all tests that effectively introduced random noise in the scheduling.

3.6.3 Cfengine run-time blocking

During copy chain and copy tree tests, sometimes a cfagent process would hang. This would cause an interruption in the pattern, causing the root node not to receive updates from the node with the hung cfagent process or its children. For every run-time phase during which this situation endured, the measured latency of the leaf node data increased one phase period. The manifestation of this, and the symptoms by which it was noticed, was a high standard deviation as well as a very flat, rather than normal, distribution. It was confirmed by checking the aggregated file output, which showed the time at which each node's data was passed up the client or tree. It was also confirmed by viewing the start time of any running cfagent processes on the troublesome nodes. When encountered, the interval that included corrupt data was removed from analysis. The exact cause is unconfirmed; some occurrences corresponded to the approximate time when new cfengine configuration files were deployed (meaning that changing cfengine's configuration files at a precise point in its operation could yield unstable behavior), but not all. The error, though compensated for in this study unless otherwise specified, is a form of

noise and should be considered in further work.

3.6.4 Node failure

Some tests yielded results that were similar in nature to the behavior described for cfengine run-time blocking, though the symptom of having a process age greater than the run-time period was not present. Investigation determined that the troublesome nodes in this case were all virtual machines on the same physical machine and were experiencing a kernel level error, which prevented the nodes from functioning normally in several ways including networking. This was only directly observed twice in a very long series of copy chain tests. Samples that exhibited the same statistical symptoms as cfengine run-time blocking were edited to exclude those anomalies. Burgess shows that in peer-to-peer networks, node failure and network failure are effectively equivalent:

Theorem 1. *A fixed network of partially reliable components, C_i, is equivalent to an ad hoc network of reliable components, on average [35].*

Therefore node failures such as those described here should be given further consideration as they are examples of real-world scenarios that introduce noise, which should be taken into account when building models.

3.6.5 TTA time discrepancy

The TTA measurements collected for this experiment have an inherent inaccuracy. The inaccuracy is surrounding the timestamp used to mark the completion of TTA. All variables available in a `cfagent.conf` are assigned near the beginning of its execution, including the variables containing timestamps used for determining TTA. This means all collected time data refers to the beginning of a cfagent execution, not when it finishes. So in cases where a cfagent process takes minutes to complete, the timestamp will be significantly inaccurate. However, it will be correct as to when the cfagent process started. Therefore, the most correct and accurate usage of this data is to discuss it in terms of run-interval phases, rather than seconds or minutes. The analysis in this thesis does that but will also present models for evaluating TTA data that assumes accurate timestamps.

3.6.6 Sample size

The intended sample size for each test is 50 but in some cases a smaller sample size is used. This variation is due to sleep delays longer than the run interval. For fixed test time lengths, tests with sleep delays longer than the run interval would yield less samples than those tests where the sleep delays were less than the run interval. Data verification (to identify aforementioned errors) and a longer sampling time should have been used to obtain 50 samples for every test.

Chapter 4

Results

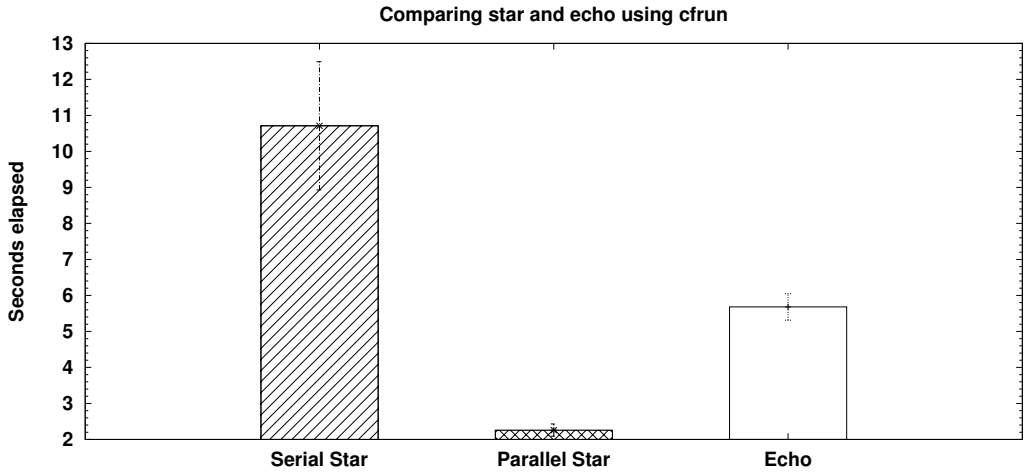
The experimental data is included below for echo and star pattern experiments, copy chain experiments, and copy tree experiments.

4.0.7 Echo/star/`cfrun` experiment results

Figure 4.1 provides a summary of star and echo pattern test results along with a barchart of the values. Serial star has the highest (worst) TTA while the parallel star has the best and the echo pattern is in-between.

Figure 4.1: Average results and standard deviation for 50 tests of each pattern listed. The workload is the number of maximum parallel `cfrun` operations occurring on any node, where lower is better.

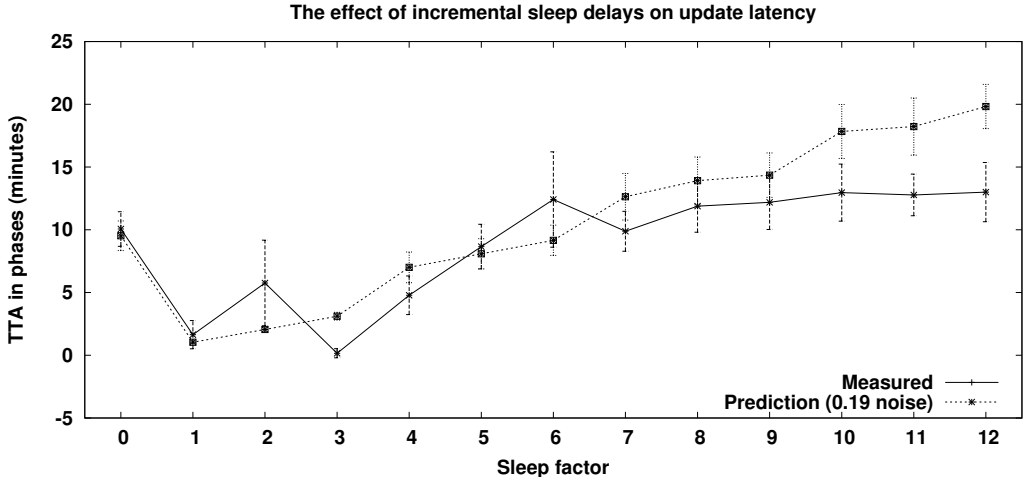
Pattern	**TTA** (minutes)	**Std Dev**	**Workload**
Serial Star	10.71	1.78	1
Parallel Star	2.26	0.17	20
Echo/`cfrun`	5.68	0.37	4



4.0.8 Copy chain experiment results

The following figure shows the average TTA for each sleep factor as well as a plot of the prediction model.

Figure 4.2: Average copy chain TTA for each different sleep factor (standard deviation plotted as the error) plotted against the predicted model with a random noise multiplier of 0.19. Sleep factor 2 is remarkable both because of its high standard deviation as well as its distance from the prediction. Sleep factor 3 is remarkable due to it being the lowest of all value and low standard deviation. Sample sizes for each measurement are shown in Table 4.1.



Additionally, histograms of the measurements for each sleep factor are shown in Figure 4.0.8. The value in these histograms is to indicate some similarity to a normal distribution, which is confirmed in most of the histograms. The notable exceptions are sleep factors 2 and 3.

Figure 4.3: Histograms showing the distribution of measured copy chain TTA for sleep factors from 0 through 12.

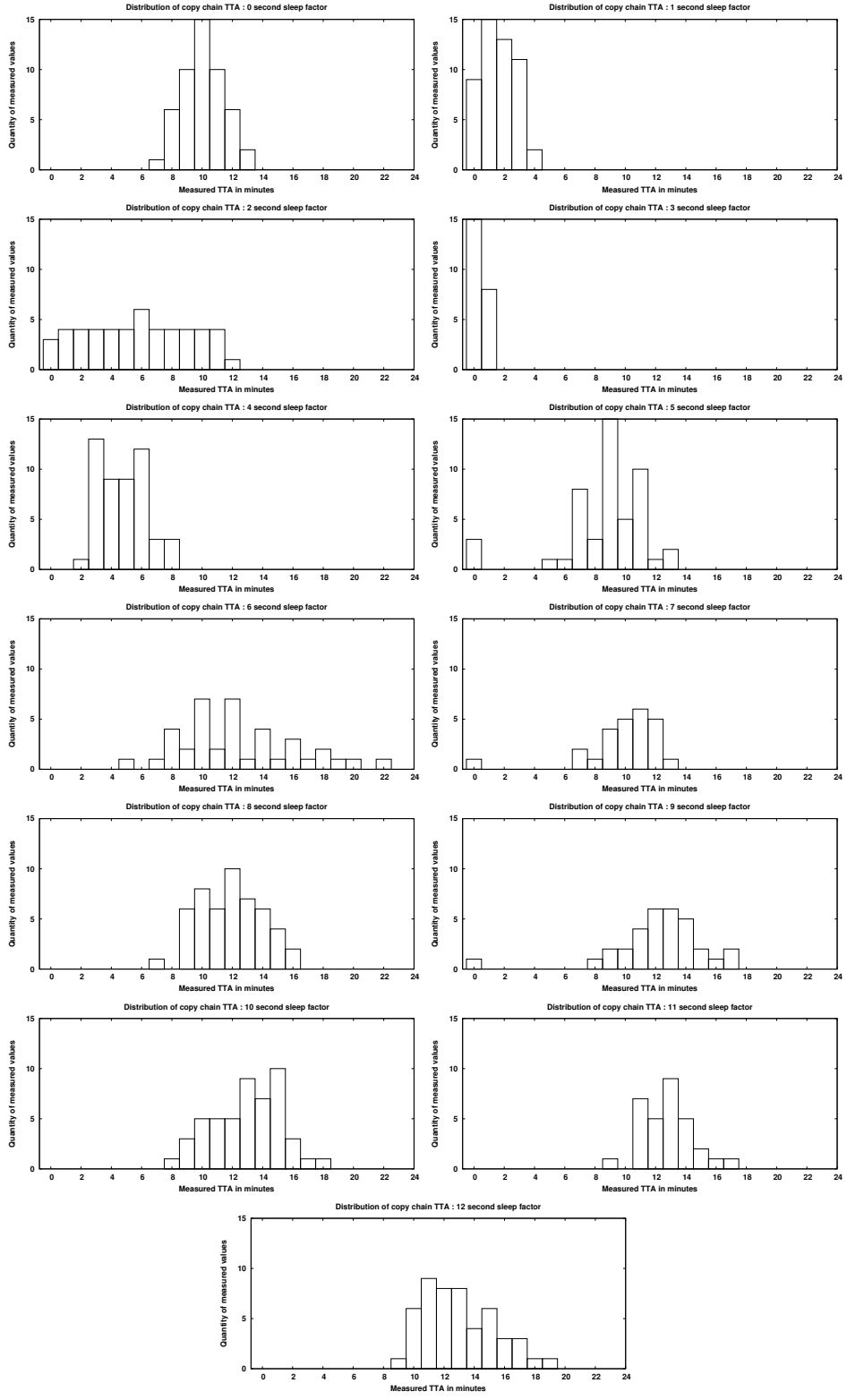


Table 4.1: The sample size for each sleep factor in the copy chain tests. 50 was used when that data was available. Smaller samples are due to sleep delays longer than the run interval and therefore less samples within a given time period (discovered after sampling was complete).

Sleep factor	**Sample Size**
0	50
1	50
2	50
3	50
4	50
5	50
6	39
7	25
8	50
9	32
10	50
11	31
12	50

Because of the pseudo-periodicity, each period demonstrates similar behavior. Therefore certain sleep factors can be categorized together. τ can be defined as a sleep delay within a period, therefore the TTA at similar τ values of different periods should be similar. Table 4.2 shows the τ values for each sleep factor categorized into τ *groups*. These values are plotted and compared in Figure 4.4.

Table 4.2: The sleep factor for the copy tree as seconds delayed on the root node and then converted to minutes. The τ *group* creates an association between sleep factors based on τ, defined here as the number of seconds beyond the most recent full run interval period. Members of the same τ group are expected to have similar propagation latency after accounting for full period rotations.

Sleep factor	**Node20 Sleep in Seconds**	**Node20 Sleep in Minutes**	**τ Group**
0	0	0:00	
1	19	0:19	A
2	38	0.38	B
3	57	0:57	C
4	76	1:16	A
5	95	1:35	B
6	114	1:54	C
7	133	2:13	A
8	152	2:32	B
9	171	2:51	C
10	190	3:10	A
11	209	3:29	B
12	228	3:48	C

τ Group	**Step 1**	**Step 2**	**Step 3**	**Step 4**
A	1.64	4.78	9.88	12.96
B	5.76	8.66	11.88	12.77
C	0.16	12.41	12.19	13.00

Figure 4.4 indicates what appears to be a convergence point near 13. This is unexpected, because the predicted behavior would indicate that the τ group plots should be near-linear and parallel to each other, as seen in Figure 4.5. Therefore, an analysis was conducted on the TTPA value at each node in the chain. For each sleep factor, the average TTPA was computed for each node, along with the standard deviation. This data is plotted in Figure 4.6 and the τ groups

are compared in Figure 4.7. The graphs are generally linear after exceeding the utility of the sleep factor as buffer (represented as the near-horizontal component of the plot).

Figure 4.4: TTA of observed copy chain τ groups compared. This graph suggests a convergence for all the τ groups at a value of approximately 13 during the fourth phase of each period.

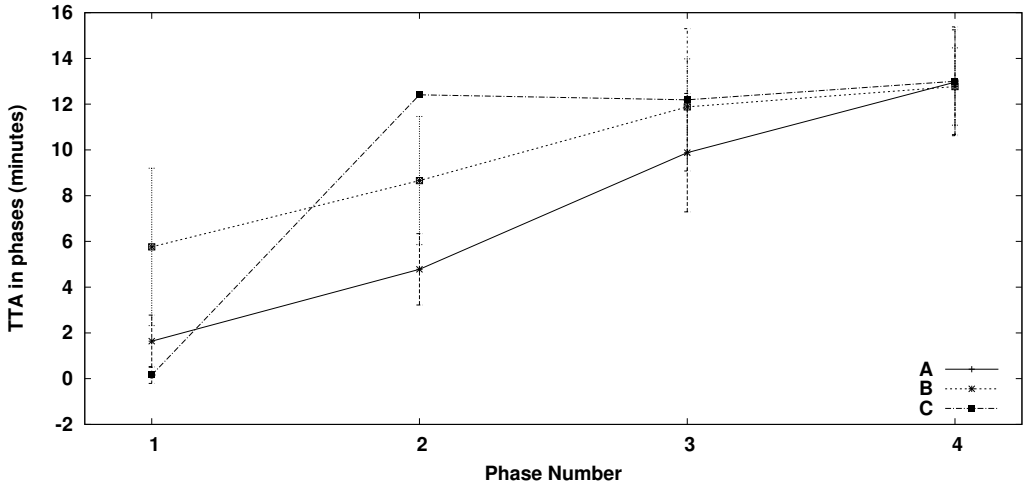


Figure 4.5: TTA of predicted copy chain τ groups compared.

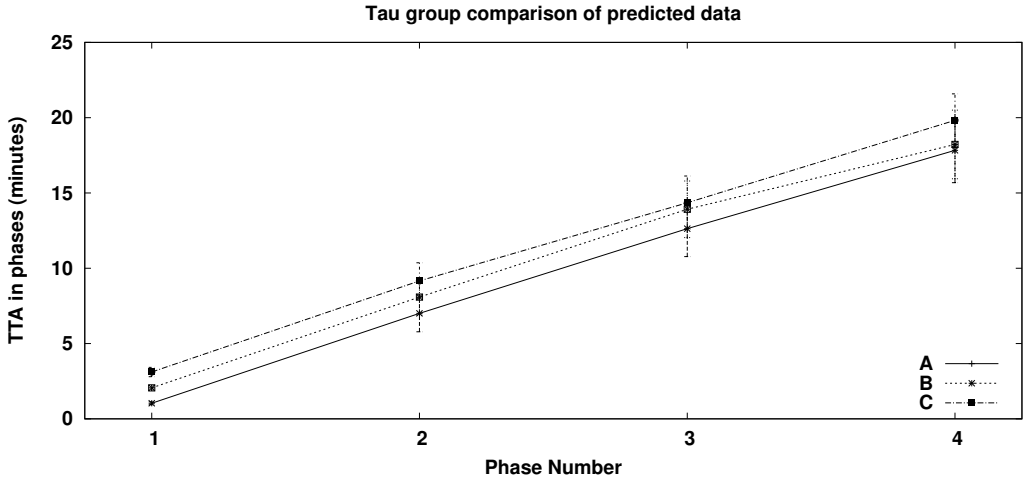


Figure 4.6: Average copy chain TTPA plotted against distance in the chain from the leaf node. Note the atypical behavior exhibited in sleep factors 2 and 3.

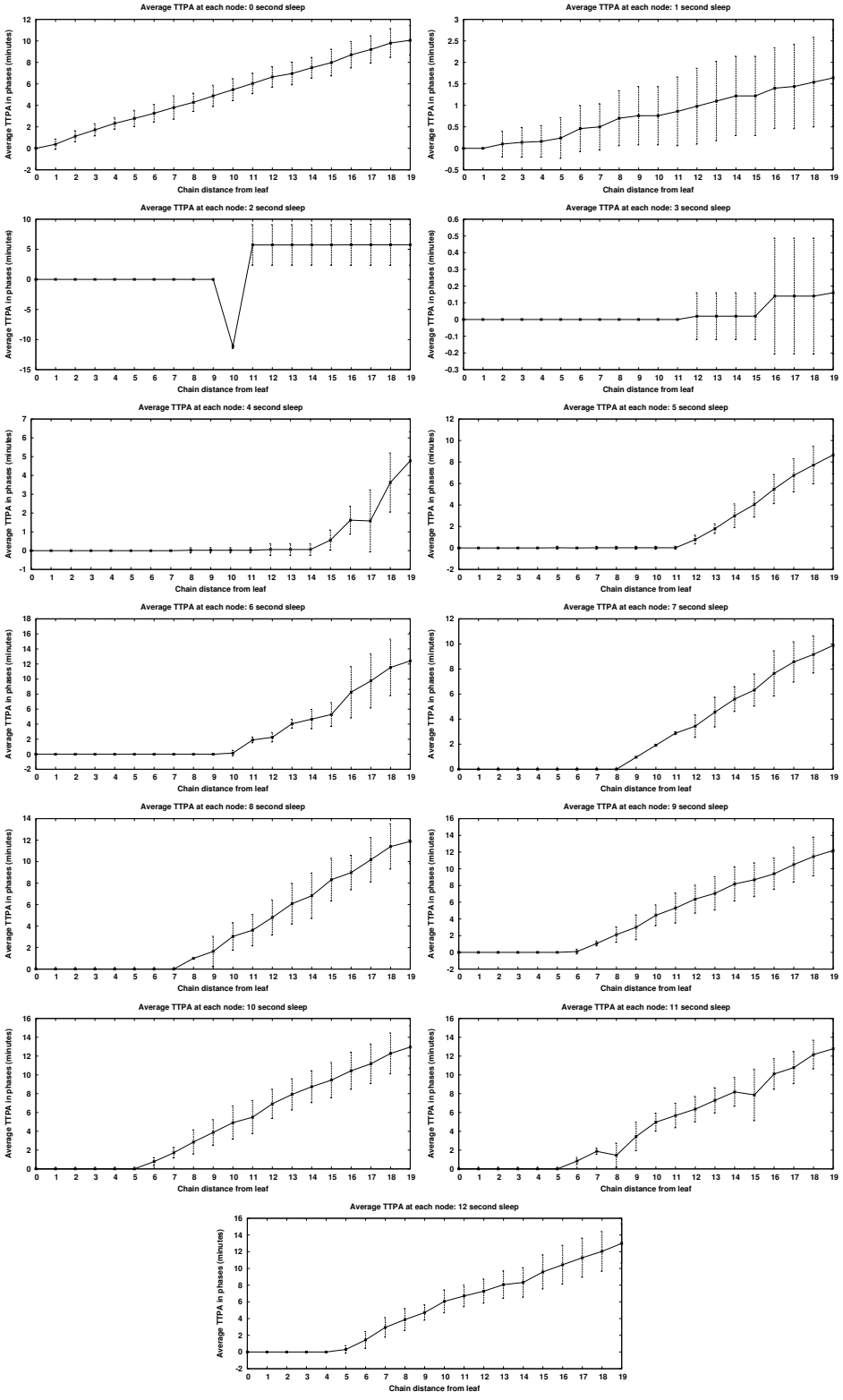
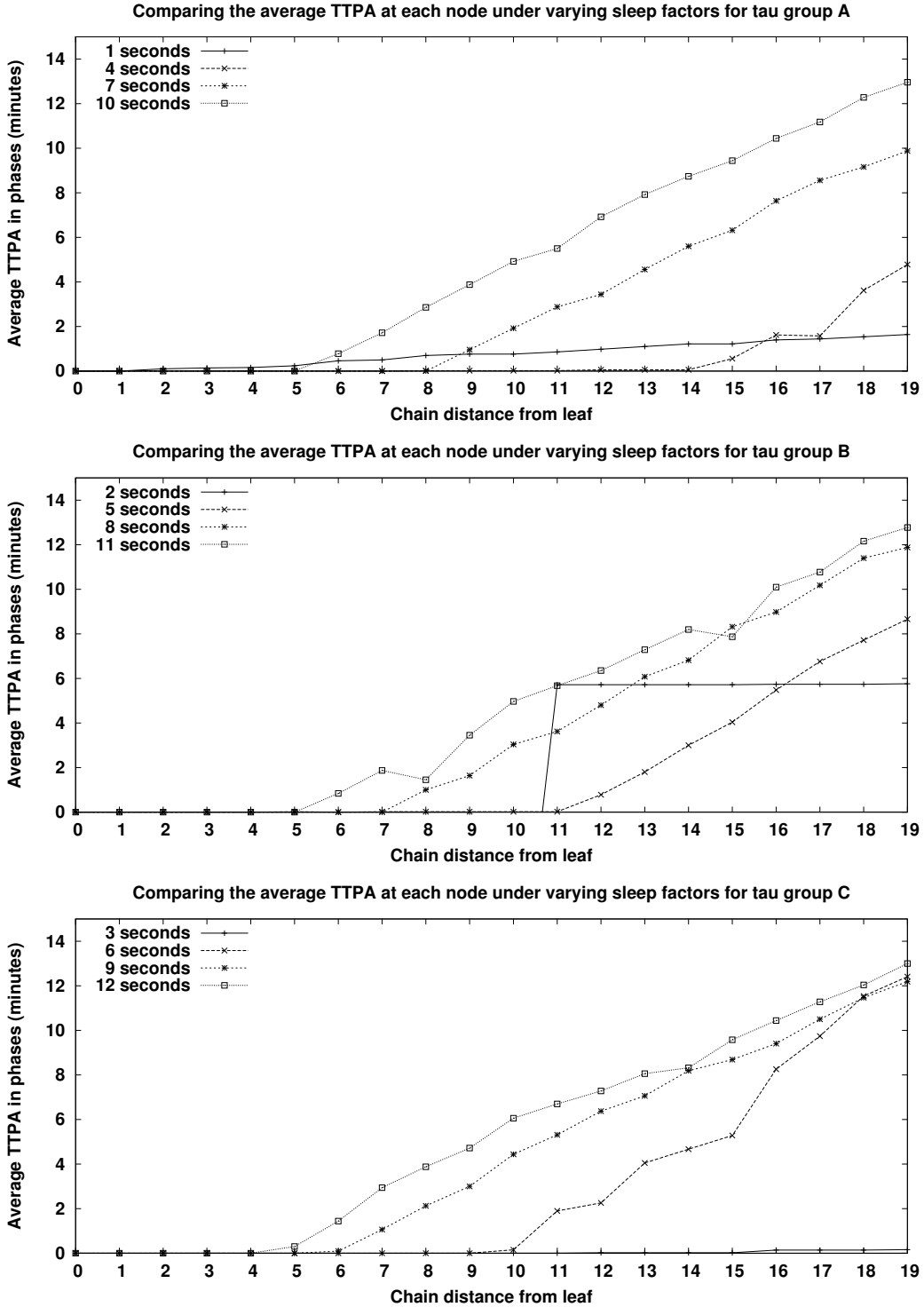


Figure 4.7 shows the average TTPA for each sleep factor, grouped together by similar τ. This shows linear and generally parallel TTPA scalability among the τ groups, with the exceptions of sleep factors 1-3.

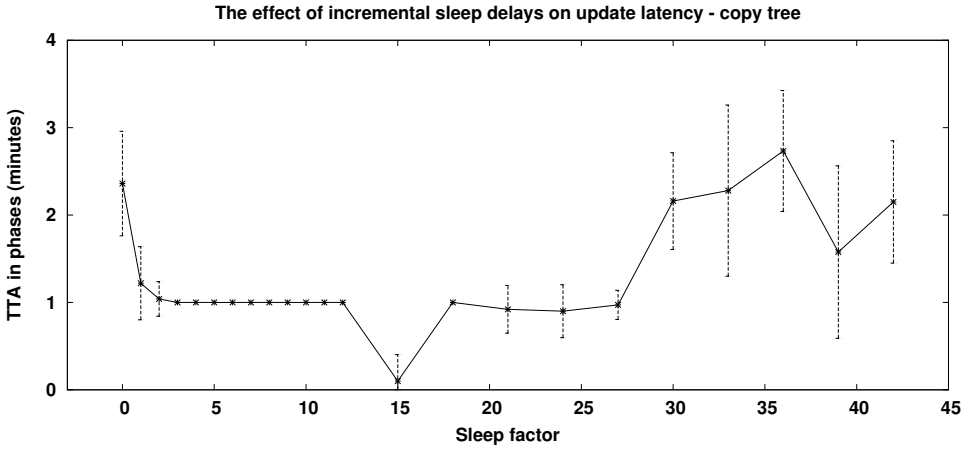
Figure 4.7: Copy chain τ groups compared: average TTPA plotted against chain distance from leaf node.



4.0.9 Copy tree experiment results

Figure 4.8 shows the average TTA for each different sleep factor in the copy tree. Note the stabilization at sleep factor 3-12 and then the dip at sleep factor 15. As the sleep factor increases, there is more variation and greater uncertainty.

Figure 4.8: Average copy tree TTA for each different sleep factor with standard deviation error. Table 4.3 shows the sample size for each sleep factor.



Histograms showing the very small distribution spread of the TTA at each sleep factor are included as Figures 4.9 and 4.10.

Table 4.3: The sample size for each sleep factor in the copy tree tests.

Sleep factor	**Sample Size**
0	50
1	50
2	50
3	50
4	50
5	50
6	50
7	50
8	50
9	50
10	50
11	50
12	50
15	50
18	50
21	38
24	50
27	36
30	25
33	25
36	30
39	26
42	40

Figure 4.9: Histogram showing the distribution of measured total time to aggregation for sleep factors of 0 through 12.

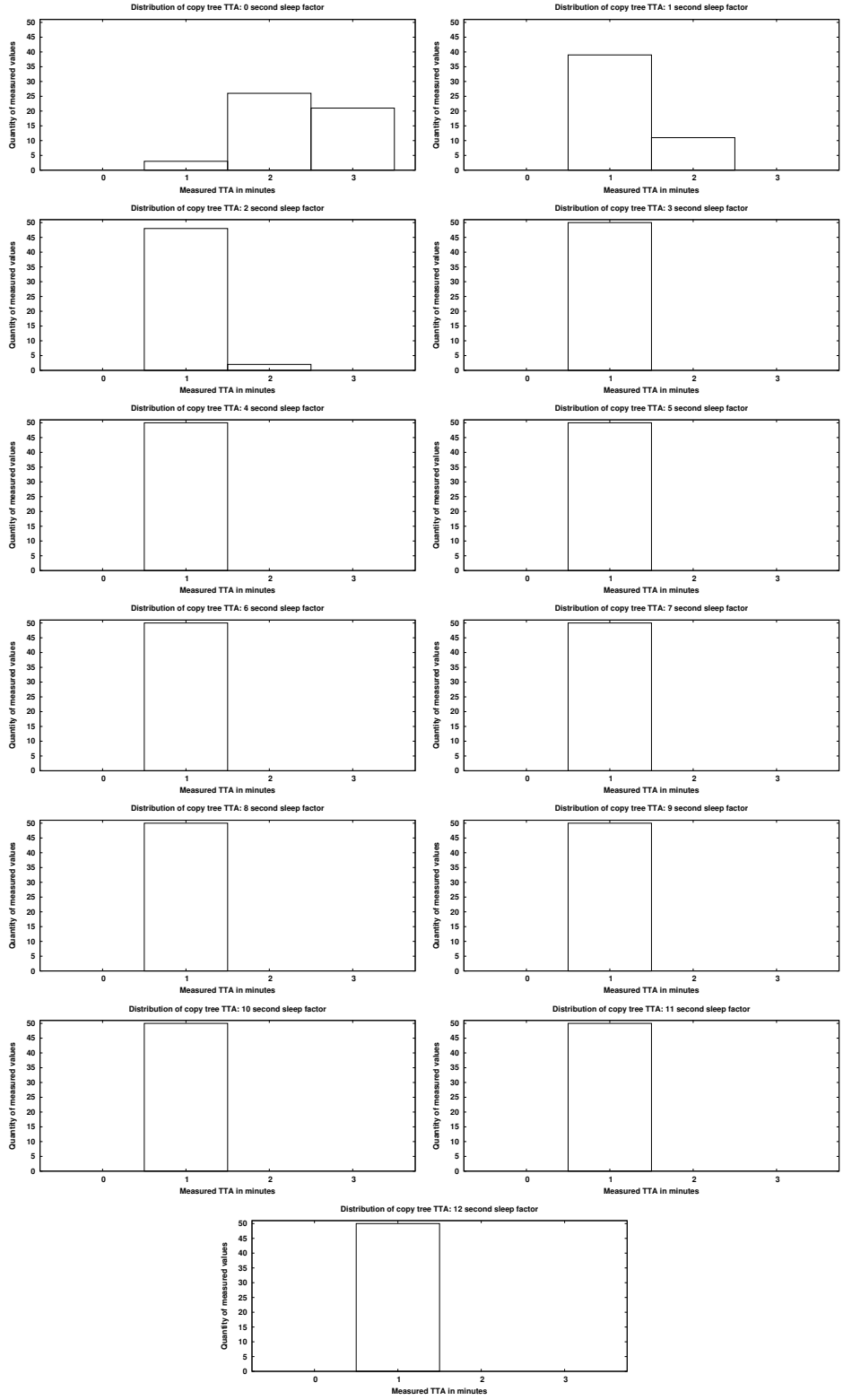
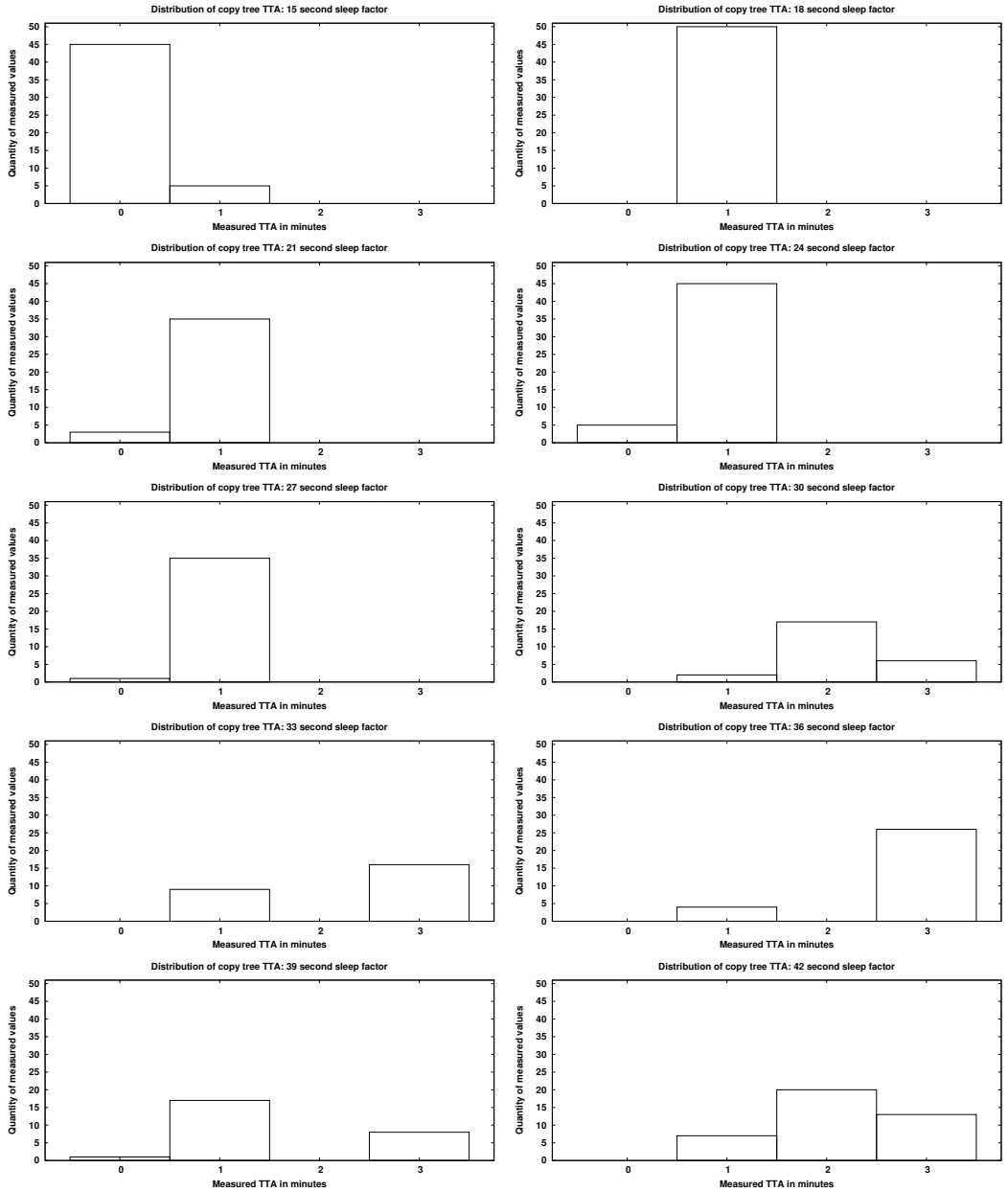


Figure 4.10: Histograms showing the distribution of measured total time to aggregation for sleep factors of 15 through 42, at an interval of 3.



Chapter 5

Discussion

5.1 General remarks on push- vs. pull-based results

It is remarkable that the TTA for the push-based results, including the star and echo patterns, are significantly less than the pull-based results, which include the copy chain and copy tree. Indeed, the TTA for the push-based patterns are measured in seconds while that for the pull-based are measured in minutes. This is not surprising, as the performance of any pull-based approach is going to rely on the poll interval. In these experiments, the pull-based poll interval was a minute. A poll interval of 1 second, which is not currently available in cfengine and for general purposes not recommended, would provide for the ability to directly compare the timing results for push-based and pull-based patterns in this thesis. Otherwise, such a direct comparison is invalid.

5.2 Echo/star/`cfrun` experiment

The results of the `cfrun` experiments comparing serial star, parallel star, and echo are shown in 4.0.7. The serial star performs the worst, naturally, but scales very well on the server side because it is always only performing one operation. The parallel star performs the best of the three methods, but it will suffer from bottlenecks on a large scale; the server is always running as many `cfrun` commands as there are nodes, including itself. Finally, echo offers both a re-

markable performance improvement and a workload improvement that is, in this case, a factor of $\frac{1}{5}$; specifically, the echo workload will always be $k + 1$, because there will be an operation running for each child as well as the parent process, because it is a polling operation and so there must also be a parent operation running that triggered the query to the children.

5.3 Copy chain experiment results

5.3.1 TTA characterizations

The initial results from the copy chain experiment were surprising because they did not require N number of phases to commute data from the leaf to the root. Test results show that, without a sleep delay, the average TTA of leaf node data on the root node was approximately 10 phases rather than the expected 19 phases. See Figure 4.2.

Investigation into this surprisingly good performance uncovered the cause: some adjacent nodes in the chain were randomly being executed in sequence at a very small interval. For example, node19 is configured to aggregate data from node20. Cfagent is executed on both hosts every minute. In some cases, node19 will execute one second after node20 simply due to timing synchronization issues or low-level resource contention issues delaying the execution of cfagent on node19. When this happens, node20 will have had enough time to complete its own aggregation routine when node19 copies its data. This ambiguity can be considered random scheduling noise.

5.3.2 The effect of sleep factors on TTA

The response to discovering the noise was to perform experiments that used an incremental sleep factor as a buffer to reduce the TTPA and thereby reducing the TTA, as discussed in Section 3.4. Very large sleep factors were included because the periodic behavior produced by sleep delays that overlap run-time intervals for software that safely handles such timing conflicts is largely unexplored.

The results, shown in Figure 4.2 are promising. That plot includes a model developed in cooperation with Prof. Mark Burgess, the advisor on this work,

which is discussed in detail in Section 2.5. It is an effective predictor of the observed pseudo-periodic behavior.

5.3.3 Sleep factor 2 results analysis

The histogram for sleep factor 2 (see Figure 4.0.8) is unusual in 2 ways. First, it is flatter than normal. The expected behavior at this sleep factor is a more concentrated distribution rather than a flat one. Second, the values are out of place given the context of the values for sleep factor 1 and sleep factor 3; while it is not always the case that this should be true, here it is expected because the incremental time buffer increase is not crossing a period threshold.

5.3.4 Sleep factor 3 results analysis

The histogram for sleep factor 3 is also unusual. However, in contrast to the histogram for sleep factor 2, it is very concentrated. Indeed, the histogram for sleep factor 3 is essentially the desired outcome: a high occurrence of TTA that is within a single phase. Therefore, based on the collected data, a sleep factor of 3 can be considered optimal for the function being aggregated in this case. However, it is important to remember that sleep factor suitability is subject to function being aggregated.

5.3.5 Periodic groupings and comparison

Figure 4.4 indicates what appears to be a convergence point near 13. This is unexpected, because the predicted behavior would indicate that the τ group plots should be near-linear and parallel to each other, as seen in Figure 4.5. To explain this, there are two possibilities. The first is that there is some kind of limit at 13, to which sleep factors beyond 12 will converge as well. The second possibility is simply that this is coincident and not actually representative of the data.

To make a determination as to which is the case, an analysis was conducted on the TTPA value at each node in the chain. For each sleep factor, the average TTPA was computed for each node, along with the standard deviation. This data is plotted in Figure 4.6 and the τ groups are compared in Figure 4.7. If

there were a limit being approached, these graphs would indicate some kind of asymptote. But that is missing and all the graphs, with the exception of that for sleep factor 2, indicate linear scaling. Thus it is reasonable to expect higher sleep values to scale linearly; clearly there is no limit at 13 phases.

However, these data plots reveal some erroneous data in the sleep factor 2 tests that was not as obvious from earlier analysis. By manually checking the data, a time synchronization error can be seen in node10 being 11 minutes behind the other nodes, resulting in a negative value for the TTPA. The data has been left uncorrected in this case to further justify the TTPA analysis. Additionally, and perhaps due to a related cause, node10 only received an aggregate from node11 approximately every 10 minutes. The exact cause for the high periodic latency is unknown because the error was not discovered until the tests were complete but it can be explained by a potential misconfiguration of the run-time interval (so that node10's cfagent ran every 10 minutes rather than every 1 minute).

5.4 Copy tree experiment results

Studies of chains, while useful for studying fundamentals of tree-based patterns, is an unlikely topology in real distributed systems. In most cases one would expect a node to be able to connect to several other nodes and allow a greater centralization of data during aggregation. Therefore, the copy chain experiments have been repeated on a binary tree.

The results are shown in Figure 4.8. Clearly, for a similar number of nodes to the copy chain, the TTA for a copy tree is much lower. The number of varied sleep factors had to be increased for the copy tree, relative to the copy chain tests, to show periodicity. Notice that the TTA stabilizes at a value of 1 phase from sleep factors 3-12, then makes a drop to 0. This stabilization at 1 shows that the aggregated function, in this case a sequence of serial copy actions and then a file concatenation via an editfiles action, cannot be completed within the given poll interval. At a sleep factor of 15, it can. The next sleep factor step, 18, pushes the root node sleep delay close to 60 seconds and the TTA for the tree is likely to accordingly exceed that to a small degree, which causes delays from cfengine's duplicate run-time prevention.

The jump between the TTA values prior to a sleep factor of 27 and that of 30 and later is not fully understood. It is possible that those data points are marred by an insufficiently large sample size, though the generally higher la-

tency than lower sleep factors is obviously expected.

The histograms and standard deviation indicate that the tree is less sensitive to sleep factor as well, suggesting that the tree may be a more stable option to a chain.

5.5 Amdahl's law and speedup

Observed speedup values, determined by Amdahl's law, along with the serial fraction, determined using the Karp-Flatt metric formula are shown in Table 5.1. In the case of the parallel star, 83% (or $1 - e$) of the serial star workload benefitted from parallelization. The sequential component includes the aggregation computation at the center node and the input/output operations that follow (e.g. writing information to files and terminal but not including any temporary file creations that can occur in parallel). Considering the actual computational intensity for the tested operation (merely echoing the hostname of each node) is so low, the value $e = .17$ is surprisingly high. For a star pattern of 20 hosts and a seemingly low computational intensity, a ratio closer to 1:20 rather than 1:5 would be expected. Therefore, it is reasonable to expect high communications overhead.

The echo pattern speedup and serial fraction are included for reference. The serial component is much higher than that for the parallel star. More information on communications overhead would be needed to further validate this data, but it could be useful for future experiments.

Figure 5.1: Average results and standard deviation for 50 tests of each pattern listed. The workload is the number of maximum parallel `cfrun` operations occurring on any node, where lower is better. The calculations for the aforementioned values are included in Appendix B.

Pattern	**TTA** (minutes)	**Speedup**	**Serial fraction**
Serial Star	10.71	0	1
Parallel Star	2.26	4.74	.17
Echo/`cfrun`	5.68	1.89	.50

5.6 Cost function

The following theory was developed in cooperation with Prof. Mark Burgess and is also discussed in [25].

Another aspect of scalability is the indirect cost associated with delivering performance goals. Is there an argument for choosing pattern dimensions according to cost? Is there an optimum answer? This thesis does not provide a definitive answer to these questions, as such concerns are a matter for policy. However, consider the following.

The rate of power consumption of a node is proportional to its CPU frequency[36] squared. Thus if a node maintains a minimal CPU frequency and dynamically increases that frequency to cope with demand from aggregation of k neighbors, indirect cost scales as k^2 which represents power, cost of cooling or shortened battery life, etc. The risk, on the other hand, associated with not getting data quickly is proportional to the effective depth of the network pattern $(N-1)/k$. So there is a cost function that is a balance between these two as demonstrated in Equation 10:

Equation 10. $\text{Cost}_{1<i<N} = v_i \left(\bigoplus_i (a_i \overset{-d_i}{\rightarrow} a_{i-1}) \right) = \alpha \left(k_i^{(-d)} \right)^2 + \frac{(N-1)}{k_i^{(-d)}}.$

A plot for the arbitrary policy $\alpha = 0.1$ is shown below. This shows the existence of an optimum aggregation degree, in this case $k = 5$. Such arguments should also be taken into account in the scaling argument, as the cost can be seen rising sharply with increasing centralization.

5.7 Promise theory

The following application of promise theory to patterns was created in cooperation with Prof. Mark Burgess and is also discussed in [25].

5.7.1 Propagation model

Consider the following promise designations:

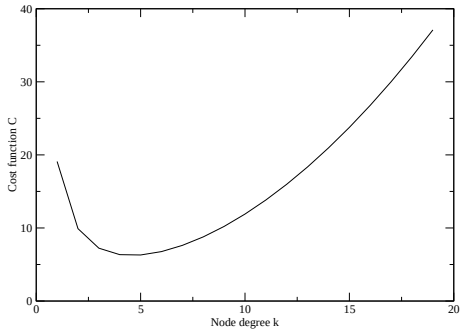


Figure 5.2: Cost considerations can plausibly lead to an optimum depth of network pattern when power considerations are taken into account. The minimum cost here is given for $k = 5$. Such considerations require an arbitrary choice to be made about relative importance of factors.

- $+d$ Server provides data,
- $-d$ Client receives/uses data,
- $+a$ Branch node aggregates data,
- $+t$ Server provides time/clock, and
- $-t$ Client uses time/clock.

When referring to network "nodes" below, it will be understood that each node is modeled as an "agent" in promise theory parlance.

Assume an underlying network substrate with partially reliable communication. The interactions between the agents form directed graphs which can be typed with the promise labels. A forward pointing promise graph forms a transpose adjacency matrix

Equation 11. $a_i \xrightarrow{b} a_j \;\Leftrightarrow\; A^{\mathrm{T}(b)}_{ij}$.

Since the communication requires bilateral $\pm b$ promises, there is an implicit promise graph of opposite sign whose structure is the transpose of the matrix above.

Nodes can, in principle, have any in- or out-degree equal to or greater than zero. It is conventional to restrict network patterns to tree structures however so that the data flow promises of type $+d$ form a monotone confluence. The adjacency matrix $A_{ij}^{(-d)}$ for the forward part of Figure 5.3 and similar ones can be used to represent the typed node degrees through the relations:

Equation 12. $A_{ij}^{+d} = A_{ij}^{(a)} = \left(A_{ij}^{(-d)}\right)^{\mathrm{T}}$

Equation 13. $k_i^{(+)} = \sum_j \left(A_{ij}^{(+)}\right)^{\mathrm{T}} = \sum_j \left(A_{ij}^{(-)}\right)$

Figure 5.3 shows the overlain complementary trees of $\pm$ data promises.

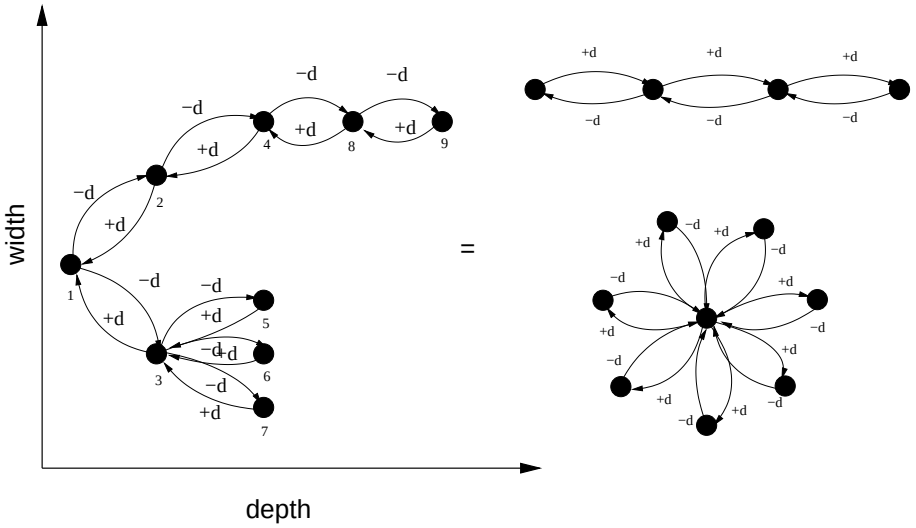


Figure 5.3: A bilateral promise tree of type $\pm d$ indicating "depth" and "width". The structure can be thought of a cross between the star topology and a chain, which are the extreme cases.

5.7.2 Kinematics of the system

The rate and timing of the information flowing through the promise network structure is important to its observable properties. There are two processes taking place that 'interfere' with one another.

- Sampling process at each node.
- Propagation of data from node to node.

If the stream is not synchronized by waiting, information will not propagate faithfully along the full length of the chain. This is obvious in a one-shot enactment of the promises, but the matter becomes much more interesting in the case where the promises are implemented in a periodic schedule. If nodes have memory of previously measured values then one can obtain the illusion of propagation of data, but at the price of having undetermined or inaccurate results.

Start by considering the basic rates of transport in the promise structures. Then consider the time it takes to transport across network depth and the time it takes to aggregate across network width as separate processes. By dimensional analysis[37], it is clear that the behavior of the system will depend essentially on the various dimensionless ratios that can be formed from the basic scales of the system.

5.7.3 The sampling process

Let q_i be a variable at node a_i and $\frac{dq_i}{dt}$ be the rate at which q_i changes in time. Assume that time increases at the same rate for all observers, within the limits of measurement; in other words, all agents have clocks that run at the same rate. This is a more likely approximation to the truth than assuming that they are all synchronized. Finally, let Q_i mean the size in bytes of the representation of the value of q_i that has been measured. This is important to know when transporting the data over fixed capacity channels.

Use t_i to mean the time at which a_i sampled the value of q_i, and let P_i be the periodic sampling rate of the agent node a_i, so that after ℓ samples, the time will be $t_i + \ell / P_i$.

This generality is suggestive of the earlier copy chain results. For the present, however, consider the idealized case in which all agents sample at the same basic rate P, the period of the system.

Promise $a_i \overset{+d}{\rightarrow} a_j$

Let $C_{ij}^{(+d)}$ be the rate at which a_i can transmit data to a_j. This is a property of the communication channel between the agents as well as the agents' own limitations. We now introduce the promise-valuation function $t(a_i \overset{b}{\rightarrow} a_j)$ to be the time to complete the promise $\langle a_i, b, a_j \rangle$. Then the time to transmit the data is:

Equation 14. $t\left(a_i \overset{+d}{\rightarrow} a_j\right) = \frac{Q_i}{C_{ij}^{(+d)}}$

Promise $a_i \overset{-d}{\rightarrow} a_j$

Let $C_{ij}^{(-d)}$ be the rate at which a_i can receive data from a_j. Then the time to receive the data is:

Equation 15. $t\left(a_i \overset{-d}{\rightarrow} a_j\right) = \frac{Q_j}{C_{ji}^{(-d)}}$

Agent constraints on $\pm d$

It must be generally true that the rate at which a promise is used is less than or equal to the rate at which it is provided.

Equation 16. $C_{ji}^{(-b)} \leq C_{ij}^{(+b)}$

for any promise body b. Moreover, if we let $C_i^{(\pm b)}$ be the maximum communication capacities of the agents for sending/ receiving (a property of the agents

rather than the channel between them) we must have the additional constraint that, if an agent has several neighbors, the sum of communications with all neighbors cannot exceed the agent's own capacity:

Equation 17. $\sum_j A_{ij}^{(-b)} C_{ij}^{(-b)} \leq C_i^{(-b)}, \quad \sum_j (A_{ij}^{(+b)})^{\mathrm{T}} C_{ij}^{(+b)} \leq C_i^{(+b)}$

Promise $a_i \xrightarrow{+a} a_j$ (aggregation)

Each node that receives data from more than one source promises to aggregate the information according to some algorithm, which is deployment-specific and shall not be specified here. We define the aggregation to be the following computation:

Equation 18. $a_i \equiv q_i + \sum_k A_{ij}^{(-d)} a_j, \quad i, j = 1 \ldots N$

For leaf nodes, the second term is zero and the aggregation is simply equal to the contribution from the agent node itself. The recipient of this promise might be the aggregation agent above the promiser in the aggregation tree, or it could be an agent external to the tree, such as a policy agent or an external observer. The final recipient at the top of the tree makes this promise either to itself or to a suitable policy agent or observer.

This aggregation promise has a number of consequences. The result of the computation is dependent on the values collected by the $-d$ promises of neighbors. Thus there must be a strict ordering of events before the promise can be completed in a given time frame. We must always remember that the aggregating agent knows only what it has been promised in a scheme of voluntary cooperation.

The semantics of the aggregation promise must now be defined. Several alternatives present themselves:

1. The node polls its sources in turn.
2. The node aggregates from its sources in parallel.
3. The node does not promise its result until all sources have kept their promises to provide data, i.e. we have conditional promises:

Equation 19. $a_{i-1} \overset{+d/a}{\rightarrow} a_i,\ a_i \overset{a/-d}{\rightarrow} a_{i+1}$

4. The node does not wait for its sources and provides its best answer and there is no condition on the dependents in the chain:

Equation 20. $a_{i-1} \overset{+d}{\rightarrow} a_i,\ a_i \overset{a}{\rightarrow} a_{i+1}$

These details are important to fully describe the propagation of data within a network structure.

The time to complete the aggregation promise is straightforwardly given by Equation 21:

Equation 21. $t\left(a_i \overset{+a}{\rightarrow} a_j\right) = \frac{Q_i + \Sigma_k A_{ik}^{(-d)} Q_k}{C_i^{(a)}}$

5.8 Supporting patterns in cfengine

Existing syntax for patterns in cfengine does not scale well because the entire topology has to be expressed in policy files. The challenges for this, along with proposed solutions, are described as follows.

Scalability of described method

Appendix A.3 demonstrates a configuration for the tree used in the echo pattern tests; the tree is described using classes to conditionally define a variable that lists the children for each node. There are two undesirable traits of this approach.

First, the syntax does not scale well because it has to be modified each time the tree changes. This might be due to node or link failure or simply the addition of a new node, for example. Second, the expression itself is relatively long. At least two lines are required for each node that has children. Ideally, the tree definition could be automatically computed and distributed. However, there is also a bootstrapping problem inherent in initially communicating the topology to all nodes.

Bootstrapping problem

Sites using cfengine must initialize their policies out-of-band, that is, not using cfengine. Once they have a valid `update.conf`, cfagent can download future policy updates from servers specified in the `update.conf`. There are at least three options for bootstrapping a topology for cfengine:

Pre-seed using out-of-band distribution. The tree can be designed and deployed to all the nodes using an out-of-band distribution method. This has the benefit of the nodes assuming the desired topology from the beginning of the deployment but requires relatively more manual intervention.

Pre-seed star, then update to tree. The initial out-of-band policy could be a star. During the first update, download the topology information (which would have been seeded on the central host) and assume the appropriate role in the topology thereafter. This requires less manual intervention but has an initially high load on central resources.

Service discovery by broadcast or multicast. Each cfengine node could announce a request for topology. If a response comes from a trusted node, included topology information would be loaded. This method has the benefits of little manual intervention and no centralized resource load spike. However, the topology information will need to have been seeded on at least a single node, and that information will propagate through the network slowly (relative to aforementioned approaches).

The two former approaches are currently available in cfengine. However, the latter approach is most interesting both from a research and an autonomics perspective. In combination with GAP, it would allow for a completely hands-free topology. This is because the methods above do not take into account how the topology will be initially defined. GAP includes a specification for this [18], which could be implemented in cfservd.

Cfengine interface to network and execution graphs

Based on the understanding provided from the work in this thesis, a cfengine node only needs to know about its immediate neighbors and potentially, for the purposes of performance optimization, its distance from its leaf node. This clarification allows for several syntax suggestions.

The first suggestion is to separate the topology graph from the execution graph. This is the easiest to immediately accomplish in cfengine and also allows the most control via policy. The topology would be automatically distributed through cfservd announcements and responses (as mentioned previously) but there would be no automatic execution based on this information; the administrator would need to use the topology information explicitly. An example cfservd.conf is listed in Figure 5.4 with a corresponding cfagent.conf in Figure 5.5. This is a cfengine-centric approach.

Figure 5.4: A `cfservd.conf` with potential topology and patterns options.

```
control:

  # Only trust graphs provided by these hosts:
  TrustGraphsFrom = ( $(trustedhosts) )

  # Our trustedhosts will be queried for topology
  # information. One ore more of them will provide
  # this node with information that populates
  # the variables children, parent, and root.

  # Only allow connections from trusted hosts,
  # peers, and the root.
  AllowConnectionsFrom = ( $(trustedhosts) $(children) $(parent) $(root) )

grant:
  # Allow our trustedhosts and peers access to
  # operation-critical files and directories.
  /var/cfengine/inputs  $(trustedhosts) $(children) $(parent) $(root)
  /var/cfengine/bin/cfagent $(trustedhosts) $(children) $(parent) $(root)
  /var/cfengine/rpc_out $(trustedhosts) $(children) $(parent) $(root)

  # Allow aggregates to be copied by the parent
  # or children nodes.
  /var/childrenfiles $(parent)
  /var/parentfiles $(children)
```

Figure 5.4 shows how the security approaches used for the `MethodPeers` option (for listing voluntary RPC peers [38]) and other cfservd security options such as `AllowConnectionsFrom` can be similarly used for an option called `TrustGraphsFrom`. The network nodes listed in this option would be the ones from which cfservd is willing to trust information about graphs (both topology and execution graphs, i.e. patterns).

The copy action in Figure 5.5 shows two important suggested enhancements. The first is the creation of a new built-in variable that coincides with cfengine's context-sensitive list iteration. This variable, called `$(_)` here, would be a reference to the current list item during a list iteration. This provides for a looping mechanism very similar to a `foreach` construct and the underscore as the variable name is taken from Perl's default variable which serves the same purpose: `$_`. The other suggested enhancement is the capability to run a copy action in

Figure 5.5: A `cfagent.conf` with potential topology and patterns options, including echo pattern support and full parallel generic aggregation support made possible by an enhanced copy action able to iterate through a server list like a `for` loop and copy in parallel.

```
control:
  actionsequence = ( shellcommands copy tidy )


classes:
  # Define a class if we have children, using
  # the variable inherited from cfservd.
  HasChildren = ( IsDefined(children) )

shellcommands:
  # Execute cfrun if the push class is defined
  push::
    "/usr/local/bin/cfrun $(children) -- push \
     2>&1 /tmp/echorun.$$"  background=true
    "/usr/bin/pgrep cfrun > /dev/null;
     while [ $? = 0 ]; do pgrep cfrun > /dev/null; done"
    "/bin/cat /tmp/echorun.*"

copy:
  # Copy some files from our parent,
  # moving data from the root to the
  # leaves.
  /var/parentfiles
    dest=/var/parentfiles
    server=$(parent)
    elsedefine=parentdown

  # This is a iterative copy that is
  # currently unsupported by cfengine.
  # Copying data from the children,
  # moving it from the leaves to the
  # root.
  /var/childrenfiles
    dest=/var/childrenfiles/$(children)
    # The $(_) is a proposed variable that would
    # always reference the current element in an
    # iterated list, similar to Perl's $_ default
    # variable.
    server=$(_)
    background=true

  # If the update from the parent fails,
  # try to copy the files from the root node.
  parentdown::
    /var/parentfiles
      dest=/var/parentfiles
      server=$(root)

tidy:
  # Clean up after our push.
  push::
    /tmp pattern=echorun.* age=0
```

the background or parallel. This functionality already exists in shellcommands actions. Such an option for copies would allow for the transmissions between parents and children to occur in parallel, thus providing patterns that work in parallel.

There are several attractive features of the aforementioned enhancements to cfengine, in addition to the general assertion that the additions would behave as described. First, the changes do not affect the scope or general syntax level of cfengine. Second, the new options or functionality are consistent with existing cfengine security models. Finally, by separating any instruction set from the exposure of graph information, node autonomy is preserved.

Another suggestion for implementing patterns in cfengine is not as easy to conceptualize. It would mean giving the pattern precedence over the policy; the pattern would be defined first and then the policy executed on that graph. This is a patterns-centric approach that does not seem to fit well with cfengine, but could be accomplished by effectively adding a patterns layer (or in practical terms, an encapsulation or wrapper) on top of the existing cfengine concepts.

These suggestions merely touch upon what is a very complex issue with many details that are similar to challenges in mobile ad-hoc network routing. If implemented, certain suggestions could require significant restructuring of cfengine functionality. This topic is therefore recommended for further study.

Chapter 6

Future work

6.1 Exploring cfengine enhancements for patterns

Section 5.8 includes suggestions for improving support for patterns in cfengine. A continuation of this work would be to implement these or similar suggestion and repeat the copy tree experiments in this work. Trees with different node degrees should also be considered. Also, the topology management components of patterns should receive integration attention and evalution as well.

6.2 Voluntary RPC trees

Burgess has proven the concept of voluntary RPC trees on a small scale [38], but future work should implement them on a larger scale and similarly analyze phasing issues. It is possible that a voluntary RPC GAP implementation in cfengine will yield similar phasing results to those described for the earlier copy chain analysis. However, a voluntary RPC echo pattern implementation is certain to be more time-consuming than the one described in this thesis. Specifically, when considering the echo pattern time complexity as described in Equation 1, a periodic poll timing parameter will need to be added. Interesting questions include whether or not phasing can be manipulated, as in this study, to reduce TTA.

6.3 Implementation of more patterns

This work has focused on echo and GAP, the foundational patterns. Once GAP has been suitably implemented in cfengine, patterns derived from GAP should also be implemented and evaluated in cfengine.

6.4 Experiment on a larger scale

Experimentation on a much larger scale would be interesting to determine if there are unpredicted limits to specific applications of patterns in cfengine. PlanetLab[39] is a global scale cooperative research grid that may be an effective setting for such work. It would also provide a realistic scenario in terms of node and network reliability.

6.5 Economic interactions and policy

There is an economic aspect to dynamic cooperative distributed systems and patterns should be considered through this lens. Promise theory, economics, and various network interaction models provide a framework for this. Agents should be able to opt out of participation in a pattern (because the agents are autonomous) but it is currently unclear what currencies should be evaluated by nodes in a pattern to scope their involvement. Understanding this can provide motivation for the development of quality of service components.

6.6 Autonomic resource distribution

Future work that implements topology recovery in cfengine will naturally be in the direction of autonomic networking. Additionally, sensors could be added to cfengine that provide appropriate context and decision support for dynamically reorganizing the topology so that the nodes with the most resources are placed appropriately.

Chapter 7

Conclusions

This study of patterns in cfengine has proven to be instructive work. Cfengine was not originally intended to extend into a lower layer to manage the flow of communication among nodes in a network. And patterns were not conceived with voluntary cooperation in mind. In terms of implementation, there is no doubt that bringing these two technologies together is, as the idiom provides, trying to fit a square peg in a round hole. However, while the implementations discussed here have not been completely true to the original patterns in all senses, patterns are clearly complementary to cfengine. This work shows that patterns can be used to success in cfengine to offset scalability limitations and costs.

The push-based echo pattern was fully implemented in cfengine. Evaluation showed a significant improvement in time to aggregation over the serial star pattern while providing a much lower workload than the parallel star pattern. Parallel speedup analysis indicated a surprisingly high serial work component for the aggregated function. This implementation shows that patterns with true parallelism can be accomplished in cfengine but is not endorsed for production use or as the focus for future work; it violates the premise of voluntary cooperation while the generic aggregation protocol offers more flexibility on that point as well as topology recovery.

A limited form of the generic aggregation protocol was implemented in cfengine in two topology configurations, first a chain and then a tree. The chain study showed better time to aggregation than anticipated due to random scheduling noise. Adding an incremental sleep factor to each node in the chain, so that each node would sleep longer than its child, was successful in buffering the noise and providing a stable and low time to aggregation. Some data sug-

gested a convergence and a time to aggregation ceiling, which would have been important to understand. Further analysis on the time to partial aggregation at each node in the chain invalidated the hypothesis that a limit exists and showed linear scalability.

The corresponding study showed the scalability benefit, in time to aggregation, of using a limited pattern on a binary tree topology in cfengine. Tests on total time to aggregation with incremental sleep values were also performed and determined a range of near-optimal sleep factors. The form of the generic aggregation protocol in cfengine, both for chains and trees, is not recommended for current production use currently as it is weak in its parallelism and also does not include topology recovery. However, this topic should be a focus of future work as the protocol's definition includes topology recovery and cfengine would foreseeably benefit generally from the capability to parallelize copy actions. These experiments show the linear scalability in even a partial implementation of the generic aggregation protocol.

An important next step that this thesis aimed for but did not attain is the implementation and analysis of voluntary RPC trees in cfengine. However, appropriate background information and suggestions for future work on voluntary RPC is included.

There was some experimental uncertainty in the results due to a variety of errors. The direct errors are now well-understood with the minor exception of a single test (the sleep factor 2 test in the copy chain tests). In a repeat of these experiments, the existing scripts and graphing strategies could be used to validate the data immediately after the tests complete. Some of the higher sleep factor tests had sample sizes smaller than 50; this should be corrected in repeats of this work or related future work by extending the test duration as necessary to yield enough samples.

The experiment design, while generally sound, was constrained by inherent limitations in cfengine. For example, the copy tree tests should have been conducted on a star topology as well as a tree with a higher node degree but a lack of support for parallel copy actions in cfengine makes this a low priority. Additionally, better low-level resource monitoring in experiments could provide more information about the communication overhead in terms of parallel speedup.

Modeling patterns using promise theory provides a quality-of-service perspective that reinforces the importance of analysis at the local node level because there are not transitive service promises.

Consideration of difficulties in implementing patterns in cfengine provides some proposed direction for future integration work. The manual interaction necessary in using patterns in cfengine is generally discouraging; providing an interface that automates the automatic formation and management of a network topology and the application of patterns will promote adoption and reduce likelihood of error.

Over the long term, research in implementing patterns in cfengine will promote the cause of scalable autonomic network and system management. Cfengine already contains a significant amount of context-awareness sensors and in the coming years it is reasonable to expect the addition of policy-driven autonomic actions based on the information from such sensors. In the future, a network of autonomous cfagents that are aware of topologies, execution graphs, and system resource consumption could cooperate without human intervention to modify the overlay network or re-provision services as necessary to distribute system and network resource load as appropriate throughout the network. Foreseeable related research includes integrating service auto-discovery techniques, reducing overhead in voluntary scenarios, auto-negotiated quality of service dynamics, optimized overlay network auto-negotiation for further performance improvements, and reductions in failure recovery times.

This thesis commences a new discussion on the fundamental behavior of patterns in periodic execution scenarios and promotes the integration of patterns and cfengine. The outcome includes propositions for further implementation and analysis of patterns in cfengine and, hopefully, additional foundation for continued discussions and cooperative research.

Bibliography

[1] R. Evard. An analysis of unix system configuration. *Proceedings of the Eleventh Systems Administration Conference (LISA XI) (USENIX Association: Berkeley, CA)*, page 179, 1997.

[2] P. Anderson. Large scale system configuration (lssconf) homepage. http://homepages.informatics.ed.ac.uk/group/lssconf. Accessed March 5, 2006.

[3] S. Traugott and J. Huddleston. Bootstrapping an infrastructure. *Proceedings of the Twelth Systems Administration Conference (LISA XII) (USENIX Association: Berkeley, CA)*, page 181, 1998.

[4] P. Anderson. Why configuration management is crucial. *;login:*, 1:5–8, 2006.

[5] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[6] K. Stone. System cloning at hp-sdd. *Proceedings of the Large Installation System Administration Workshop (USENIX Association: Berkeley, CA, 1987)*, page 18, 1987.

[7] Hal Stern, Mike Eisler, and Ricardo Labiaga. *Managing NFS and NIS*. O'Reilly & Associates, Inc., Sebastapol, CA, 2001.

[8] Tivoli systems/IBM. *Tivoli software products*. http://www.tivoli.com.

[9] P. Anderson, G. Beckett, K. Kavoussanakis, G. Mecheaneau, and P. Toft. Technologies for large-scale configuration management. http://www.gridweaver.org/WP1/report1.pdf. Accessed March 5, 2006.

[10] A. Couch and M. Gilfix. It's elementary, dear watson: Applying logic programming to convergent system management processes. *Proceedings of the Thirteenth Systems Administration Conference (LISA XIII) (USENIX Association: Berkeley, CA)*, page 123, 1999.

[11] P. Anderson. Towards a high level machine configuration system. Proceedings of the Eighth Systems Administration Conference (LISA VIII) (USENIX Association: Berkeley, CA):19, 1994.

[12] M. Harlander. Central system administration in a heterogeneous unix environmentl genuadmin. *Proceedings of the Eighth Systems Administration Conference (LISA VIII) (USENIX Association: Berkeley, CA)*, page 1, 1994.

[13] Imazu Hideyo. Omniconf - make os upgrade and disk crash recovery easier. In *LISA '94: Proceedings of the 8th USENIX conference on System administration*, pages 27–32, Berkeley, CA, USA, 1993. USENIX Association.

[14] J.P. Rouillard and R.B. Martin. Config: a mechanism for installing and tracking system configurations. *Proceedings of the Eighth Systems Administration Conference (LISA VIII) (USENIX Association: Berkeley, CA)*, page 9, 1994.

[15] M. Burgess. Cfengine - a configuration engine. *University of Oslo, Dept. of Physics report*, 1993.

[16] J. Kramer, J. Magee, and M. Sloman. Configuration support for system description, construction and evolution. In *IWSSD '89: Proceedings of the 5th international workshop on Software specification and design*, pages 28–33, New York, NY, USA, 1989. ACM Press.

[17] K.S. Lim and R. Stadler. A Navigation Pattern for Scalable Internet Management. In *Proceedings of the 7th IFIP/IEEE Symposium on Integrated Network Management*, May 2001.

[18] M. Dam and R. Stadler. A generic protocol for network state aggregation. *Radiovetenskap och Kommunication (RVK)*, June 2005.

[19] F. Wuhib, M. Dam, R. Stadler, and A. Clemm. Decentralized Computation of Threshold Crossing Alerts. In *16th Workshop on Distributed Systems: Operations and Management (DSOM)*, pages 220–232, 2005.

[20] A. Gonzalez Prieto and R. Stadler. Distributable Real-Time Monitoring with Accuracy Objectives. Technical report, KTH Royal Institute of Technology, Sweden, 2005.

[21] Fetahi Wuhib, Mads Dam, Rolf Stadler, and Alexander Clemm. Robust monitoring of network-wide aggregates through gossiping. In *The Tenth IFIP/IEEE International Symposium on Integrated Network Management (IM 2007)*, May 2007. To Appear.

[22] M. Burgess. Recent developments in cfengine. The Hague, 2001. Unix.nl Conference.

[23] Mark Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. In *DSOM*, pages 97–108, 2005.

[24] Mark Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. In *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775*, pages 97–108, 2005.

[25] M. Burgess and M. Disney. Understanding scalability in network aggregation with continuous monitoring. Submitted to the 18th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2007) conference., May 2007.

[26] M. Burgess and K. Begnum. Voluntary cooperation in a pervasive computing environment. *Proceedings of the Nineteenth Systems Administration Conference (LISA XIX) (USENIX Association: Berkeley, CA)*, page 143, 2005.

[27] G.M. Amdahl. Validity of a the single processor approach to achieving large scale computer capabilities. In *Proceedings of the AFTPS Spring Joint Computer Conference*, 1967.

[28] A.H Karp and H.P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, 1990.

[29] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[30] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neuge bauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[31] Kyrre Begnum. Manage large networks of virtual machines. In *Proceedings of the Twentieth Systems Administration Conference (LISA XX)*, page 101. USENIX Association: Berkely, CA, 2006.

[32] Canonical Ltd. Ubuntu linux. http://www.ubuntu.com. Last accessed on March 25, 2007.

[33] Paul E. Black. "full binary tree" in Dictionary of algorithms and data structures [online], paul e. black, ed., u.s. national institute of standards and technology. http://www.nist.gov/dads/HTML/fullBinaryTree.html. Accessed on May 17, 2007.

[34] Internet Systems Consortium.

[35] Mark Burgess. *Analytical Network and System Administration — Managing Human-Computer Systems*. J. Wiley & Sons, Chichester, 2004.

[36] M. Burgess and F. Sandnes. A promise theory approach to collaborative power reduction in a pervasive computing environment. In *Springer Lecture Notes in Computer Science*, volume LNCS 4159, pages 615–624, 2006.

[37] M. Burgess. *Analytical Network and System Administration — Managing Human-Computer Systems*. J. Wiley & Sons, Chichester, 2004.

[38] M. Burgess, M. Disney, and R. Stadler. Using patterns in cfengine for robustly scaling network administration. 21st USENIX Large Installation System Administration conference, May 2007.

[39] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

Appendix A

Experimental cfengine configurations

A.1 Serial star

```
control:
        actionsequence  = ( shellcommands tidy )
        domain          = ( cftestnet )
        IfElapsed       = ( 1 )
        TrustKeysFrom   = ( 10.0.0 )

shellcommands:
        any::
                "/bin/echo $(hostname)" background=false
```

A.2 Parallel star

```
 control:
        actionsequence  = ( shellcommands tidy )
        domain          = ( cftestnet )
        IfElapsed       = ( 1 )
        TrustKeysFrom   = ( 10.0.0 )

        node1::
                serve = ( node2:node3:node4:node5:node6:node7:node8:node9:node10:
                    node11:node12:node13:node14:node15:node16:node17:node18:node19:
                    node20 )

classes:
        HasChildren = ( IsDefined(serve) )

shellcommands:
        any::
                "/bin/echo $(hostname)" background=false

        HasChildren::
```

```
                "/usr/local/sbin/cfrun $(serve) 2>&1 > /tmp/echorun.$$" background=
                    true
                "/usr/bin/pgrep cfrun > /dev/null; while [ $? = 0 ]; do pgrep cfrun >
                     /dev/null; done"
                "/bin/cat /tmp/echorun.*"

tidy:
        HasChildren::
                /tmp pattern=echorun.* age=0
```

A.3 Cfrun echo

```
 control:
        actionsequence  = ( shellcommands tidy )
        domain          = ( cftestnet )
        IfElapsed       = ( 1 )
        TrustKeysFrom   = ( 10.0.0 )

        node1::
                serve = ( node2:node3:node4 )

        node2::
                serve = ( node5:node6:node7 )

        node3::
                serve = ( node8:node9:node10 )

        node4::
                serve = ( node11:node12:node13 )

        node5::
                serve = ( node14:node15:node16 )

        node8::
                serve = ( node17:node18:node19 )

        node11::
                serve = ( node20 )


classes:
        HasChildren = ( IsDefined(serve) )

shellcommands:
        any::
                "/bin/echo $(hostname)" background=false

        HasChildren::
                "/usr/local/sbin/cfrun $(serve) 2>&1 > /tmp/echorun.$$" background=
                    true
                "/usr/bin/pgrep cfrun > /dev/null; while [ $? = 0 ]; do pgrep cfrun >
                     /dev/null; done"
                "/bin/cat /tmp/echorun.*"

tidy:
        HasChildren::
                /tmp pattern=echorun.* age=0
```

A.4 Copy chain

```
classes:

leaf    = ( node20 )
root    = ( node1 )

#########################################################

control:

  workfile    = ( "/tmp/chain-pattern" )
  tempfile    = ( "/tmp/chain-temp" )
  actionsequence  = ( shellcommands copy editfiles )
  domain          = ( cftestnet )
  IfElapsed       = ( 0 )
  TrustKeysFrom   = ( 10.0.0 )
  Split           = ( , )

  # For cfexecd
  smtpserver     = ( mail-out.hio.no )
  sysadm         = ( admin@email.address )
  EmailMaxLines  = ( inf )

  microdate      = ( ExecResult("/bin/date --rfc-3339=ns") )

node1::
  serve = ( node2 )
  sleep = ( 19 )

node2::
  serve = ( node3 )
  sleep = ( 18 )

node3::
  serve = ( node4 )
  sleep = ( 17 )

node4::
  serve = ( node5 )
  sleep = ( 16 )

node5::
  serve = ( node6 )
  sleep = ( 15 )

node6::
  serve = ( node7 )
  sleep = ( 14 )

node7::
  serve = ( node8 )
  sleep = ( 13 )

node8::
  serve = ( node9 )
  sleep = ( 12 )

node9::
  serve = ( node10 )
  sleep = ( 11 )

node10::
```

```
  serve = ( node11 )
  sleep = ( 10 )

node11::
  serve = ( node12 )
  sleep = ( 9 )

node12::
  serve = ( node13 )
  sleep = ( 8 )

node13::
  serve = ( node14 )
  sleep = ( 7 )

node14::
  serve = ( node15 )
  sleep = ( 6 )

node15::
  serve = ( node16 )
  sleep = ( 5 )

node16::
  serve = ( node17 )
  sleep = ( 4 )

node17::
  serve = ( node18 )
  sleep = ( 3 )

node18::
  serve = ( node19 )
  sleep = ( 2 )

node19::
  serve = ( node20 )
  sleep = ( 1 )


#############################################################

shellcommands:

!leaf::
        "/bin/sleep $(sleep)"

#############################################################

copy:

!leaf::

  $(workfile)

    dest=$(tempfile)
    server=$(serve)
    force=true

    define=success
    elsedefine=failure

#############################################################

editfiles:
```

```
  success::

   { $(workfile)

   AutoCreate
   EmptyEntireFilePlease
   InsertFile "$(tempfile)"
   AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date) $(microdate)"
   }

  failure::

   { $(workfile)

   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "copy-chain - no response from $(serve)"
   AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date) $(microdate)"
   }

  leaf::

   { $(workfile)

   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date) $(microdate)"
   }

############################################################

tidy:
  /tmp pattern=chain-temp* age=0


alerts:

  success::

   "Chain update succeeded"
   PrintFile("$(workfile)","25")

  failure::

   "No Chain update at $(date)"
```

A.5 Copy tree

```
classes:

leaf = ( node12 node13 node14 node15 node16 node17 node18 node19 )
root = ( node1 )
disabled = ( node5 node6 node7 node8 node20 )

############################################################

control:

  workfile    = ( "/tmp/tree-pattern" )
  global_tempfile    = ( "/tmp/tree-temp" )
```

```
  actionsequence  = ( shellcommands copy editfiles )
  domain          = ( cftestnet )
  IfElapsed       = ( 0 )
  TrustKeysFrom   = ( 10.0.0 )
  Split           = ( , )

  microdate     = ( ExecResult("/bin/date --rfc-3339=ns") )

node1::
  serve1 = ( node2 )
  serve2 = ( node3 )
  sleep = ( 3 )

node2::
  serve1 = ( node4 )
  serve2 = ( node9 )
  sleep = ( 2 )

node3::
  serve1 = ( node10 )
  serve2 = ( node11 )
  sleep = ( 2 )

node4::
  serve1 = ( node12 )
  serve2 = ( node13 )
  sleep = ( 1 )

node9::
  serve1 = ( node14 )
  serve2 = ( node15 )
node10::
  serve1 = ( node16 )
  serve2 = ( node17 )
  sleep = ( 1 )

node11::
  serve1 = ( node18 )
  serve2 = ( node19 )
  sleep = ( 1 )

!leaf::
  tempfile1 = ( "$(global_tempfile).$(serve1)" )
  tempfile2 = ( "$(global_tempfile).$(serve2)" )

############################################################


############################################################

shellcommands:

!leaf.!disabled::
        "/bin/sleep $(sleep)"

############################################################

copy:

!leaf.!disabled::

  $(workfile)

    dest=$(tempfile1)
    server=$(serve1)
```

```
    force=true
    define=success
    elsedefine=failure

  $(workfile)

    dest=$(tempfile2)
    server=$(serve2)
    force=true
    define=success
    elsedefine=failure

############################################################

editfiles:

  !leaf.!failure.!disabled::

   { $(workfile)

   AutoCreate
   EmptyEntireFilePlease
   InsertFile "$(tempfile1)"
   InsertFile "$(tempfile2)"
   AppendIfNoSuchLine "copy-tree $(host)=$(value_loadavg) at $(date) $(microdate)"
   }

  failure.!disabled::

   { $(workfile)

   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "copy-tree - no response from $(serve)"
   AppendIfNoSuchLine "copy-tree $(host)=$(value_loadavg) at $(date) $(microdate)"
   }

  leaf.!disabled::

   { $(workfile)

   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "copy-tree $(host)=$(value_loadavg) at $(date) $(microdate)"
   }

############################################################


tidy:
  /tmp pattern=chain-* age=0
  /tmp pattern=tree-temp* age=0

alerts:

  success::

   "Tree update succeeded"
   PrintFile("$(workfile)","25")

  failure::

   "No tree update at $(date)"
```

Appendix B

Speedup calculations

B.1 Serial star compared to parallel star

B.1.1 Observed speedup

$$\begin{aligned} \Psi &= \frac{T(1)}{T(p)} \\ \Psi &= \frac{T_{serialstar}}{T_{parallelstar}} \\ \Psi &= \frac{10.71}{2.26} \\ \Psi &= 4.74 \end{aligned}$$

B.1.2 Karp-Flatt metric

$$\begin{aligned} e &= \frac{\frac{1}{\Psi} - \frac{1}{p}}{1 - \frac{1}{p}} \\ e &= \frac{\frac{1}{4.74} - \frac{1}{20}}{1 - \frac{1}{20}} \\ e &= 0.17 \end{aligned}$$

B.2 Serial star compared to echo

B.2.1 Observed speedup

$$\begin{aligned}\Psi &= \frac{T(1)}{T(p)} \\ \Psi &= \frac{T_{serialstar}}{T_{echo}} \\ \Psi &= \frac{10.71}{5.68} \\ \Psi &= 1.89\end{aligned}$$

B.2.2 Karp-Flatt metric

$$\begin{aligned}e &= \frac{\frac{1}{\Psi} - \frac{1}{p}}{1 - \frac{1}{p}} \\ e &= \frac{\frac{1}{1.89} - \frac{1}{20}}{1 - \frac{1}{20}} \\ e &= 0.50\end{aligned}$$

Appendix C

Programs written and used for testing and analysis

C.1 Copy chain simulation program

```
/*****************************************/
/*                                       */
/* File: period.c                        */
/*                                       */
/* Created: Sat Apr 28 11:09:23 2007     */
/*                                       */
/* Author: Mark Burgess                  */
/*                                       */
/* Revision: $Id$                        */
/*                                       */
/* Description:                          */
/*                                       */
/*****************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define N 20

// gcc -o period period.c -lm

/* With a splay time, we cannot get a profile from
 time zero for all hosts -- only from the closest
```

```
 possible ordered set. So as soon as we introduce
 waiting, there will be a spread of values.

 In a periodic scheme with zero wait, each agent could
 measure the value at the synchronized time but would
 have to wait to propagate it.. so we can talk to
 "time to aggregation" we can only minimize the TTA */

main()

{ int i,j;
 double t[N+1],tau[N+1],dt;
 int P = 60, ts = 1;
 double sum = 0, commdelay;
 int k,ex;
 double delta = -1;
 long seed;
 double noise = 0.19;

 double d[13][75];
 double av[13],var[13];
 int branch[13];

// WIth the random seed we now have a new timescale t_fluc/P
   seed = ((long)time(NULL)*17);
   srand48(seed);

for (ex = 0; ex < 75; ex++)
   {
   //printf("EXPT %d\n",ex);

   for(ts = 0; ts <= 12; ts++)
      {
      d[ts][ex] = 0;
      av[ts] = 0;
      var[ts] = 0;
      branch[ts] = 0;
      }
   }

for (ex = 0; ex < 75; ex++)
   {
   // Starting experiment trial ex - average over 75 trials
   for(ts = 0; ts <= 12; ts++)
      {
```

```
        sum = 0;

        for (i = 1; i <= N; i++)
            {
            double random = noise*drand48();

            // MEasurement/sample time
            t[i] = (i-1) * ts + random;
            }

        for (i = 2; i <= N; i++)
            {
            // Desimal valued remainder:
            double dt1 = t[i]-(((int)t[i])/P)*P;

            // Need to do this for accuracy
            double dt2 = t[i-1]-(((int)t[i-1])/P)*P;
            delta = -1;
            dt = dt1 - dt2;

            if (dt <= 0)
                {
                sum += P;  // Must wait a cycle if the order is
                    wrong
                }

            delta = t[i]-t[i-1];

            // Just a guess to make some error bars
            // Note they grow with ts due to waiting delay
            commdelay = fabs(dt*noise/0.19*drand48()*4.5);
            sum += (delta + commdelay);

            }

        d[ts][ex] = sum;

// Note the case of ts=0 is much more sensitive
// to noise as there is no buffer margin to absorb
// it...hence error bars bigger if noise < ts

        printf("Result: %d %f = %f\n",ts,sum,d[ts][ex]);
        printf("Predic: %d %d\n",ts,(N-1)*ts + (((N-1)*ts)/P)*P)
            ;
        printf("-----------------------------------\n");
```

```
      }
   }


for (ts = 0; ts <= 12; ts++)
   {
   double test = 0;
   av[ts] = 0.0;
   var[ts] = 0.0;

   for (ex = 0; ex < 75; ex++)
      {
      av[ts] += d[ts][ex]/75.0;
      }

   for (ex = 0; ex < 75; ex++)
      {
      var[ts] += (d[ts][ex] - av[ts])*(d[ts][ex] - av[ts])
         /75.0;
      }

   printf("%d %f %f\n",ts,av[ts]/60.0,sqrt(var[ts])/60.0);
   }

// duration of offsets plus (number of times it
// wraps around -1)*P
}
```

C.2 Copy chain data analysis

```
#!/usr/bin/perl
# Name:    chainparse.pl
# Author: Matt Disney
# Description: A script to parse chain tests
#   output from cfengine, as specified in my
#   thesis. The output of this script can be
#   read by gnuplot. Separate sections can be
#   referenced using gnuplot's index parameter.

use Getopt::Std;
use Date::Manip qw(ParseDate UnixDate);
use Time::Local;
use List::Util qw(sum);
```

```
getopts ( 'hf:', \%opt ) or usage();
if ( $opt{h} or !$opt{f} ) { usage() };

sub usage()
{

print STDOUT << "EOF";
Usage: $0 [-h] -f file
EOF

exit;

}

open (FILELIST, $opt{f}) || die;

my $j = 0;
my $accum = 0;
my $stdev = 0;
my @diffary;
my %histo;
my %phases;

print "\#ID Diff\n";
while (<FILELIST>)
{
  my $i = 1;
  my $node;
  my $cruft;
  my $node1 = 0;
  my $node20 = 0;

  open (DATAFILE, $_) || die;

  while (<DATAFILE>)
  {
  # Format example:
  # copy-chain node20=35 at Tue Apr 24 13:50:09 2007
     2007-04-24 13:50:09.952333000+02:00
    @line = split(' ', $_);
    if ( $line[3] =~ /succeeded/ ) { next; }

    ($node,$cruft) = split ('=',$line[1]);
    $node =~ s/node//;
```

```
    @middle = split('_', $line[3]);
    ($hours,$min,$secs) = split(':',$middle[4]);
    if ( $hours =~ /2007/ )
    {
      ($hours,$min,$secs) = split(':',$middle[3]);
    }

    $date = ParseDate($line[4]);

    if (!$date) {
      print "Date didn't work\n";
    } else {
      ($year,$month,$day) = UnixDate($date, "%Y", "%m", "%d");
    }

    my $epoch_seconds = timelocal(0,$min,$hours,$day,$month,
       $year);
    my $epoch_minutes = $epoch_seconds / 60;

    if ( $node == 20 )
    {
      $node20 = $epoch_minutes;
    }
    elsif ( $node == 1 )
    {
      $node1 = $epoch_minutes - $node20;
    }

    push(@{$phases{$node}{'values'}}, $epoch_minutes -
       $node20) || die "Couldn't push element onto array: $!";

    $i++;

  }

  close (DATAFILE);
#  $diff = $node1 - $node20;
#  $diff += 60 if ($diff < 0);
#  print "$j $node1 $node20 $diff\n";
  print "$j $node1\n";

  $j++;
  $accum = $accum + $node1;
  push(@diffary, $node1);
```

```
}

close (FH);

print "\n\n\n";

print "\#Histogram\n";
$histo{$_}++ for @diffary;
foreach $key (sort(keys %histo))
{
  print "$key $histo{$key}\n";
}
print "\n\n\n";


print "\#Average TTA at each node\n";
print "\#Distance_from_leaf avg_tta stdev\n";
for $n (sort {$b <=> $a}(keys %phases))
{
  $phases{$n}{'avg'} = sum(@{$phases{$n}{'values'}})/$j;

  foreach $m (@{$phases{$n}{'values'}})
  {
    $phases{$n}{'stdev'} = $phases{$n}{'stdev'} + ($m -
       $phases{$n}{'avg'}) ** 2;
  }
  $phases{$n}{'stdev'} = sqrt( $phases{$n}{'stdev'} / $j );

  print 20-$n." $phases{$n}{'avg'} $phases{$n}{'stdev'}\n";
}

print "\n\n\n";

print "\#Other stats\n";
print "Average: $phases{'1'}{'avg'}\n";
print "Standard Deviation: $phases{'1'}{'stdev'}\n";
```

C.3 Copy tree data analysis

```
#!/usr/bin/perl
# Name:    treeparse.pl
# Author: Matt Disney
# Description: A script to parse tree tests
#   output from cfengine, as specified in my
```

```
#    thesis. The output of this script can be
#    read by gnuplot. Separate sections can be
#    referenced using gnuplot's index parameter.
#
#    The principal differences in this script
#    and the chainparse script is that this one
#    uses the longest value obtained and it also
#    does not provide TTPA data.

use Getopt::Std;
use Date::Manip qw(ParseDate UnixDate);
use Time::Local;


getopts ( 'hf:', \%opt ) or usage();
if ( $opt{h} or !$opt{f} ) { usage() };

sub usage()
{

print STDOUT << "EOF";
Usage: $0 [-h] -f file
EOF

exit;

}

open (FILELIST, $opt{f}) || die;

my $j = 0;
my $accum = 0;
my $stdev = 0;
my @diffary;
my %histo;

print "\#ID Diff\n";
while (<FILELIST>)
{
  my $i = 1;
  my $node1 = 0;
  my $oldest = 0;

  open (DATAFILE, $_) || die;
```

```
while (<DATAFILE>)
{
# Format example:
# copy-chain node20=35 at Tue Apr 24 13:50:09 2007
  2007-04-24 13:50:09.952333000+02:00
  @line = split(' ', $_);
  if ( $line[3] =~ /succeeded/ ) { next; }

  @middle = split('_', $line[3]);
  ($hours,$min,$secs) = split(':',$middle[4]);
  if ( $hours =~ /2007/ )
  {
    ($hours,$min,$secs) = split(':',$middle[3]);
  }

  $date = ParseDate($line[4]);

  if (!$date) {
    print "Date didn't work on $line[4]\n";
  } else {
    ($year,$month,$day) = UnixDate($date, "%Y", "%m", "%d");
  }

  my $epoch_seconds = timelocal(0,$min,$hours,$day,$month,
    $year);
  my $epoch_minutes = $epoch_seconds / 60;

  if ( $line[1] !~ /node1=/ )
  {
    if ( $oldest == 0 )
    {
      # First value...
      $oldest = $epoch_minutes;
    }
    elsif ( $epoch_minutes < $oldest )
    {
      # If the current line has a value older (less) than
      # the existing $older, replace $older with that value.
      $oldest = $epoch_minutes;
    }
  }
  else
  {
    $node1 = $epoch_minutes - $oldest;
  }
```

```
    $i++;

  }

  close (DATAFILE);
  print "$j $node1\n";

  $j++;
  $accum = $accum + $node1;
  push(@diffary, $node1);
}

close (FH);

print "\n\n\n";

$average = $accum/$j;

for $n (@diffary)
{
  $stdev = $stdev + ($n - $average) ** 2;
}
$stdev = sqrt($stdev / $#diffary);

print "\#Histogram\n";
$histo{$_}++ for @diffary;
foreach $key (sort(keys %histo))
{
  print "$key $histo{$key}\n";
}
print "\n\n\n";

print "\#Other stats\n";
print "Average: $average\n";
print "Standard Deviation: $stdev\n";
```