Auto-configuration by File Construction: Configuration Management with Newfig

William LeFebvre and David Snyder – CNN Internet Technologies

ABSTRACT

A tool is described that provides for the automatic configuration of systems from a single description. The tool, newfig, uses two simple concepts to provide its functionality: boolean logic for making decisions and file construction for generating the files. Newfig relies heavily on external scripts for anything beyond the construction of files. This simple yet powerful design provides a mechanism that can easily build on other tools rather than a single monolithic stand-alone program. This provides for a great deal of flexibility while maintaining simplicity. The language is a combination of boolean logic and output statements, and also provides for macros and other essential elements. All output is written to channels: an abstraction which provides for extensive configurability. Examples are provided that show the tool's power and flexibility. A description is also provided of the efforts undergone at CNN to integrate this tool in to a sizable infrastructure. The paper concludes with a discussion of future improvements.

Introduction

As the number of systems in an infrastructure grows, the administration problem grows with it. Unless the engineering staff grows accordingly, at some point the management of the systems will need to be automated. This realization is nothing new, and several successful tools have been developed to meet this need.

Configuration data used by a system to control how it behaves can be broken down into three basic categories: information which is unique to a system, information which is common to a proper subset of systems, and information which is common to all systems in the infrastructure. Unique information includes a system's hostname, local disk partitions, and network address assignments. Global information includes such files as */etc/services* and */etc/networks*. The partially global information is the most interesting: it is common across systems which share a similar function but not necessarily others. The task of configuring a system requires that all of this information be in place and correct. The most complicating factor in achieving a correct configuration is in the management of files which combine data from more than one category.

A file that only contains unique data can be copied into place when a system is installed. Such files are rarely touched after system installation. Files that contain only global information can be easily updated from a central repository. Likewise, files that contain partially global information can be updated from a central authority provided there is some mechanism that distinguishes among the different classes of systems and is able to determine which data are appropriate for the system. However, when a file contains a mixture of these data categories, its maintenance becomes significantly more difficult. This is most noticeable in files such as *fstab*, *inetd.conf*, *hosts.allow*, rc script directories, and sometimes *passwd*.

When we sought out a tool for automated systems configuration, we were looking for certain properties that we believe would be beneficial in our environment. We wanted the system to be idempotent, congruent, deterministic, transportable, extensible, and fail-safe. We wanted a specification language that was both clear and concise, to minimize training and maximize understanding. When a tool could not be found that met all of these criteria, we set out to design a new solution to the problem. The result, *newfig*, takes a new approach to the problem while providing all of the qualities that we believe are important. As a result, very little is built in to *newfig*. Instead it is a framework which can utilize other tools and scripts to accomplish results. Rather than provide a wide range of built-in mechanisms, *newfig* uses file construction as its only primitive operation. Its configuration is a purely declarative language based on boolean logic. The files that *newfig* constructs, called *channels*, can be used to replace existing files on the system or as input to external programs (including scripts). As a result, the system is naturally extensible. *Newfig* is used to generate input for and monitor the execution of the programs and scripts which perform the actual modifications to the system. It provides a structure around which system administrators can do what they do best: automate through scripting. We believe that the resulting system meets all our initial design goals and provides us with an excellent platform for automated configuration.

Related Work

Prescriptions [8] is a declarative language for describing the desired state of configuration for distributed systems. It provides mechanisms for specifying operations that may be used to bring systems into conformance with a specification. However, it does

not seem to provide mechanisms for file distribution and synchronization. We have not been able to find recent work on this project since Thornton's 1994 technical report.

CFengine [1] is the most widely known work in this area. It is to automated systems configuration what awk(1) is to scripting. The main concepts are patterns, actions, and an execution context that is managed by an interpreter. It provides a rich body of built-in actions, but has little room for expansion beyond that set. The configuration drives actions to perform on a system, whereas *newfig* provides a description of the desired target for the system and enforces conformity to that description. CFengine implements convergence by providing a mechanism that brings a system closer to an ideal. In our environment we sought a tool that implements congruence, which is a tighter standard than convergence. The idea of file construction is not central to CFengine, and a CFengine configuration that uses such a methodology tends to be cumbersome. CFengine is also not able to remove changes implicitly: such steps must be explicitly given in the configuration.

Psgconf [6] takes a highly modular approach to configuration management, providing hooks for various data stores, policy rules, and actions. The configuration parsing is order dependent, making psgconf a procedural configuration tool rather than declarative, which has some advantages and some drawbacks.

Site [3] uses declarative statements to describe the configuration of a computing site at three levels of abstraction. At the lowest levels, drivers written in C are used to construct the contents of configuration files. The paper describes a prototype implementation only. In contrast, *newfig* provides no restrictions on the language used to construct channels, which is good for admins who have long since shed their systems programming scales (or never had them to begin with).

PIKT [5] is an interpreted scripting language, preprocessor, and scheduler that is primarily intended to monitor systems, reporting problems and taking corrective action when possible. Over time, it has been extended to include configuration management features, but most of the terminology in the language is built around monitoring. For example, scheduling periodic execution of a script involves adding it to the "alarms" section of the alerts.cfg file.

Radmind [2] takes a file based approach to configuration management by integrating intrusion detection with centralized system management. An advantage is that complex, out-of-band changes can be captured and incorporated into the configuration, but maintaining consistency of configuration data that is duplicated across multiple files may present a challenge without factoring tools.

ISconf [9] is a highly order dependent configuration tool based on make(1) files. A description of changes is provided to the tool, and it ensures that those changes are carried out on each system in an exact order. ISconf version 2 requires that the description be created and maintained manually, whereas later versions provide more automated ways for generating the description. The basic premise of ISconf is that "order matters": it is easier to replicate the order in which operations are conducted than it is to determine and accommodate the interactions of those operations. Like newfig, ISconf implements congruence. The difficulty with ISconf is the monotonic increase to the description and the steps required to recreate a system. As changes are piled on top of changes, the time required to build or rebuild a system continues to increase. Although preservation of the order of changes is sufficient to achieve congruence, it is our belief that it is not necessary.

Design

Newfig is a system designed to provide for the automatic configuration of individual machines from a common description. Boolean algebra is used to control the generation of output to a number of channels. Each channel can be used to control the contents and characteristics of a file. Channels can also be used as input to scripts for operations which are more complicated than basic file construction. All channel definitions are part of the configuration, allowing the functionality of *newfig* to be extended with ease. *Newfig* is designed to be idempotent, transportable, extensible, conformant (rather than convergent), and fail-safe.

The configuration consists of a series of boolean phrases interspersed with output statements. Boolean algebra is used as the logical structure for the *newfig* configuration language. Clauses are used to infer the logical value of a symbol from other symbols. The algebra supports the three basic logical operations: and, or, not. Parentheses are also recognized for grouping operations. Between the boolean clauses are statements that send lines to channels. Each channel must be explicitly defined in the configuration along with its characteristics. A channel can be associated with a file, in which case its contents becomes that of the file. A channel can also be associated with an external command or script, in which case the script is used to process the channel's contents. External commands are also used to perform syntax and semantic checks of channels' content to ensure correctness.

Processing is performed in several distinct phases in *newfig*: read, intrinsic definition, inference, macro definition, generation, filtering, instantiation. No changes are performed on the system until the instantiation step, giving *newfig* ample opportunity to discover problems before changes are made. If any problems are detected before instantiation, *newfig* will be fail-safe and not make any changes to the system.

Decisions about a system are solely dependent on the boolean clauses in the configuration; the role of

the first few phases is to evaluate the clauses. First, the entire configuration is read in and parsed. Then certain facts about the system are used to determine a set of intrinsically true symbols. The name of the system is one such symbol: it is always true. Additionally, the following symbols are intrinsically true: the name of the operating system (in all lower case), the name combined with the operating system release, and the platform type. Thus, a system named *sammy* running Solaris 9 on a SPARC platform will have the following intrinsically true symbols: *sammy*, *sunos*, *sunos-5.9*, *sparc*. Another intrinsically true symbol represents the network of the system's IP address (or addresses). For example, a system with the address 10.5.2.12 would have the symbol net.10.5.2 defined as intrinsically true. The configuration can also invoke external commands to augment the set of intrinsic symbols with either true or false values.

Once the intrinsics have been determined, the values of the other symbols in the configuration are determined through *inference*. Not every symbol's value can be determined. When inference is complete there will be a set of symbols known to be true, a set known to be false, and a set of symbols whose values are unknown. For all the remaining phases, the only symbols which matter are those known to be true.

The configuration language allows for the definition and expansion of macros in the statements and the channel declarations. In the *macro definition* phase, the values for all macros are determined. A special append operator (+=) is available to add to an existing macro definition, such as a PATH.

The *generation* phase creates the content of every channel. This is done by processing the output statements associated with true symbols.

Once the contents of each channel is known, *syntax checking* is performed by an external program as specified in the channel definition. Since *newfig* itself has no knowledge of the intended content of a channel, syntax checkers provide a way to verify that a channel's content are correct. Although this phase is named *syntax* checking, any sort of checks can be carried out by an external program. Examples of checks which could be performed are: ensure that each line of a channel has the correct number of fields, ensure a passwd channel has a line for root, verify that every program listed in *inetd.conf* exists. If any of the syntax programs indicates an error, then *newfig* will fail-safe and not make any changes to the system.

A close variant to syntax checking is the *filtering* phase. This phase is similar to syntax checking, except that the external programs invoked in this phase alter the content of the channel in addition to performing simple syntax checking. Each program acts as a Unix-style filter, reading the channel contents from standard input and writing to standard output. The results of the filter are used to replace the content of the channel. Although the order of records in most files is unimportant, it may be desirable to sort the output to improve human readability. The passwd file is commonly sorted by UID, and a filter could easily perform this task for generated passwd files. Optimization is another good use of filtering. A generated hosts.allow file may contain overlapping address ranges, and a filter could remove them as well as reformat the output to improve readability. It is important to the integrity of the entire system that filters perform no side effects and that their actions are restricted to producing output and error messages.

The final phase is *instantiation*, which is performed once all of preceding phases have executed without error. This phase ensures that the underlying system matches the configuration. For file channels, the file is compared to the generated content of the channel. If there is no difference, the file is left untouched (thus modification times are unchanged). If the channel contains different content, a new copy of the file is created and put into place. A file channel can also specify ownership and permission modes for the file. An action channel has its action command invoked with the channel contents available as standard input. Action commands usually perform side effects.

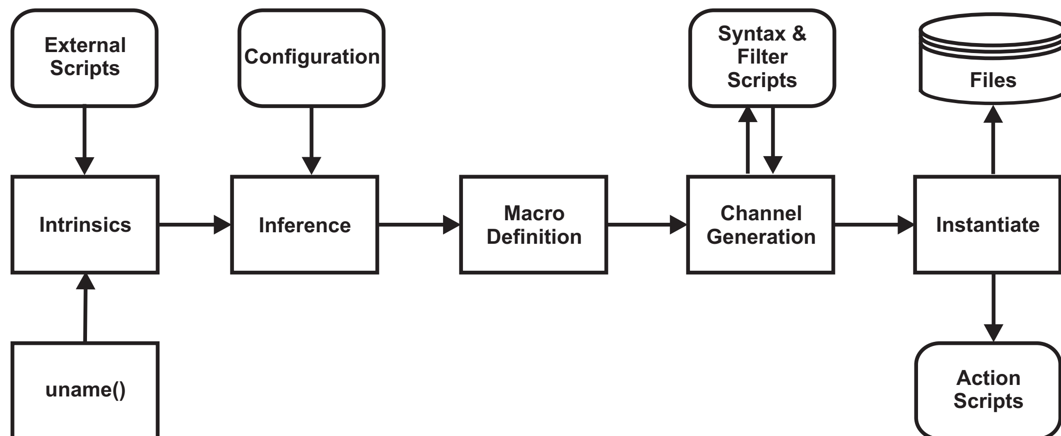

Figure 1: The phases of newfig.

A channel can have both a file and an action associated with it. In such cases, the action is only performed when the file is changed. Thus an action can be used to restart or "tickle" a daemon only after its configuration file has been updated by *newfig*. Figure 1 show the basic flow of information through the various phases.

Configuration Specifics

Each configuration statement has two separate components: a logical relationship and a body of statements. A statement need not contain both components.

Logical Clauses and Expressions

The logical relationships are specified using a simple boolean algebra. The rules for symbol names are very generous, allowing numbers, letters, and many special characters. Expressions can contain and, or, inversion, and grouping. There are two ways to specify a logical relationship: a standard clause and implication. A standard clause takes the following form:

```
symbol: expression
```

If the value of the expression can be determined, then it becomes the symbol's value. An expression consists of any combination of the following elements:

```
or      symbol1 symbol2
xor     symbol1 ^ symbol2
and     symbol1 & symbol2
not     !symbol1
```

Parentheses can be used to group expressions together, as in:

```
symbol: !(a b c d)
```

Wherever a clause can be used, an expression may also be used. An expression is just a clause with no left hand side, as in:

```
!(a b c d)
```

The second relationship is implication, where a single symbol's value is used to imply the value of a list of symbols (expressions do not make sense in this context). This takes the following form:

```
symbol -> symbol1 symbol2 symbol3 ... ;
```

This is equivalent to:

```
symbol1: symbol;
symbol2: symbol;
symbol3: symbol;
...
```

A clause may be optionally followed by a body. When present, this body must be contained in braces. The body consists of a mixture of any number of the following: a statement that sends output to a channel, a macro definition, or an expression with its own body. A body is processed when the symbol or expression with which it is associated is known to be true.

The symbols *all* and *true* are preassigned the value of true, and the symbol *false* is preassigned the value of false. Other symbol values are determined by intrinsic settings. Symbols which appear in the configuration only on the right hand side of a clause (or only on the left hand side of an implication) are called terminal symbols. The value of a terminal symbol cannot be determined by anything in the configuration. Terminal symbols which do not have an intrinsic value are assumed to be false.

Macros

Macros can be defined and expanded much like Makefile macros, although all macros are created before any statements (thus any expansion) takes place. Macros are created as follows:

```
MAC=value;
```

There is a special append operator to allow definitions to be augmented:

```
MAC+=value;
```

When expanded, such macros will have spaces separating each added component. Expansion is invoked with a dollar sign followed by the macro name in braces, for example:

```
${MAC}
```

Some macros have special meaning. The PATH macro is used as the execution path for all commands invoked by *newfig*. HOSTNAME is preset to the name of the current host. Unlike other statements, macro definition can appear outside of a clause.

Much like make(1), all macros are defined before any are expanded. Thus definition need not precede use in the configuration. The order in which macros are defined is only significant in two cases. First, if a macro is redefined then the last definition will be used. Second, the order in which macro append definitions are processed will determine the order in which the strings appear in the final macro. Currently there is no way to directly control the order in which definitions are processed other than basic sequential order.

Channel Statements

Channel statements are very simple. The statement begins with a channel name, is followed by any number of fields, and ends with a semi-colon. When processed, all fields are written to the named channel, and separated by a single space. Here are some examples of channel statements:

```
resync /opt/proftpd;
hosts.allow "ALL: 127.";
inetd.conf telnet stream tcp nowait
  root /usr/sbin/in.telnetd in.telnetd;
```

The usable characters in an unquoted field are limited. Quoted strings are written exactly as they appear after macro expansion. A dollar sign can be represented with two dollar signs. In the second example, the macro *MAC* is not expanded:

```
"string ${MAC}"
"string $${MAC}"
```

A special form of channel statement is the %include statement. This allows the entire contents of an existing file to be included in a channel. For example:

```
%include passwd /etc/base.passwds;
```

Clause and Body

The entire clause with its body must end with a semi-colon. Thus any of the following are valid clauses:

```
symbol: !(a b c d)
{
    channel line;
};
symbol: e & f;
symbol1:
{
    channel line2;
};
symbol2
{
    channel line2;
};
```

A statement or a macro definition within a body may be guarded with a boolean expression. Thus the following two clauses do the same thing:

```
x { y { channel a; }; };
x & y { channel a; };
```

The first form is more useful when the body has a number of statements, only one of which needs to be guarded, as in:

```
x {
    ch1 a;
    ch2 b;
    y { ch3 c; };
};
```

In this form, ch1 and ch2 receive output whenever x is true, but the ch3 statement is only processed when both x and y are true. This can be useful in situations where certain statements are needed only for particular operating systems.

If a clause defines the value of a symbol, then any body associated with the clause is performed whenever that symbol is true, even if it became true via a different clause. Consider the following example:

```
x: a b
{  output line1; };
x: c d
{  output line2; };
```

Note that both lines are sent to the output whenever the symbol *x* is true. More specifically, if *a* is true while *b*, *c*, and *d* are false, then the symbol *x* is asserted to be true and both lines are sent to the output, even though the second expression (“c d”) is false.

Channel Definitions

There are no predefined channels. All channels must be explicitly defined in the configuration. This definition is done with a %channel clause. Following the keyword %channel is the channel name and then, in braces, its definition. For example:

```
%channel hosts {
    file /etc/hosts;
};
```

A channel may have a number of characteristics as declared in the body of the channel definition. However, a channel may only be defined once.

A channel may take on several forms, sometimes in combination. A channel that is associated with a file, or “file channel,” ensures that the file contains exactly what appears in the channel. During instantiation, the contents of the channel are compared to the file, and if they differ, the file is rewritten to exactly match the channel. If they are the same, the file is left untouched. Mode and ownership are also compared and corrected where necessary.

An “action channel” has an action associated with it. The action is an external program that is run during the instantiation phase and uses the channel contents as standard input. It is expected that an action program will perform side effects.

A channel may be both an action channel and a file channel. In these cases, the action is only performed when the file is changed. This form is useful for updating daemon configuration files, as the action can ensure that the daemon is signaled or restarted.

A directory may be associated with a channel to form a “directory channel.” Such a channel is expected to contain a list of files (one per line). The target directory is checked to ensure that it contains the listed files and nothing else. The contents of those files is checked and updated. Consider the following series of statements where “xinet” is a directory channel for /etc/xinetd.d:

Property	Definition
action	external action program
directory	target directory for directory channel
file	target file for file channel
filter	external program to check and augment channel contents
syntax	external program to check channel contents
owner	required file owner (for file and directory channels)
group	required file group (for file and directory channels)
mode	required permissions (for file and directory channels)
singlesource	channel contents may only come from one body
after	channel must be processed after another channel

Figure 2: Channel properties.

```
xinet /opt/proftpd/xinetd/proftpd;
xinet /opt/rsync/xinetd/rsync;
```

Each of the named files will be copied in to the target directory, and *newfig* will ensure that no other files exist in that directory.

Channel Properties

Figure 2 shows properties that can be set for each channel.

External Programs

The filter, syntax, and action properties invoke external programs, either binaries or scripts. The interface for all of these programs is the same. The contents of the channel is readable on standard input. Programs that exit with a zero status code indicate normal success, a non-zero exit code indicates that an error occurred. Lines written to standard error are considered to be error messages. If formatted correctly then *newfig* will be able to match the message up with the line that caused it. The error message format is a line number followed by a colon then the message.

Filter programs must write a revised copy of the channel contents to standard output. These results will be taken as the new channel contents. Error messages generated by filters are treated the same as the ones generated by a syntax program. Note that, in both cases, the contents of standard error is ignored unless the program exits with a non-zero status code.

In order to preserve the idempotent characteristic of *newfig*, syntax and filter programs should not produce any side effects. This includes modifying, creating, or removing files, and signaling, stopping, or starting processes. These programs can, however, use temporary files provided they are removed when the program finishes. The action programs are expected to produce side effects: that is their purpose. Although the channel contents is made available on standard input, an action program need not read it.

Interesting Properties

Newfig has a number of interesting properties that make it a strong utility. These features are generally desirable in a tool that automatically adjusts a system's configuration and behavior.

Idempotency

An operation that is *idempotent* is one that acts as if it was only invoked once even if it is invoked multiple times. An idempotent operation does not have a cumulative effect. *Newfig* is designed to be idempotent, but its ability to retain this characteristic is entirely dependent upon the action commands that it invokes.

The intrinsic symbol values for a system are entirely dependent upon characteristics of the system itself: its platform, operating system, and IP address. As long as these remain constant, the intrinsic symbols will always have the same value. However, it should be noted that the system does provide an external mechanism for augmenting the intrinsic set. If this mechanism does not provide a constant set of results, the idempotent behavior may be compromised.

As long as the intrinsic set remains constant between invocations, the results of inference will always be the same. This is insured by the boolean algebra which drives the inference phase. The set of true and false symbols derived from inference is the only means of selection for the remaining phases, guaranteeing that their results will always be the same. The only exception to this is the use of external programs for syntax, filtering, and action. Both syntax and filter commands are required to have no side effects, thus they can easily be idempotent. This leaves the action scripts. *Newfig* performs idempotent operations if and only if the action scripts it invokes also perform idempotent operations.

Transportability

The term *transportability* is being adopted to describe a characteristic of *newfig* that is unique to an automated configuration tool. A tool that is transportable is one that is capable of creating the same result regardless of the actual system on which it is run. Transportability is essential for the testability of a site's configuration. In order to perform regression tests on an infrastructure configuration, the testing mechanism must be able to determine the results of the configuration tool for a variety of systems (perhaps all) in the infrastructure. Without transportability, the generation of this data must take place on each system in the test set.

Newfig provides transportability through its strict use of boolean algebra to drive all the decisions. Assume that both the *newfig* configuration files and the external programs used by *newfig* are constant (functionally equivalent) across the infrastructure. All that *newfig* needs to be able to generate the output of each channel for a given host is the intrinsic set and the list of predefined macros for that host. *Newfig* provides mechanisms for generating just this information and using it instead of the local set. This provides transportability.

Extensibility

One of the problems with systems such as CFengine is the limited range of capability. In fairness,

```
samba {
    inetd.conf netbios-ssn stream tcp nowait root /opt/samba/bin/smbd smbd;
    inetd.conf netbios-ns dgram udp wait root /opt/samba/bin/nmbd nmbd -d2;
    resync /opt/samba;
};
alpha -> samba;
```

Figure 3: Sample Samba configuration.

CFengine has a great deal of functionality already built in, but its ability to extend that functionality is limited. *Newfig* is designed to be extensible through extensive use of external commands. These commands may be anything that can run on the native system: scripts, perl, python, pre-compiled binaries, *etc.* There is very little capability actually built in to *newfig*. The design philosophy is similar to that of the original Unix: *newfig* is a tool which can easily be used as a building block for other tools.

Conformance

The configuration for *newfig* is a complete description of the files which it controls. Rather than providing a series of steps to edit an existing file, the configuration provides enough information to reconstruct the entire file. This approach ensures that the act of removing data from a file's description will get implemented correctly on the affected machines.

Some automated configuration systems, such as CFengine, provide convergence toward an ideal. In some environments, especially those with loose control over root access, convergence is an appropriate tool for effective centralized management. However, installations which primarily consist of servers and where configuration changes are typically co-ordinated by a single organization do not need the flexibility of convergence. It is simpler to provide a complete description of a configuration and enforce conformance to it. This ensures that changes are implemented completely and with a single iteration. A complete configuration is also descriptive in what it does not contain, making the removal of unnecessary items simpler.

Consider a system that is configured to include a samba [7] server. This configuration might look like the one in Figure 3.

Such a configuration would ensure that any system for which the symbol *samba* is true would have the lines needed for samba in inetd.conf and the directory /opt/samba synced up correctly with a central repository. The last line of the configuration ensures that when *alpha* is true *samba* is true also. Thus the system *alpha* would have the smbd and nmbd lines added to inetd.conf. Other parts of the configuration would contain the remaining lines that are expected to appear in inetd.conf. Now consider what happens when the association between alpha and samba is removed, such as would be the case if it was decided that alpha should no longer provide samba service. The next time *newfig* is invoked, it will rebuild all the channels, including the inetd.conf channel. But this time it will build it without the smbd and nmbd lines. *Newfig* will detect that the resulting channel is different from the file inetd.conf and will instantiate the new channel contents as inetd.conf. Finally, if the channel definition for inetd.conf has an action command, *newfig* will run the action providing an opportunity to send a signal to inetd. The removal of this data happens as a natural consequence of the information from the configuration itself.

Fail-safe Operation

The *newfig* design defers any modifications to the system until all other processing is complete. The soundness of the configuration is checked during reading and inference. The integrity of each channel's content can be checked and augmented with syntax and filter commands. No changes are made to the system until the final phase. If any problems are found prior to instantiation, *newfig* can decide not to continue. This provides a fail-safe mechanism to ensure that an incorrect configuration is not applied to any systems.

The original design goal of *newfig* was to provide an entirely fail-safe design: any sort of errors would prevent *newfig* from instantiating any changes and executing any action. During the deployment of this 100% fail-safe model we discovered that this may not be a desirable design.

One obvious function that *newfig* can supply is driving the distribution of files from a central repository (commonly called a *gold* server). For example, a channel can contain the names of directories and files which must be kept in sync with a central server, and the action for that channel can provide the mechanism which performs the syncing. Such a channel is an obvious choice for keeping the *newfig* configuration itself in sync. Unfortunately, if *newfig* is 100% fail-safe, then any error in the configuration will completely disable this mechanism, making it impossible to recover without a mechanism outside of *newfig* itself.

Experience with a 100% fail-safe system has made it obvious that certain types of errors need not hinder the update of unrelated items. As an example, consider a configuration which maintains password files and hosts.allow files. A mistake in an entry for hosts.allow files could be something as simple as forgetting the dot at the end of a network pattern (such as "172.16.1" without the trailing dot). A properly written syntax command will catch that mistake and correctly flag it. However, a 100% fail-safe design will also prevent the passwd file from being updated, even though it has nothing to do with hosts.allow.

A better design would be to contain the failure, failing only what is affected. In the case of hosts.allow it would be contained to just its channel and none other. If the configuration states a dependency between channels, such as the *after* property, then failure of a channel should also cause failure of any channels that depend on it. *Newfig* can easily be adopted to fit this more limited idea of fail-safe.

Declarative Language

The configuration language is designed to be declarative. As a result, the order in which files are processed, and the order in which statements and clauses appear in the file do not matter. This allows

the maximum amount of flexibility for organization of the configuration information. Many of the files of concern do not depend on the ordering of lines, some important files do have this requirement. In order to accommodate this, certain guarantees are made about the order in which output statements are processed, thus affecting the order of the lines within the channels. Clauses can be processed in any order, but statements within the body of a clause will be processed in the order they appear. Line ordering in an included file will be preserved. Consider the following example:

```
x: { output a; };
x: { output b; };
```

There is no guarantee that the output will be ordered *a*, *b*. Although generally this will be true, *newfig* makes no guarantees and configurations should not rely on it. However, in the following example it is guaranteed that the lines will appear *a* followed by *b*, since both statements appear in the same body:

```
x: {
    output a;
    output b;
};
```

Examples of Practical Applications

Examples of some common applications for *newfig* are as follows.

hosts.allow

Access to services controlled by tcp wrappers [11] is set with the use of hosts.allow. This file can be controlled by *newfig*, allowing for the creation of any arbitrary hierarchy to define allowable access. The channel definition would be:

```
%channel hosts.allow {
    file /etc/hosts.allow;
    filter allow-filter;
    owner root;
    mode 444;
};
```

Although it is optional, a filter for processing this channel provides improved results. The filter can optimize the channel contents to ensure that there are no extraneous specifications in the channel. Consider the following usage of this channel:

```
all  { hosts.allow "ALL: 10.2.1.5"; };
beta { hosts.allow "ALL: 10.2.1."; };
```

For host *beta* both lines will appear in hosts.allow. A channel filter would be able to detect that the first entry is extraneous and remove it. The filter can also collapse multiple lines for the same daemon (or for ALL) in to a single line.

Services

Something as simple as /etc/services can easily be maintained by *newfig*. It's channel definition only needs to provide the link between the channel and the file. If desired, a syntax checking step can be written and added to the channel definition:

```
%channel services {
    file /etc/services;
    syntax services-syntax;
    owner root;
    mode 444;
};
```

This channel can be used in the configuration to add lines to services, as follows:

```
rsyncd: {
    services "rsync 873/tcp";
    services "rsync 873/udp";
};
```

Since *newfig* generates files from scratch, the entire contents of the file must be specified by the configuration. This means that there must be a baseline of data available to add to the services channel to ensure that all the standard entries are there. Although each entry could be listed separately, it would be easier to include the baseline from a separate file (shown here with a broken line):

```
all: {
  %include services /opt/newfig/base/services;
};
```

cron

Each crontab file needs to be controlled as a separate channel. For most systems, only root and a few system crontabs need to be managed. Since it is unwise to edit a crontab file directly, this channel uses a proxy file instead. For the best results, the proxy should be persistent across invocations of *newfig* so that crontab itself is only invoked when there has actually been a change. For ease of demonstration, it is assumed that the macro PROXYFILES contains the name of a directory that holds proxy files.

This channel definition has to define different actions for each operating system it supports, as there is wide variation in the use of the crontab command. The channel also invokes a syntax checker to ensure the channel's contents are correct.

```
%channel cron.root {
    file "${PROXYFILES}/crontabs/root";
    syntax "syntax-cron";
    linux { action "crontab -u root -"; };
    sunos { action "crontab"; };
};
```

Typical usage of this channel would be:

```
sunos {
 cron.root "10 3 * * 0 /usr/lib/newsyslog";
};
```

sysctl

Linux *sysctl* is used to configure various kernel and driver parameters at runtime. The desired settings are kept in the file sysctl.conf and the command *sysctl* is run to set the parameters. *Newfig* can easily be used to drive the generation of sysctl.conf and provides central control over these settings. If a system is configured to be a web server, then its sysctl.conf can contain the settings needed to provide maximum performance.

If it is then changed to run a database server, *newfig* would alter the contents of sysctl.conf as described by its configuration to use the appropriate settings.

A sysctl channel for linux would probably look like this:

```
%channel sysctl {
  linux {
    file /etc/sysctl.conf;
    action /sbin/sysctl -p;
  };
};
```

The linux conditional is not strictly necessary, but does allow greater flexibility in the use of the channel. On non-linux systems, the channel contents will be ignored rather than generating an error.

Symbolic Links

Although *newfig* does not provide any built-in mechanisms for managing symbolic links, adding the functionality is as easy as writing a script. All that is needed is an action script that reads pathname pairs from standard input and ensures that one is a symbolic link to the other. This is a simple script to write, and it can be made as elaborate as necessary. It would also be beneficial for the script to have a syntax checking option.

An example channel definition for handling symbolic links is as follows:

```
%channel symlink {
    syntax "link-action -s";
    action "link-action";
};
```

This channel would be used as follows:

```
sunos {
  symlink /etc/inet/hosts /etc/hosts;
};
```

Management of an application's multiple versions can be handled via symbolic links as follows:

```
proftpd-1.2.9: {
  symlink /opt/proftpd-1.2.9 /opt/proftpd;
  resync /opt/proftpd-1.2.9;
};
proftpd-1.2.10rc3: {
  symlink /opt/proftpd-1.2.10rc3 /opt/proftpd;
  resync /opt/proftpd-1.2.10rc3;
};
```

File Distribution

Newfig does not have a built-in file distribution mechanism, not even for its own configuration files. The focus of this tool was on automatic configuration, so it relies on other means to accomplish file distribution. *Newfig* can easily be used to drive a file distribution and synchronization mechanism. For the sake of example, consider a script, named *resync*, that takes a list of directories and files on its standard input. Each entry is synced up with a central server using rsync [10] (recursively for directories). The following channel can be used to feed the data to *resync*:

```
%channel resync {
    action resync;
};

all: { resync /opt/local; };
samba: { resync /opt/samba; };
```

Inet Daemon

The typical inet daemon is controlled through the file /etc/inetd.conf. However, some systems (such as Linux) have a more sophisticated inetd that is configured through a collection of files in a directory, typically /etc/inetd.d . An infrastructure with a mix of these systems can still be controlled with *newfig* but the two types of inet daemons must be configured separately. This example uses a *directory channel* and a null channel. A directory channel contains a list of files and ensures that the target directory contains each of the named files and only those files. The files can originate anywhere but will bear the same names in the target directory. Here are two definitions, one for sunos (which uses the traditional inetd) and one for linux:

```
%channel inetd {
  sunos {
    file /etc/inetd.conf;
    owner root;
    mode 644;
    syntax "inetd-syntax";
    action "pkill -1 -u root inetd";
  };
};
%channel xinetd {
  linux {
    directory /etc/xinetd.d;
    action "pkill -1 -u root xinetd";
  };
};
```

Typical usage would look like this:

```
rsyncd: {
  inetd rsync stream tcp nowait
    root /usr/sbin/in.tcpd
    /usr/bin/rsync --daemon;
  xinetd /opt/rsync/etc/xinetd/rsync;
};
```

Note that the body of the rsyncd clause specifies data for both inetd and xinetd. However, it does not need to distinguish between different system types. That distinction is made in the channel definitions, not their

```
ftp-standalone: {
    rc /opt/ftp/rc/ftpd;
};
ftp-inetd: {
  inetd "ftp stream tcp nowait root /opt/ftp/sbin/ftpd ftpd -a";
};
```

Listing 1: ftp multiplexed between stand-alone and inetd service.

usage. Thus any system which supports xinetd can be added to the channel definition.

Differing FTP Usages

Because *newfig* evaluates logical clauses declaratively rather than procedurally, it is able to detect conflicts between clauses. For example, the following two clauses are conflicting and are flagged as an error:

```
one: two;
two: !one;
```

This feature can be used to protect a configuration from implementing conflicting configurations. One example of this ftp: it can be used either from within *inetd* or as a stand-alone daemon. Sometimes there are good reasons to use both types of configurations within the same infrastructure. *Newfig* can handle this and can also guard against accidentally attempting to implement both methods on the same system.

Assume that a configuration has channels to support the configuration of inetd.conf and boot time "rc" scripts in the style of System V. Listing 1 shows how ftp can be defined to act as stand-alone in some cases and as an inetd service in others.

Individual machines are configured to request either of these two symbols:

```
server1 -> ftp-standalone;
server2 -> ftp-inetd;
```

To prevent a misconfiguration from setting both symbols for the same server, the following statement can be used:

```
false: ftp-standalone & ftp-inetd;
```

Since the symbol *false* is always false, this statement will cause a logic conflict whenever both ftp-standalone and ftp-inetd are true.

Deployment at CNN

We began roll-out of *newfig* in to the CNN web farm infrastructure earlier this year. The web farm consists of over 800 hosts running a mix of Solaris and Linux. Prior to the use of *newfig*, we had constructed a system to control file distribution utilizing rsync. A central repository (gold server) holds all the files that need to be distributed for the various platforms we support. The *resync* script uses information about the host to determine a list of directories that must be kept in sync, then uses rsync to ensure that they are. This system, part of an effort called *Unity*, is used to distribute binaries and configuration files for key services in our infrastructure, including web service and ftp service as well as patches.

The deployment of *newfig* was built upon the success of resync. The initial configuration for *newfig* completely replaced the functionality of the original resync, and its deployment was seemless. Once *newfig* was in place, we targeted hosts.allow as our first file to control: it is relatively simple to support but requires significant fine-grained control.

A few weeks of effort was spent collecting the existing settings from across the infrastructure and codifying them in the *newfig* configuration language. Our initial goal was to automatically generate hosts.allow files that were no more restrictive than the ones already in place. In may cases the resulting files were more generous. Testing of this configuration was accomplished by generating the file as /etc/hosts.allow.x, then comparing the results to the existing hosts.allow. We eventually reached a point where the host access being removed could be summarized in a short list, and we decided that each of the items on the list was acceptable or even desirable. Then we changed the configuration to generate the actual hosts.allow file. Of the nearly 800 machines in the infrastructure, we only experienced access problems with one.

The resulting configuration was approximately 2400 lines spread out across 52 files, plus four separate files utilized in *include* statements. A filter script was written to organize and optimize the channel results as well as check for errors. We found many errors in the pre-existing hosts.allow files which had gone undetected, usually a class C network specification with no trailing dot. The next goal for hosts.allow is to analyze the current access and determine what is still needed and what should be removed. We expect to dramatically simplify the final configuration by rationalizing host access across large groups of systems.

The hosts.allow experiment was sufficient to prove the viability of the project. With its success we intend to take control over the hosts file, the startup scripts, crontab files, inetd.conf, boot time configurations (especially default routes), and perhaps passwd and shadow. The hosts file poses a unique challenge. Our hosts are divided in to internal and external, depending on whether or not they can be accessed directly by the outside world. All external hosts receive a common hosts file, but internal hosts use DNS. We have found it desirable to use minimal hosts files on the internal hosts: ones that only contain information on the host itself, and the NIS and NFS servers. We want *newfig* to generate this file from a list of hostnames, ensuring that the IP addresses are always correct and providing central control over the hosts that appear in the file.

Future Work

Our experiences with *newfig* are still very limited. As the support staff becomes more accustomed to its use, we plan to extend its control to as many facets of our systems as is feasible.

One set of files we would like to control with *newfig* are the passwd and shadow files. Currently we use NIS for portions of our infrastructure and nothing but local files for the remainder. Controlling access by system or by groups of systems is easy with NIS, but extremely difficult with static files. The tradeoffs with NIS are well known, including unsuitable security and

a poor level of robustness. We could replace NIS with a system based on LDAP, but in order to reduce single points of failure we do not want any of our externally facing servers dependent on a central service. *Newfig* would provide us with the centralized control that we need, but there are security issues that need to be considered for the distribution of the shadow information.

We want to control startup scripts with *newfig*, but these pose a number of interesting problems. Startup scripts differ widely among systems, requiring either separate channels or a single channel loaded with extra information. Effective control of the startup scripts would also require automatic stopping and starting of the daemons those scripts control as scripts are added and removed. Our goal is to control all startup scripts, so that those scripts which are not needed simply will not be included in the configuration. This goal is complicated by operating system vendor patches that alter these scripts.

We have found the use of ''net'' symbols (such as ''net.10.1.2'') to describe a system's local network very beneficial for many aspects of system configuration. For example, the symbols are the basis for determining if a system is ''internal'' or ''external'' (the latter are accessible by the outside world). However, the current implementation is very limited, and assumes that network divisions all fall on the traditional class C boundary [4]. The usefulness of these symbols could be enhanced by extending the notation to something more general, especially something that includes a netmask. But the simple boolean logic makes this more difficult. Further thought in this area would be beneficial.

Newfig provides a natural way to implement conformance for data in files. When a configuration change requires lines to be removed from files which are controlled by *newfig* the changes happen automatically as a result of file construction. However, this characteristic does not extend to the side effects implemented by action scripts, such as the symbolic link example in 6.5 and the file distribution example in 6.6. In these cases, *newfig* is constructing the input to a process which generates side effects. Consider the case of symbolic links with the following example:

```
one: alpha {
    symlink /opt/etc/one /etc/one;
};
```

The first time *newfig* runs, host *alpha* will have the symbolic link /etc/one created. If *alpha* is removed from the definition of the symbol *one* then the output line will no longer appear in the symlink channel. However, the symbolic link will remain as there is nothing that will remove it. We have an idea on how this problem can be overcome and will be pursuing its implementation.

There are times when it is advantageous to impose an ordering on output statements for a particular channel. Rather than use syntactic rules to imply an ordering, we envision a way to explicitly specify interrelationships between the output lines. One possibility is to allow for the specification of priorities on output statements, with a default of 100, and ensuring that lines appear in priority order. Thus a line that must always appear first could be given a priority of 1, and a line at the end a priority of 200.

The manipulation of macros in the configuration files is well motivated but poorly implemented. They are not as intuitive as one would hope. Further work needs to be done on this concept to provide the necessary functionality in a way that is still easy to follow.

Software Availability

The software was developed internally at CNN. Its availability for public use and review has not yet been determined.

Author Information

William LeFebvre is a technology fellow and David Snyder is a Chief Engineer. Both work for CNN Internet Technologies, the organization that runs over 60 web sites for CNN and Turner Broadcasting.

References

[1] Burgess, M., ''Cfengine-Reference version 2.1.3,'' Centre of Science and Technology, Faculty of Engineering, Oslo College, Norway, Feb 2004.

[2] Craig, W. and P. McNeal, ''Radmind: The Integration of Filesystem Integrity Checking with Filesystem Management'' *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII)*, p. 1, USENIX Association, 2003.

[3] Hagemark, B. and K. Zadeck, ''Site: A Language and System for Configuring Many Computers as One Computer Site,'' *Proceedings of the Workshop on Large Installation Systems Administration III*, p. 1, USENIX Association CA, 1989.

[4] Harrenstien, K., M. Stahl, and E. Feinler, ''DoD Internet Host Table Specification,'' *RFC 952*, October 1985.

[5] Osterlund, R., ''PIKT: Problem Informant/Killer Tool,'' *Proceedings of the Fourteenth Systems Administration Conference (LISA XIV)*, p. 147, USENIX Association, 2000.

[6] Roth, M., ''Preventing Wheel Reinvention: The psgconf System Configuration Framework,'' *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII)*, USENIX Association, Berkeley, p. 205, 2003.

[7] Terpstra, J. and J. Vernooij, *The Official Samba-3 HOWTO and Reference Guide*, Prentice Hall, 2003.

[8] Thorton, J., ''Prescriptions: A Language for Describing Software Configurations,'' Technical Report 94-18, Jun 1994.

[9] Traugott, S. and L. Brown, ''Why Order Matters: Turing Equivalence in Automated Systems

Administration,'' *Proceedings of the Sixteenth Systems Administration Conference (LISA XVI)*, USENIX Association, p. 99, 2002.

[10] Tridgell, A. and P. Mackerras, ''The Rsync Algorithm,'' Technical Report TR-CS-96-05, The Australian National University, June 1996.

[11] Venema, W., ''TCP WRAPPER: Network monitoring, access control, and booby traps,'' *Proceedings of the Third UNIX Security Symposium*, p. 85, September 1992.