Network Patterns in Cfengine and Scalable Data Aggregation

Mark Burgess and Matthew Disney – Oslo University College
Rolf Stadler – KTH Royal Institute of Technology, Stockholm

ABSTRACT

Network patterns are based on generic algorithms that execute on tree-based overlays. A set of such patterns has been developed at KTH to support distributed monitoring in networks with non-trivial topologies. We consider the use of this approach in logical peer networks in cfengine as a way of scaling aggregation of data to large organizations. Use of 'deep' network structures can lead to temporal anomalies. We show how to minimize temporal fragmentation during data aggregation by using time offsets and what effect these choices might have on power consumption. We offer proof of concept for this technology to initiate either multicast or inverse multicast pulses through sensor networks.

Introduction

In this paper we consider an approach for scaling data dissemination (e.g., for configuration management) or alternatively for scaling data aggregation (e.g., for monitoring or archiving) by implementing Network Patterns on top of cfengine's pull-based copy methods. This follows up preliminary work on scaling in [1, 2] and Voluntary Cooperation [3] and is inspired by work on the Generic Aggregation Protocol (GAP) described in [4, 5, 6].

Consider the sharing of load in a multicast process by handing off parts of a task to decentralized processing. For example, in a distributed backup scheme, one could imagine assigning responsibilities such that local nodes collected and compressed their own data before passing them from the leaves of a tree to their parent node; the parent would then aggregate data from all of its children and adds its own data, and so on up the tree to a final repository. By introducing several tree levels one reduces the total computational burden on the final host. Such a strategy could be useful in either a fixed infrastructure network (where nodes have limited computational power) and especially in battery powered processors such as wireless ad hoc devices, sensor networks, and so on.

Network Patterns are based on generic, distributed algorithms that execute on spanning trees, designed to collate information from a topologically constrained network, such as a fixed routing infrastructure or ad-hoc substrate. They employ the basic structures used in routing and switching, like spanning trees, and can adapt to node or link failures [4, 5, 6]. We shall consider only aggregation algorithms here, where aggregates of local variables across a domain of local devices are computed using functions, such as sum, max, or average.

The overlay networks are usually created under some basic physical constraints such as geography, physical network design, allowed access, or even by wireless power limitations in an ad hoc network. In other words, certain branches and levels in the tree could be forced into the final topology by physical circumstances, hence one could not merely choose the simplest star topology for the task, even if it were not an unacceptable burden on the single bottleneck. However, we can also ask whether it makes sense to build such structures even where there are no constraints, such as local area networks with underlying star topology. There are valid resource sharing reasons for doing this in system administration, especially where resources are limited.

Network patterns allow a kind of load balancing, but they are different from the kind of service balancer which one might use on a web server: a traditional load-sharing dispatcher acts like a switch, taking a single input stream and offloading it to a separate queue: in a network pattern data are sent to all branches, like a ''smart'' multi-port repeater or amplifier/aggregator.

Inter-Domain Management and Voluntary Cooperation

A subject that is increasingly discussed in today's world of cooperative outsourcing is the issue of *inter-domain management*. In the extreme case, each node in a network is in its own administrative domain (this is approximately true for border routers, for example, as well as hand-held devices). Inter-domain management involves many issues that are often ignored in discussions of system administration. For example, we do not typically have privileged access to all of the devices we communicate with. The concept of Voluntary Cooperation was introduced to discuss ''minimal trust'' interactions with autonomous domains [3].

Even a wireless ad hoc network of personal electronic devices (or a military network deployed in the

field) could be formed from many devices with different privileges and privacy policies. Traditional models of centralized control do not begin to address these issues.

Monitoring (data collection) from a network of sensors (either in a fixed infrastructure net or in a wireless environment) is an application that has received a lot of attention. This is because ‘‘network management’’ has traditionally been about watching network traffic data. Even today as vendors advocate the virtues of autonomic computing, network managers still want to watch the automation in progress. Thus the problem of distributed aggregation with unclear domain relationships is still at the heart of network management.

Cfengine is a management system that represents state of the art research on integrating monitoring and reactive (‘‘autonomic’’) management of computers. Integrating network patterns into cfengine would allow distributed monitoring and management of a manifestly autonomic system with any chosen degree of centralization or decentralization. Cfengine is designed to be able to work in mobile, partially connected environments. It is an ideal testbed for exploring the usefulness of patterns in host based system administration. Moreover, eventually it is expected that cfengine will be able to manage routers and switches for which patterns were originally envisaged.

Network Patterns are not generic routing or switching structures, although they share similarities. They are designed to execute any computation whose data can be represented on the underlying graph. This typically involves aggregation, dissemination, maximization or minimization etc. Here we use them only for the simplest aggregation of data from every node in a network to an arbitrary but central place. They are therefore used to initiate either multicast or inverse multicast pulses through sensor networks.

Any collection of ‘‘sensor devices’’ that can run on a GNU/Linux platform could use cfengine in the way we demonstrate here, and this accounts for an ever increasing number of devices available today. One application is for collecting and correlating data from around a network from cfengine’s own sensor component `cfenvd`. Cfengine’s investment in methods of *voluntary cooperation* means that one need not give away privileges in order to implement patterns, hence risking or sacrificing security. This makes monitoring of large an fragmented organizations an easier process to swallow for security officers (the alternative being to open firewalls to unspecified network pushes). Increasingly companies are outsourcing their systems into different formal domains with their own policies and barriers. The fact that one can make patterns work with voluntary cooperation is therefore itself a valuable proof of concept.

A natural application for this kind of process is for monitoring grid systems. These are systems that are often geographically distributed and already form part of some organized structure. Patterns at the level of host based monitoring would allow grid administrators to view the performance characteristics of the component systems or even aggregate results from them with controllable accuracy.

There are various other applications for data aggregation to a point. Another one is to perform a distributed backup, collecting and compressing data as they propagate up the tree. This would offload the

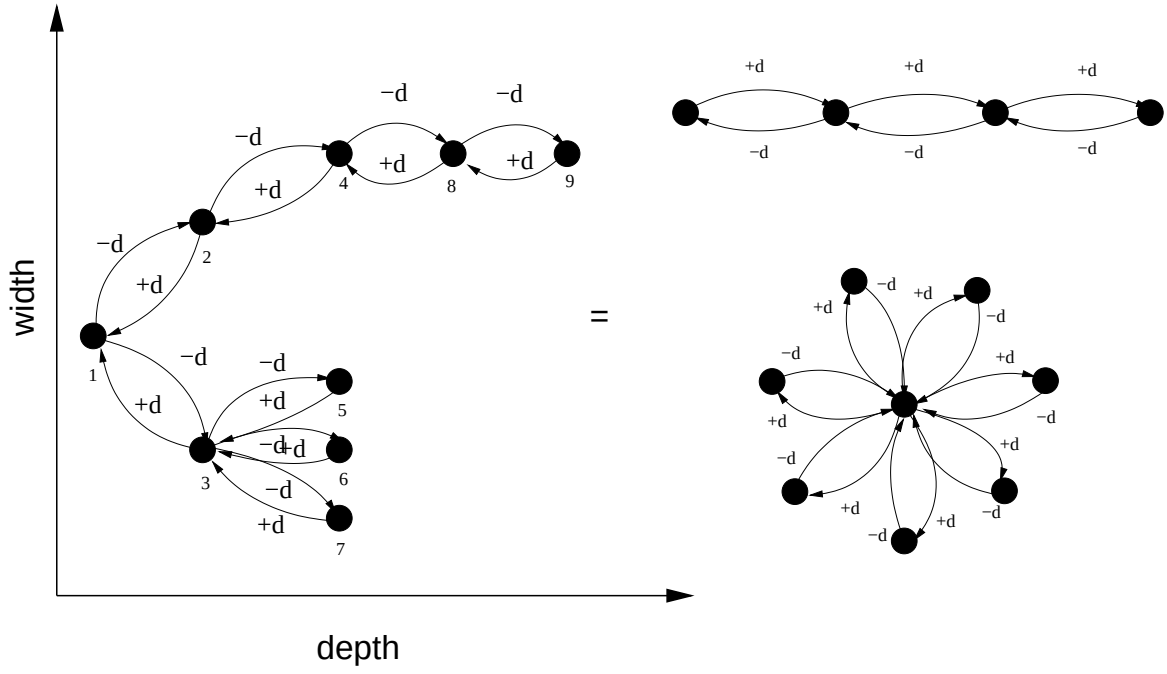

Figure 1: Depth and width in network patterns formed from promises.

burden of performing the compression, and data could be encrypted with local keys before compression. We shall not elaborate on these applications here, but simply present these tests as proof of the concept.

Some Patterns

The patterns discussed here are dissemination and aggregation algorithms that bridge the worlds of centralized monitoring and fully distributed monitoring. They are built from ‘‘component’’ pieces that represent the extreme cases of any network structure: *chain* (for maximum depth) and the *star* topology (for maximum width), see Figure 1.

Trees are structures that bridge these two extremes. We can characterize patterns by their depth and breadth. Note that a chain is also geometrically a half-ring, so it gives us a basic model for ring-topologies also.

Here we consider only two patterns: Echo and GAP and consider how these can be implemented in cfengine using existing context awareness within the system.

Echo

The simplest example of a network pattern is the *echo* pattern [7, 8]. During its execution, echo creates

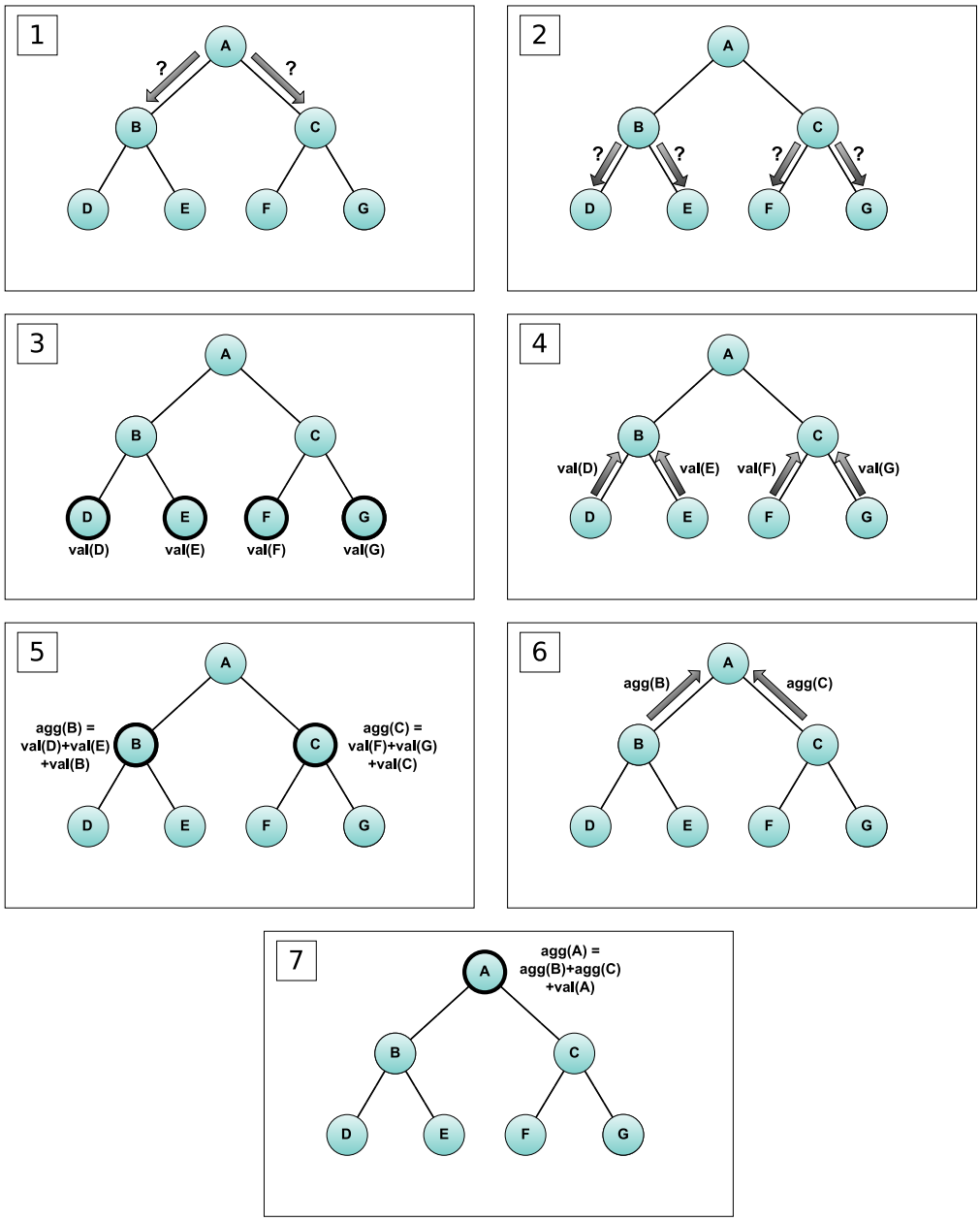

Figure 2: The expansion and contraction steps in the echo pattern. The pattern is a sequence of ‘‘pulls’’ initiated by ‘‘pushed’’ signals.

a spanning tree topology, with root of the tree chosen a by an administrator. The pattern has two phases of communication: *expansion* and *contraction* (see Figure 3). 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 recursively, until the root node is reached, which 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. The echo pattern therefore forms a wave, spreading out from the root to the edge of the network and back, collecting data as it progresses. Echo is intrinsically a ''push'' protocol, and is easily understood as a recursive descent parser.

GAP

Similar to Echo, the Generic Aggregation Protocol (GAP), creates a spanning tree along which communication and computation takes place. Unlike echo however, data in GAP are passed from the leaves of the tree towards the root, whenever the local variable in one of the nodes changes. Updates to monitored aggregates can thus be initiated by any node, not only by the root node. Thus GAP, one initialized, responds to local events rather than initiating measurements from a central observer.

GAP is an asynchronous distributed protocol that builds and maintains a Breath First Search (BFS) or a spanning tree over which aggregates are computed incrementally and continuously. The tree is maintained in a similar way to the algorithm underlying the 802.1d Spanning Tree Protocol (STP). In GAP, each node maintains a table of its peers and especially its nearest neighbours along with an estimation of the nodes' aggregate values. GAP is event driven, in the sense that each update from a leaf node triggers a cascade of events through the tree branches, updating the local aggregates as it goes. Update events can be triggered by changes in topology, loss of a node, a timer, etc.

The advantage of GAP over echo is that there is no ''push phase'' required to initiate a reading of the values from the network. As each change occurs in the network, new values can be percolated back to the centralized root node initiating an update only in those tree nodes that are in the path to root. This avoids the need for much unnecessary traffic and computation during updates.

The Topology Manager

A key feature of the patterns above is the algorithm by which the topology of the spanning tree is decided. The GAP algorithm incorporates the topology adjustment mechanism into the GAP aggregation algorithm, by using nearest neighbour communications, hence combining these features into a robust protocol. However, they can be separated also. The GoCast algorithm finds such a spanning tree, for example. At this stage of the work we shall not attempt to encode automated topology management, as this requires additional subsystems. Rather we consider how patterns can be used at the logical level for distributed load balancing, using existing mechanisms within cfengine. We note however that cfengine has implemented peer neighbour management functions for some time in the form of the

```
SelectPartitionNeighbours
SelectPartitionLeader
```

functions. These functions take a flat list of all known hosts and partition this list into clusters of a specified size. Each cluster is assigned an identified leader which can be used to single out a root or responsible node for each group, and in this way any host can autonomously be made aware of its nearest neighbour topology based only on the shared information of the flat list. These functions, or functions like them could in principle be used to provide an implementation of GAP topology management in future, for automatically adaptation for fault tolerance. However, we shall not pursue the details of the topology here, since it turns out that the implementation of patterns throws up a number of issues that are more fundamental.

The problem of building soft-overlays for computational load sharing is slightly different to the problem of finding a spanning tree through a physically redundant topology however. In principle, any kind of overlay could be built in software, but physical constraints can limit the potential optimizations. What we find interesting in a cfengine environment is that we must deal with a combination of these issues. If cfengine is used in a simple star network, any kind of overlay can be built. However, if it is used for inter-domain management, or between zones with different administrative regimes, then these amount to essentially physical constraints.

Cfengine Principles and Patterns

By implementing network patterns in cfengine we hope to achieve two things: i) an efficient way of aggregating data for centralized analysis and decision-making, and ii) open for the possible load sharing optimizations that are possible with patterns. An obvious goal for centralized decision-making would be to use this to build an ''autonomic nervous system'' from cfengine's autonomous agents so that centralized monitoring and decision-making can be added to its local stimulus-response approach to management. Although many configuration management schemes boast ''centralization'', this can often be seen as a weakness, as it is a clear limitation on scalability, and such systems usually only disseminate data from a centralized source: we

are advocating stimulus-response in a distributed system, something like a central nervous system. While individual machines work autonomously, we collect, process and return data to the nodes on a continuous basis.

The desired model is not without its own challenges however: cfengine maintains strong principles of autonomy that are largely responsible for its record of security and reliability. The challenge is to implement aggregation/dissemination patterns without sacrificing those strong principles.

A cfengine host is, by default, a completely autonomous entity with no obligations towards other agents in a physical network. Every node is therefore individual and is not part of a pattern a priori. Leaf nodes cannot initiate a push of new data in response to events, because the parent node does not accept data from any outside source, unless it explicitly pulls the data itself. To use patterns as a form of inter-peer collaboration, we must encode them as policy rules that are compatible with cfengine's pull-only principle of communication. There are several questions to be answered about this:

- Is the underlying physical network topology important in building a logical load sharing topology?
- How will the topology be decided?
- How will the topology respond to the failure of nodes?

We shall not be able answer all of these questions, but we present the basic approach to building GAP-like patterns using cfengine's internal mechanisms, and provide tools for readers to experiment on their own.

Periodic Execution

Cfengine is normally used for regular (periodically) scheduled maintenance sweeps, yet the traditional idea of a network probe is to ask a question and get back the answer on demand (as with probes like ping and traceroute). The Echo pattern is a ''push-me pull-you'' strategy for connecting to all elements in a managed network and transmitting or collecting data: a kind of broadcast ping. The principal advantage of this kind of approach is that the timing of the distributed process is event driven. It does not require an elaborate clock synchronization and timed firing to coordinate the distributed execution, since the interactions are themselves synchronous. However, it is inherently fragile as it involves the privilege to push and collect through a chain of dependencies. If the top node loses communications with its children, none of the network operations will be executed. A better approach would be allow all nodes in the network to operate *autonomously* and have them cooperate when they are able.

A typical cfengine approach to the problem to execute the distributed agents periodically (with period *P*, anything from a few minutes to an hour). Neighbouring cfagents could download from their children servers to aggregate the results, but now the timing plays a role. Since there is no push possibility to coordinate the operations, the process is fragile to time coordination [9]. There are two issues: i) clock synchronization and ii) clock schedule for ensuring the data are updated in time before the data values are pulled downstream. If either of these requirements is not met, data that are pulled will be out of date and will not give an accurate representation of the true values.

So what happens if the nodes are not properly synchronized? Since cfengine operates autonomously and its copying is fault tolerant, a missed update could simply be captured at a later time. This might not seem like a problem, unless one begins to measure the spread of times in the ''current'' data. The situation is somewhat analogous to asking post office branches to report to their head office on how many customers they have each day using their own postal delivery. At any given regular delivery, the letters that arrive at the central office have a variety of postmarks. Some of them are delivered on the same day, and some of them take perhaps a week to deliver. Thus updates might arrive eventually, but how shall we understand the results that arrive? Do we group letters by their postmarks and only combine results that were originated on the same date? Or do we ignore the post-marks and combine data that were received on the same date? In the first case, we might have to wait a long time for the data, but we are certain of what we are seeing. In the latter case, the result is available quickly but the meaning of the data is in question.

Each hop in a chain of delivery adds new possibility for delay. If the mail does not arrive before one post office sends its own delivery, the incoming mail will have to wait a whole day for the next delivery (a whole scheduling period *P*). A single failure could not bring down the entire system, but it could skew the impression received at the central monitoring station. It is therefore advantageous, if not imperative, to develop patterns that do not have this strong dependency feature.

To avoid the dependency and delay problem, we based our work on the assumption of time synchronization. As we shall see, even this is susceptible to noise. Apart from a proof-of-concept implementation, we did not pursue the echo pattern for this reason (in spite of its ready comprehensibility) and instead were inspired by the Generic Aggregation Protocol (GAP) approach. For GAP we shall not attempt a complete implementation, but rather emulate its operation as a first step to making progress. GAP includes an algorithm for automatic renegotiation of the structure. This has several implications which require some soul searching when implementing in cfengine. Further research by KTH based on the cfengine experience can also help to adapt the GAP algorithm for pull-based scenarios.

Promise Agreements and Voluntary Cooperation

The notion of promises was introduced as a way of modeling networks of agents cooperating in an ad hoc fashion. Cfengine can be viewed as a reference implementation of the abstract promise-theoretic scenario. Promise theory was introduced precisely as a modeling framework that could describe cfengine, where others could not.

Promise theory is a high level graphical description of constrained behaviour in which ensembles of agents document the behaviours they promise to exhibit. Agents in promise theory are truly autonomous, i.e., they decide their own behaviour, cannot be forced into behaviour externally but can voluntarily cooperate with one another [10]. A promise is a directed edge

$$A_1 \xrightarrow{b} A_2 \qquad (1)$$

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 behaviour b (written $A_1 \xrightarrow{+b} A_2$, and promises to accept something or make use of another's promise of behaviour 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 behaviour or receive the service.

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 fits nicely with our view of collection of distributed information for measurement purposes.

We shall consider the following promise designations: $+d$ server provides data, $-d$ client receives/uses data, $+a$ branch node aggregates data, $+t$ server provides time/clock, and $-t$ client uses time/clock. Although we speak mainly of network nodes below, it will be understood that each node is modeled as an "agent" in promise theory parlance.

Promise theory allows us to see the relationship between network patterns and policy for autonomous agents. Each arrow in the promise graph attaches to a rule in the policy to either grant access to data or to fetch available data. In this way we can build dissemination processes over graphs using node location data or context sensitivity information.

A common mistake is to think of promises as communication transactions, rather than as abstract behavioural specifiers. A promise says nothing necessarily about the details of what is communicated between agents at a given moment, only that it intends to behave within the confines of its promise. However, one usually assumes that a promise means a best effort to comply with the announced constraints and that no promise means that nothing will happen. A reliable binding between two hosts requires both a promise to serve and a promise to use the promised service.

$$A_1 \xrightarrow{+b} A_2, A_2 \xrightarrow{-b} A_1 \qquad (2)$$

The Echo and GAP patterns are particularly well suited to implementation using voluntary cooperation, because the propagation of data along tree-like pathways does not depend strongly on whether data are pushed or pulled. The main challenge in a voluntary cooperation scenario is for an agent in the graph to know when its child has data waiting. When data are pushed, we essentially send a signal "do it now", and no other time synchronization is required. This becomes more complicated in a pull regime however. Regular polling of a host's servers is an obvious answer to the question of when to download data. If clocks in the network are synchronized correctly we can even ask for data to be copied only if they have been updated since the last copy. However, this requires the extra overhead of time synchronization and it still does not guarantee that data will be ready for collection at a given moment.

This issue becomes most pronounced when one attempts to request regular pollings of data and the time for data to propagate through the network approaches the time interval for the polling. We have discussed this issue in a separate paper [9], but some of the effects can be seen in Figures 5 and 6.

Using Context Awareness for Making Network Patterns

Cfengine agents are aware of location and context through their evaluation of the environment into a set of classes. These classes are then used as Boolean flags to attach policies conditionally to scenarios. This context sensitivity enables a set of distributed promises to be coded into a single document.

A method in cfengine is like a pair of promises, provided it is voluntarily declared by both parties. An MD5 hash is used to verify that the methods are in fact the same.

The first (service) promise identifies the function being performed, as the body b(). The class expression A_1:: says that this rule applies to the context of agent A_1, which is the service provider (server host). The server=A_1 attribute matches the context expression and, from this, the agent deduces that it is the provider.

```
methods:
  A_1::
     b(params) server=A_1
```

$$A_1 \xrightarrow{+b} A_2, A_2 \xrightarrow{-b} A_1 \qquad (3)$$

The second part applied to agent A_2 and has the form:

```
methods:
  A_2::
     b(params) server=A_1
```

This identifies the function being performed and signals to A_2 that it will use the results performed by server A_1. Since this is not its own identity, this implies that the result is a use-promise.

If we assume that two agents use an identical configuration specification, then a remote procedure call binding can then be written methods:

```
A_1|A_2::
   b(params) server=A_1
```

The same text either in both contexts and a single link in a logical overlay network is added.

Echo

Cfengine's modus operandi is to "pull" data rather than to push. This is a natural side effect of its philosophy of voluntary cooperation. Push is disallowed, with one exception: we are allowed to send a single invitation to each peer to execute its existing policy using the command cfrun. The host is free to disregard this message, but for cooperation purposes it is normal for the peer to respond to such an invitation by executing its policy compliance-checking agent. We can use this mechanism to start an echo avalanche, with a pre-arranged pattern.

The start host executes cfrun to a number of "children". Each child then voluntarily executes cfengine, which in turn encapsulates the execution of cfrun directed at another set of children, which encapsulates cfrun to another set, and so on. Since cfengine aggregates the data from encapsulated processes automatically, it automatically aggregates the entire tree in a synchronized manner. This is the simplest implementation of echo which uses context sensitivity to identify parent-child relationships.

Both serial and parallel star collation can be performed in cfengine echo. The difference is that 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 is statically defined.

The use-promises are encoded as follows as in Figure 3.

```
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:
        "/bin/echo $(hostname)"
 HasChildren::
        "/usr/local/sbin/cfrun $(serve) \
                  2>&1 > /tmp/echorun.$$"
              background=true # parallelize
        "/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
```

Figure 3: The only kind of push structure that can be implemented in cfengine is the echo pattern, using nested cfrun commands. These must be authorized in advance.

Promise Chains (Forwarding)

Two implementations of chains are shown in Figures 7 and 8 for readers to try. 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. This is a basic problem with all structures of significant depth.

Using a chain length of 20 nodes, we consider the periodic execution of cfagent each minute and measured the time to propagate data from one end of the chain to another, in repeated trials. 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 used

the cfengine *copy* action to copy a partially aggregated file from its child. Then the node used the cfengine *editfiles* action to append its own load data to the bottom of the file.

The results of the experiments are shown in Figure 4. We shall report on a detailed explanation elsewhere.

The graphs can be understood roughly as follows. The solid line shows a prediction based on the assumption of regular deterministic behaviour. For zero time-delay between receiving and sending in the chain the age of the data is about ten periods. This is what one would expect by random chance: about half the nodes are correctly ordered on average. As the delay is increased to one minute (greater than noise) the noise becomes irrelevant and an optimal number of nodes is correctly ordered for direct transmission. This gives the fastest result. Then as the delay increases, the time increases in steps. If the wait time times the length of the chain is greater than a period, then the nodes on the period boundary will be out of step and will have to wait a whole period to update, hence the jumps in the graph. What is interesting is that the effect of noise is to improve this handicap. There is no room here for a full discussion of this phenomenon, but the result is essential to understand for monitoring.

Promise Trees (Aggregation)

The chain is an unlikely topology in a real distributed system. 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. We have repeated our experiments for binary trees and the results are show in Figure 5. The cfengine configuration patterns for these tests are shown in the aggregation examples, Figures 8 and 9.

The data from the tree results are not directly comparable to those of the chain, for several reasons. A number of scales change when performing local aggregation and these changes interfere with the time-scales of system noise. Understanding the tree results is therefore rather more complicated than understanding the chains. The parallel arms of the trees interfere sometimes destructively for parallelized copy and sometimes constructively in serialized copy. Thus our graph seems to reveal a relative stability compared to the chain. This is slightly misleading however. The same basic behaviour is common to both cases; however, the tree is able to delay the onset of temporal instability from chain depth (see [9] for more explanation).

Suppose we put aside the restrictions on topology due to local environment, e.g., the finite range of a wireless network, and ask whether there are reasons for building a tree with a particular number of neighbours (node degree) for aggregation or dissemination. This question should be answered differently depending on who initiates a transmission through the network, how often and at what relative times. In the cfengine model of maintenance in which data are

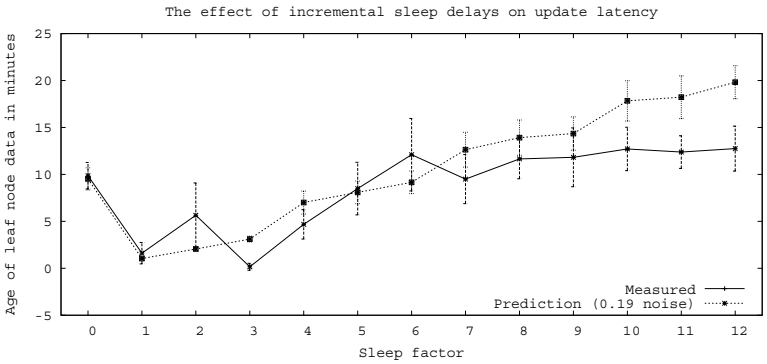

Figure 4: Predicted versus experimental results for the chain propagation. The presence of noise or time-variations actually improves performance compared to a deterministic prediction.

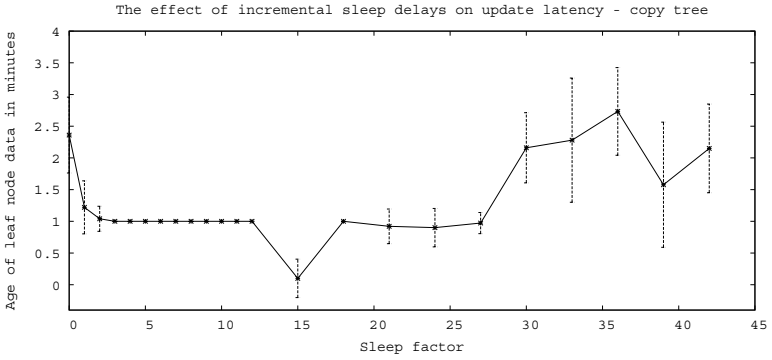

Figure 5: Experimental results for binary tree propagation.

sampled at regular intervals, the behaviour of an aggregation process is something of a cross between the GAP protocol and a Gossip approach [11]. The periodic checking of cfengine promises adds a level of complexity to the data quality of the final result. However, the synchronization of the binary tree is much less sensitive to the size of small offsets than for the chain so it would seem to be advantageous to choice a tree over a chain.

Clearly then the tree is more efficient in terms of time and the decreased network depth gives more freedom in choosing the synchronization parameters. Increasing the node degree (number of children) in the tree increases the processing burden on the aggregator in order to maintain the same accuracy of service level however. A question therefore presents itself: is there an optimal node degree for distributed monitoring?

Scalability

Scalability is about how well a system continues to perform in all its parts as it grows. The burden of size can have a variety of negative effects on a system.

For scalability, we seek to minimize the time to delivery from the leaves of the data structures to the roots (i.e., obtain the lowest value on the vertical axis), while maintaining meaningful data by minimizing temporal fragmentation (partially represented by the error bars). Thus we would like to be as close the lower left of the figures as possible. Our results tell us something about how to achieve this by adjusting overlay topology.

The two structural poles for the network patterns were illustrated in Figure 1: the star pattern for maximum parallelization and centralization (hence maximum burden per root node) and the chain for maximum off-loading and decentralization (hence maximum temporal fragmentation). With centralization, the fraction of the central node's capacity that is available to its children decreases in proportion to the number of clients, so since the capacity is fixed scaling means a reduction of workflow on the children proportional to their own number [1]. In a chain, every node can use its maximum processing capacity on its neighbour and that chain can grow as long as we like until the load of data aggregation (which grows in proportion to its length) becomes a significant burden.

There are thus advantages and disadvantages to each of these structures, with regard to both organization and processing capacity. A tree is essentially a compromise between the two: any tree can be seen as a number of stars chained together. We must decide as a matter of policy what *node degree* or number of branches these stars should have in order to compromise on these two dipolar effects of growth.

One interesting example in which the topology of a network pattern could have a direct effect on scalability is power consumption. Since we envisage network patterns finding application in mobile ad hoc networks which run off batteries, e.g., sensor networks with limited resources, we should think about the possibility that the choices we make will affect the lifetime of the devices. Power consumption too might have to be traded against speed and accuracy.

We have no generic answer to the question of which kind of structure is best in a given case, 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 [12] squared. Thus if we design at maximum utilization to cope with demand from aggregation of k neighbours, we must scale cost 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 we have a cost function that is a balance between these two

$$Cost = \alpha k^2 + \frac{(N-1)}{k}. \qquad (4)$$

A plot for this for the arbitrary policy $\alpha = 0.1$ is shown below. This shows the existence of an optimum aggregation degree, in this case $k = 5$. If k were a constant all over the network, i.e., the network formed a regular graph, this would be the optimal answer for minimizing power consumption. However, there are many constraints in ad hoc networks that would make it unlikely to be able to maintain such a regular tree, moreover there are other concerns than power consumption. In general one must compromise between a number of different optimization parameters competing for attention. More detailed considerations then need to be applied to the problem. As we see, the cost rises sharply with increasing centralization, however this does not help roaming hand-held devices with limited range that both cannot centralize and do not want the computational burden focused in one place.

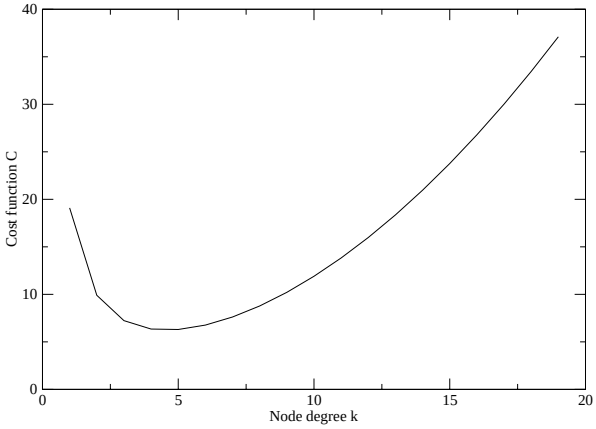

Figure 6: Cost considerations can plausibly lead to an optimum depth of network pattern when power considerations are taken into account. The minimum cost here is given for $k = 5$. Such considerations require an arbitrary choice to be made about relative importance of factors.

Our work here does not offer a simple answer to this conundrum, but shows network managers how to investigate and locate their own compromise as a matter of policy.

Conclusions

In the present work we have provided a proof of concept for implementing network data aggregation and dissemination patterns at the host level, using promise theory inspired methods. We have shown that we can avoid scalability bottlenecks only at the expense of temporal fragmentation of data. If users make logical star networks, they will have the greatest level of certainty about their data but the most fragile architecture in the face of growth. If they choose a number of star topologies chained together they can make a suitable compromise. Most importantly, we point out that the uncertainties incurred should be monitored and presented as part of the data's time-stamps.

We feel that our hybrid network/system study is a stepping stone towards integrating host and network administration within a common framework. Our work has been based on KTH's distributed protocols, and our investigation must be seen as tentative. We have not implemented all the features of the GAP protocol here. The adaptive creation of a network overlay is a topic for a later time, nevertheless some experimental peer to peer features of cfengine are already similar to the ideas used in GAP, and we intend to explore these further. Some partial approximations for this are implemented as SelectPeerNeighbours, SelectPeerLeader functions in cfengine, with failover options. However, the full details of the algorithm still have to be understood. This will probably take another six months to a year to find the time to complete. Tests are proceeding and will drive a discussion as to the most appropriate way for deciding a topology in a cfengine peer network.

Our microscopic investigation of propagation uncertainty in [9] shows that distributed structures lead to uncertain results. The uncertainties measured in a cfengine network are not simply related to errors in aggregation due to unreliable nodes, as studied in [5, 11], so it is not clear whether the generalization A-GAP would be a realistic solution to the problem here.

The syntax of cfengine's voluntary cooperation model is based on peer to peer interactions, just like promise theory. It was designed with simple one-to-one contracts in mind. We did not consider the possibility of widespread interconnection of contractual relationships. This results in clumsy and cumbersome policy files for encoding patterns in cfengine. Further work is expected to be able to enable regular expressions of some form to more efficiently encode the bilateral promises required for pattern policies.

As we write this, the team at Stockholm has developed a new pattern which they refer to as MGAP, in which every node in a structure can receive a copy of the total aggregate. It seems likely that this pattern will find a special place in cfengine for extending cfengine's peer to peer monitoring capabilities. We look forward to reporting on this is future work.

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

Author Biographies

Mark Burgess is professor of Network and System Administration at Oslo University College. He was the first professor with this title. Mark obtained a Ph.D. in Theoretical Physics in Newcastle, for which he received the Runcorn Prize. His current research interests include the behaviour of computers as dynamic systems and applying ideas from physics to describe computer behaviour. Mark is the author of the popular configuration management software package cfengine. He made important contributions to the theory of the field of automation and policy based management, including the idea of operator convergence and promise theory. He is the author of numerous books and papers on Network and System Administration and has won several prizes for his work. Reach him electronically at Mark.Burgess@iu.hio.no .

Matthew Disney has been working in systems administration since 1998. He has a B.S. in Computer Science from the University of Tennessee and an MS in Network and System Administration from the University of Oslo. He is currently working as a cyber security administrator at Oak Ridge National Laboratory.

Rolf Stadler is a professor at the Royal Institute of Technology (KTH) in Stockholm, Sweden, since 2001, where he leads the network management group. He received an M.Sc. degree in mathematics and a Ph.D. in computer science from the University of Zurich, Switzerland, in 1984 and 1990, respectively. Over the last 10 years, Dr. Stadler has been instrumental in the network management research community and served as PC co-chair for premier IEEE conferences in the field, including DSOM'99, NOMS'02, and DSOM'07. He further serves on the editorial board of IEEE Transactions on Network and Service Management (TNSM). His current research interests include scalable networks and systems, autonomous computing, and self management.

Bibliography

[1] Burgess, M. and G. Canright, ''Scalability of Peer Configuration Management in Partially Reliable and Ad Hoc Networks,'' *Proceedings of the VIII IFIP/IEEE IM Conference on Network Management*, p. 293, 2003.

[2] Burgess, M. and G. Canright, ''Scaling Behaviour of Peer Configuration in Logically Ad Hoc Networks,'' *IEEE eTransactions on Network and Service Management*, Vol. 1, Num. 1, 2004.

[3] Burgess, M. and K. Begnum, ''Voluntary Cooperation in a Pervasive Computing Environment,''

Proceedings of the Nineteenth Systems Administration Conference (LISA XIX), USENIX Association, Berkeley, CA, p. 143, 2005.

[4] Lima, K-S and R. Stadler, ''A Navigation Pattern for Scalable Internet Management,'' *Proceedings of the VII IFIP/IEEE IM Conference on Network Management*, 2001.

[5] Gonzalez, A., Prieto, and R. Stadler, ''Adaptive Distributed Monitoring with Accuracy Objectives,'' *ACM SIGCOMM Workshop on Internet Network Management (INM 06)*, Pisa, Italy, 2006.

[6] Dam, M. and R. Stadler, ''A Generic Protocol for Network State Aggregation,'' *RVK 05*, Linkping, Sweden, June 14-16, 2005.

[7] Tel, G., *Introduction to Distributed Algorithms*, Cambridge University Press, 2nd Edition, pp. 181-202, 2000.

[8] Chang, E. J. H., ''Echo Algorithms: Depth Parallel Operations on General Graphs,'' *IEEE Transactions on Software Engineering*, Vol. 8, Num. 4, pp. 391-401, 1982.

[9] Disney, M., *Exploring Patterns for Scalability of Network Administration with Topology Constrants*, Master's Thesis, Oslo University College, 2007.

[10] Burgess, Mark, ''An Approach to Understanding Policy Based on Autonomy and Voluntary Cooperation,'' *IFIP/IEEE 16th International Workshop on Distributed Systems Operations and Management (DSOM)*, LNCS Vol. 3775, pp. 97-108, 2005.

[11] Wuhib, F., M. Dam, R. Stadler, and A. Clemm, ''Robust Monitoring of Network-Wide Aggregates Through Gossiping,'' *10th IFIP/IEEE International Symposium on Integrated Management (IM 2007)*, 2007.

[12] Burgess, M. and F. Sandnes, ''A Promise Theory Approach to Collaborative Power Reduction in a Pervasive Computing Environment,'' *Springer Lecture Notes in Computer Science*, LNCS Vol. 4159, pp. 615-624, 2006.

Appendix: Examples

```
#################################################
#
# CHAIN 4 machines 1,2,3,4 (promise chain)
#
#################################################
classes:
 always = ( any )
 leaf   = ( node4 )
 root   = ( node1 )
#################################################
control:
  workfile   = ( "/tmp/chain-pattern" )
#################################################
methods:
  #
  # Pattern has to be coded in classes (from)
  # and servers (to)
  #
node1|node2::             # -b | +b - binding
  Aggregate("$(workfile)")
     server=node2
     action=method_pattern.cf
     returnvars=ret
     returnclasses=chain_link
node2|node3::
  Aggregate("$(workfile)")
     server=node3
     action=method_pattern.cf
     returnvars=ret
     returnclasses=chain_link
node3|node4::
  Aggregate("$(workfile)")
     server=node4
     action=method_pattern.cf
     returnvars=ret
     returnclasses=chain_link
#################################################
editfiles:
   !leaf::
   { $(workfile)
   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "$(Aggregate.ret)"
   # Handle errors so no strange loops
   ReplaceAll "Aggregate.ret" With "FAILED"
   }
  leaf::
   { $(workfile)
   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "$(value_loadavg)"
   }
#################################################
alerts:
  root.Aggregate_chain_link::
   "Chain aggregate $(n)$(host)=$(value_loadavg)
 at $(date) $(Aggregate.ret) "
```

Figure 7: A promise chain fully represented as a contract between parties by voluntary cooperation.

```
##############################################
#
# Netlab config
#
##############################################
classes:
 leaf   = ( netlab4 )
 root   = ( netlab1 )
##############################################
control:
  workfile   = ( "/tmp/chain-pattern" )
  tempfile   = ( "/tmp/chain-temp" )
netlab1::
  serve = ( netlab3 )
netlab3::
  serve = ( netlab4 )
##############################################
tidy:
##############################################
copy:
!leaf::
  $(workfile)
    dest=$(tempfile)
    server=$(serve)
    type=checksum
    define=success
    elsedefine=failure
##############################################
editfiles:
  success::
   { $(workfile)
   AutoCreate
   EmptyEntireFilePlease
   InsertFile "$(tempfile)"
   AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
   }
  failure::
   { $(workfile)
   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "copy-chain - no response from $(serve)"
   AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
   }
  leaf::
   { $(workfile)
   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
   }
##############################################
alerts:
  success::
   "Chain update succeeded"
   PrintFile("$(workfile)","6")
  failure::
   "No Chain update at $(date)"
```

Figure 8: A simplified version of the promise chain built using a simple pull method. This is much more trusting than the previous example and assumes a certain control over the children.

```
#################################################
#
# Depth aggregation (promise tree)
#
#################################################
classes:
 leaf         = ( netlab3 netlab4 )
 aggregator   = ( netlab1 )
#################################################
control:
  workfile   = ( "/tmp/chain-pattern" )
  children = (
             A(netlab1,"netlab3,netlab4")
             A(netlab3,"netlab3,netlab4")
             A(netlab4,"netlab3,netlab4")
             )
#################################################
methods:
netlab1|netlab3|netlab4:: # 2 servers, 1 client
  Aggregate("$(workfile)")
     server=$(children[$(host)])
     action=method_pattern.cf
     returnvars=ret
     returnclasses=chain_link
#################################################
editfiles:
   aggregator::
   { $(workfile)
   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "$(Aggregate_1.ret)"
   AppendIfNoSuchLine "$(Aggregate_2.ret)"
   # Handle errors so no strange loops
   ReplaceAll "Aggregate.*ret" With "FAILED"
   }
  leaf::
   { $(workfile)
   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "$(average_loadavg)"
   }
#################################################
alerts:
  aggregator.(Aggregate_1_chain_link|Aggregate_2_chain_link)::
   "Chain aggregate $(n)$(host)=$(average_loadavg) at $(date) \
    $(Aggregate_1.ret) $(Aggregate_2.ret) "
```

Figure 9: A two to one aggregation of text data. This example uses a full promise approach.

```
##############################################
#
# Breadth aggregation by pull
#
##############################################
classes:
 leaf   = ( netlab4 netlab3 )
 root   = ( netlab1 )
##############################################
control:
  Split      = ( , )
  workfile   = ( "/tmp/chain-pattern" )
  tempfile   = ( "/tmp/chain-temp" )
 #                                  1
 # One link in a binary tree      / \   aggregation
 #                               3   4
netlab1::
  serve = ( "netlab3,netlab4" )
##############################################
copy:
!leaf::
  $(workfile)
    dest=$(tempfile)_$(this)
    server=$(serve)
    type=checksum
    define=success
    elsedefine=failure
##############################################
editfiles:
  success::
   { $(workfile)
   AutoCreate
   EmptyEntireFilePlease
   InsertFile "$(tempfile)_$(serve)"
   AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
   }
  failure::
   { $(workfile)
   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "copy-chain - no response from $(serve)"
   AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
   }
  leaf::
   { $(workfile)
   AutoCreate
   EmptyEntireFilePlease
   AppendIfNoSuchLine "copy-chain $(host)=$(value_loadavg) at $(date)"
   }
##############################################
alerts:
  success::
   "Chain update succeeded"
   PrintFile("$(workfile)","6")
  failure::
   "No Chain update at $(date)"
```

Figure 10: A simpler pull version of the aggregation example.