Configuration Management at Massive Scale: System Design and Experience

William Enck, Patrick McDaniel
Pennsylvania State University
{enck,mcdaniel}@cse.psu.edu

Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel
AT&T Research
{sen,psebos}@research.att.com
sspoerel@att.com

Albert Greenberg
Microsoft Research
albert@microsoft.com

Sanjay Rao
Purdue University
sanjay@ecn.purdue.edu

William Aiello
University of British Columbia
aiello@cs.ubc.ca

Abstract

The development and maintenance of network device configurations is one of the central challenges faced by large network providers. Current network management systems fail to meet this challenge primarily because of their inability to adapt to rapidly evolving customer and provider-network needs, and because of mismatches between the conceptual models of the tools and the services they must support. In this paper, we present the PRESTO configuration management system that attempts to address these failings in a comprehensive and flexible way. Developed for and deployed over the last 4 years within a large ISP network, PRESTO constructs device-native configurations based on the composition of *configlets* representing different services or service options. Configlets are compiled by extracting and manipulating data from external systems as directed by the PRESTO configuration scripting and template language. We outline the configuration management needs of large-scale network providers, introduce the PRESTO system and configuration language, and demonstrate the use, workflows, and ultimately the platform's flexibility via an example of VPN service. We conclude by considering future work and reflect on the operators' experiences with PRESTO.

1 Introduction

Configuration management is among the largest cost-drivers in provider networks. Such costs are driven by the immense complexity of the supported services and infrastructure. For a large provider, thousands of enterprises with diverse services and configurations must be seamlessly and reliably connected across huge geographic areas and rapidly evolving networks. Moreover, the initial turn-up installation and subsequent support of a single customer may span many organizations and systems internal to the provider. The stakes for the supported enterprise are extremely high: an outage may result in loss of business, delays in "getting to revenue" since service turn up precedes revenue, failure to meet contractual obligations, or disruption of key organizational workflows.

Given the complexity and stakes, it may be surprising that the common configuration management practice involves pervasive manual work or ad hoc scripting. The reasons for this are multi-faceted. From a provider perspective, every customer is in some ways unique. Though service orders have a lot in common, many new installations made to realize those orders represent unique combinations of services and network configurations. Interactions with customer networks, stale, incomplete, or imperfect information, and service interactions make both turn-up as well as ongoing maintenance of configurations complex and error-prone processes. Moreover, the devices in the network and definition of services they support change at a dizzying rate. New firmware versions, customer requirements, or supported applications appear every day. Market demands further dictate that the time-to-market for new services is a critical driver of revenue: delays caused by tool configuration, extension, or development can mean the difference between profitability and loss. In short, there is an unserved need in provider networks for tools that address these complex and sometimes contradictory challenges while constructing service configurations.

The PRESTO configuration management system develops network device configurations from composed collections of *configlets* that define the services to be supported by the target device. Written in our general-purpose hybrid scripting and configuration template language, the PRESTO system extracts specific information from external systems and databases and transforms this information into complete device configurations as directed by the PRESTO compiler. Extensive interactions with diverse engineering teams charged with managing operational IP networks led us to the conclusion that, to gain wide buy-in and adoption, the PRESTO language must adhere closely to the complex and often low level configuration languages supported by network device vendors (e.g., for Cisco devices, the IOS command language). PRESTO empowers network designers, “power users” comfortable with native network device configuration languages, and automates the unambiguous translation of their design rules into precise network configurations. Specifically, the PRESTO system generates complete device-native configurations, which can then be downloaded into a device and archived by network operators.

In this paper, we present the motivation, design, and workflow of the the PRESTO configuration management system. We outline the challenges faced by a large network provider in installing and maintaining millions of diverse devices for thousands of customers and organizations, and reflect on the failures of past network management systems to address these needs. We outline the PRESTO workflow and configuration language and demonstrate its features through an extended example. Finally, we discuss future work and detail preliminary experiences in deploying services in operational networks.

To date, PRESTO has concentrated on developing “greenfield” configs–configuration of new routers or services in a new installation. Such an approach avoids the inherent complexity of dealing with post-turn-up manipulation of fielded configurations resulting from software updates, performance tuning, or other maintenance. PRESTO’s role in the long-term maintenance of routers, called “brownfield”configuration, is currently evolving. While the body of the following work focuses on greenfield use, we revisit this latter objective and the challenges therein in our concluding remarks.

The PRESTO system evolved out of decades of experience in network management. Configuration management is about more than just getting routing and filtering correct. It must meld together many different services that exhibit subtle interactions and dependencies. Therein lies the challenge of configuration management in a provider network—*How do we glue together many information sources of myriad organizations in real time to build a functioning device configuration?* Revisited in the following section and throughout, it is the lessons gleaned from our experiences in meeting that goal that drive the PRESTO design.

The remainder of the paper proceeds as follows: Section 2 discusses motivation and requirements; Section 3 overviews the PRESTO work flow; Section 4 describes the language extensions provided by PRESTO; Section 5 incorporates these language extensions into a usable system, recognizing that input data is rarely pristine; Section 6 describes our experience using PRESTO for a real application within an enterprise network; Section 7 discusses related work; and Section 8 concludes.

2 Configuration Automation

In this section, we discuss the need for automation by describing current best practices and their limitations. We then describe the challenges an automated configuration generation system must face in large provider networks.

A router configuration file[1] provides the detailed specification of the router’s configuration, which in turn determines the router’s behavior. In essence, the configuration file is a representation of the sequence of specific commands that if typed through the command line interface determine the wide set of interdependent options in router hardware and software configuration. In practice, this may represent thousands of lines of complex commands, per router. These configuration files are text artifacts – described in a device specific command language, with a device specific syntax in a human and machine readable format, in some cases in XML. It is worth noting that a plethora of network devices beyond routers, e.g. Ethernet switches and firewalls, rely on configuration files of this type. To some degree, this state of affairs reflects natural technology evolution and the marketplace – networks started (and often still start) small and therefore often gravitated toward manual or (ad hoc) scripted configuration.

Today’s configuration languages offer myriad complex options, typically described in precise low level device-specific languages, such as Cisco’s IOS command language [8]. While the learning curve for such languages might be steep and the cost of inadequate learning severe (small slipups may cause large network outages), these languages are in extremely wide use for the entire lifecycle of network management – starting with configuration, but encompassing all other aspects, including performance, fault and security management. The tack that the PRESTO system takes is to leverage these “native” languages, and empower the user of these languages to enforce the precise translation of design intent into detailed device configuration.

Our interactions with network designers revealed that using templates to describe design intent is essentially universal. That is, designers create parameterized chunks of design configurations to describe intent. Accordingly, PRESTO provides full and flexible support for template creation in the native device configuration language.

2.1 Need for Automation

Decades of experience in network management have taught us that manual configuration practices are limited in the following ways, i.e., *the configuration process is*:

• *costly, time-consuming, and unscalable:* There is a significant initial investment in the interpretation and documentation of network standards and device-specific interfaces in developing any new service or support for a device. The result of that investment is a "model" configuration document (sometimes termed an Engineering and Troubleshooting Guidelines (ETG) document) used by enablers[2] to manually configure each target device. Typically performed by a large network engineering organization and depending on the complexity of the service or device, this process can take many person months of effort to complete and is an expensive process. The subsequent manual application of the model configurations to customer networks is also costly—some large customers may have tens of thousands of network elements, and applying a new configuration to even a fraction of them verges on the intractable.

• *prone to misinterpretation and error:* Even under the best of circumstances, engineering guidelines will not be perfect. Because network designers cannot anticipate all possible target environments, the guidelines are necessarily ambiguous, sometimes imprecise, and often subject to multiple interpretations. Thus, different enablers may interpret the same rules differently or adopt different local conventions. Differences between interpretations can and often do result in configuration mismatch errors. Making matters worse, while some errors might be easier to detect, others might have no immediate effect. These latter configuration problems are the most vexing, as they may become manifest at periods of high load (possibly the worst possible time) or introduce undetected security vulnerabilities.

• *fraught with ambiguous, incorrect, changing or unavailable input data:* Configuration information is not only spread across multiple data sources, but may be incomplete and imperfect. For example, customer order databases may not reflect the latest needs of the customer, e.g., order updates may only exist as emails to human contacts and may not quickly (or ever) be reflected in a database. As another example, information such as IP address assignments may be missing at the time of initial configuration. Finally, rules might be ambiguous. We have, for example, encountered examples where a particular service mandated that a site may have dual routers with ISDN backup, but it was not obvious which router must be the backup and which the primary.

2.2 Requirements

The pervasive practices and technical organizational problems detailed above makes automation in provider networks difficult. These issues lead to the following requirements, i.e., *the configuration process must*:

• *Support existing configuration languages:* While there have been many prior strong efforts at automating configuration generation, most have focused on developing abstract languages or associated formalisms to specify configurations in a vendor neutral fashions, e.g., IETF standard SNMP MIBs (Simple Network Management Protocol, Management Information Bases) [5], and the Common Information Model (CIM) [9]. These information models define and organize configuration semantics for networking/computing equipment and services in a manner not bound to a particular manufacturer or implementation. However, such generalized abstractions invariably introduce layers of interpretation between the specification and device. Gaps open between general abstract specifications and the concrete language of specific device configuration. It is very difficult to avoid extending or creating specialized common models to describe the realities of today's rapidly evolving devices and services. The artifacts of efforts to create standards or libraries often lag the marketplace. In truth, network experts often do not have the time or inclination to understand such abstractions, and today nearly universally find that working within native configuration interfaces is much more efficient for initial installation and later maintenance.

• *Scale with options, combinations, and infrastructure:* Customer configurations are dependent on, in particular, selected service offerings, devices, firmware revisions, and local infrastructure. For example, consider a site connecting to a provider backbone. The seemingly simple customer order has many options—does the customer require multiple routers to connect to the backbone or just one? Should each have multiple links or one? Further, each router may have several WAN (Wide Area Network) and LAN (Local Area Network) facing interfaces, and each interface may admit specific naming conventions that depend on the router model and the WAN card. The physical local infrastructure (e.g., routers and network topologies), will often have major impact on the workflow and content of the configuration.

• *Support heterogeneous and diverse data sources:* Putting together a router configuration involves collecting all necessary router configuration information. Such configuration information may not be all available in one central repository at the time the information is needed, but rather maybe distributed amongst a variety of databases, which are populated by upstream workflows. Take, for example, the customer order database. The customer information may itself have arrived at different times, and may be split across various forms. In large operational networks, information regarding customer orders and the resulting router deployment and maintenance is potentially spread across various systems spanning many internal organizations. An automated configuration system has to be cognizant of the diversity of information sources (and quality of data). Importantly, these data sources typically have their own persis-

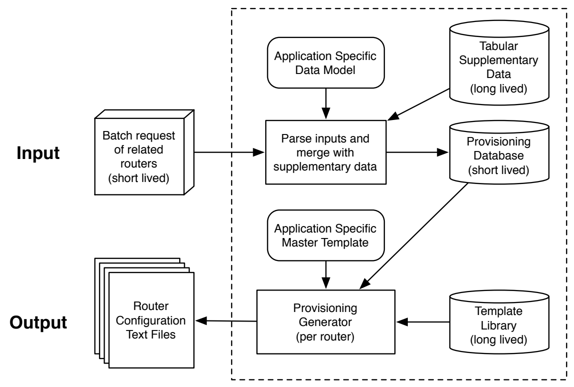
Input
Output
Batch request of related routers (short lived)
Application Specific Data Model
Tabular Supplementary Data (long lived)
Parse inputs and merge with supplementary data
Provisioning Database (short lived)
Application Specific Master Template
Router Configuration Text Files
Provisioning Generator (per router)
Template Library (long lived)

Figure 1: PRESTO Workflow

tent databases (each representing large investments), and a configuration management solution faces a huge real-world hurdle if it were to attempt to replace or replicate these databases, or even add a new persistent database rather than extend an already existing one. To the greatest extent possible, a configuration management system should strive to be stateless if it is to succeed in diverse operational environments.

3 PRESTO Overview

Two kinds of users interact with PRESTO—domain experts and provisioners. Domain experts initially codify the configuration services and options in *active templates*. These are the PRESTO equivalent of the ETG, where execution of the "active" template by the compiler directs the interpretation of the service definition for a particular environment. At installation time, provisioners obtain necessary configuration information from customers and other sources, that, when combined with the templates written by the domain expert, produce the end router configuration.

A more complete description of the PRESTO workflow is presented in Figure 1. In practice, deployment is a multi-stage process including requirements reconnaissance, initial staging configuration creation, and device turn-up. PRESTO is concerned with the second part of the process, the creation of the configuration. The configuration creation process begins with a batch of new configuration requests resulting from a network upgrade request or customer order. The requests are submitted to PRESTO as a collection of specification inputs, where the relevant environment data is provided as direct input or extracted from supplementary inventory and configuration databases. The data is cleansed and projected into a service or device (application-specific) data model created for the target configurations. Finally, an active template is executed for the set of specified target devices resulting in complete device-native configurations.

Note that these active templates, called *configlets*, may conditionally or deterministically import arbitrary other templates or extract data as needed at compile time to complete a configuration. It is this "composition by recursive import" and dynamic data extraction from external sources that allows PRESTO to deal with the potentially huge number of configuration options for a configuration.

PRESTO adopts automation to the extent possible, but allows for human intervention where needed. The process of mining and preparing configuration information is decoupled from the process of combining input data with configlets to produce configurations. Hence, as discussed further in Section 5.1, PRESTO can be viewed as a 2-step process, each of which is automated, but with manual intervention in between the steps. The first step involves tools that can, in an automated fashion, parse disparate databases to extract necessary configuration information. The information so produced may potentially be inspected by an enabler, overridden if necessary, and augmented with manually supplied missing information. The cleansed information is then the input to the second stage of the automated processing to produce the configuration.

4 Active Template Language

The PRESTO template language lays out the foundation over which the rest of the system is built. PRESTO does not define a new language, but rather a language *extension*. This allows domain experts to leverage knowledge of flexible native configuration languages, e.g., Cisco IOS, while creating useful abstractions, defining services, which take the form of *configlets* in PRESTO. Interestingly, this is in direct opposition to traditional network management interfaces which provide a single abstraction to which any policy would have to adhere. PRESTO provides the following key characteristics:

• *Data Modeling:* The data used to configure a network device is derived from a complex calculus of data from many diverse sources. Hence, dealing with this requires some means of organizing and accessing the data. We use the most natural choice for this task: a relational database. The schema of that database, called a *model*, is specific to the services and devices to which it applies, i.e., each collection of configlets works in tandem with a model defined by the domain experts.

• *Rich Template language:* A straightforward view of templates is that this merely involves direct substitution of "variables" by user-supplied inputs. For example, an architect may insert a variable for IP address information, that is then supplied by inputs from the user.

However, more complex operations and data manipulations are needed. For example, based on whether or not a router has a particular feature, the configuration may need to omit or include executable script code. Again, a device may have one or more interfaces, and configuration blocks may need to be created for each (of possibly many) interfaces. Moreover, the number of interfaces to be configured may only be available at run-time. In short, more sophisticated constructs than purely variable substitution are needed, e.g., variable expressions, conditionals, and loops.

• *Support for Template Decomposition and Assembly:* PRESTO provides support for the architect to write multiple smaller templates targeted for very specific elements of a configuration, i.e., services. There are several advantages to such an approach. First, supporting multiple smaller templates simplifies creation and maintenance. Second, this allows for templates to be written by multiple designers based on their expertise. This is analogous to programming modularity, where each programmer or group can be develop and maintain a part of the larger system. In the case of PRESTO, for example, a designer may better understand how to deal with various WAN interfaces, while another may better understand issues with BGP configuration. Third, breaking templates down promotes reusability, as there is the potential to create template libraries, and reuse them across multiple applications.

Before discussing specific details, we provide an example scenario to aid in the understanding of language constructs and design motivations. Consider the configuration of a gateway router. The gateway connection may have one or more external connections. If there are multiple connections, they may be dispersed across multiple routers. Hence, these routers require configuration knowledge of the other routers participating in the gateway connection, e.g., IP addresses, to coordinate failover, e.g., HSRP [17]. The template language must support these relationships between connections on one router and between routers.

We now discuss each part of the template language in turn. After discussing the core language concepts, we introduce additional language features that enable better software engineering practices.

4.1 Data Model

The PRESTO template language revolves around the data model. The templates require access to small data chunks describing router properties. Furthermore, multi-router relationships dictate a need to perform quick lookups for peer specific information. Such a capability is required for instance when configuring one router (eg., a spoke) involves extracting information for another router (eg., the hub). A relational database provides just this capability: router properties are stored in table fields and accessed as variables; peer router properties are queried by specifying the router hostname. Hence, the data model becomes a database schema. We refer to this database as the provisioning database and provisioning relational database schema where necessary to remove ambiguity.

The schema definition is application dependent. Each application has different requirements on data accessibility. It is no surprise that defining the schema is the most delicate part of applying PRESTO to a new application, hence complete flexibility is required. Despite the supported flexibility, past experience has resulted in a few recommended guidelines. The data model should contain a `ROUTER` table indexed by a globally unique identifying value, e.g., the router hostname. One row will exist for each router in a provisioning request. The `ROUTER` table should contain the bulk properties; however, whenever multiple instances of a property occur, a new table should be created. For example, multiple LAN or WAN interfaces are semantically equivalent. Sub-router tables should use a multi-column primary key consisting of the `ROUTER` table index and a unique identifier, e.g., interface number. Upon querying the database for all LAN interface records matching a specific router hostname, the template language produces an interface loop. For example; suppose `LAN` is a table holding all LAN-facing inputs. Then

```
SELECT * FROM LAN WHERE ROUTER=THIS_ROUTER
```

selects each LAN-facing interface on a given router. Iteration specifics and syntax are discussed below.

4.2 Variable Evaluation

Variable substitution is integral to any template language; PRESTO is no exception. Variables are defined by the data model. Templates gain access to variables by querying the provisioning database. The returned record defines a variable namespace, or *context*[3], used to access the variable, e.g.:

```
<CONTEXT.VARIABLE>
```

Variables of this form are directly substituted in the template text.

Templates are written to produce a configuration file for one router. PRESTO begins template evaluation by querying the `ROUTER` table in the data model for the row corresponding to the current router. The returned record populates the `ROUTER` context, which consequently allows templates to use `ROUTER` variables at any point. The template creates new contexts by making a new database query; however, those variables are only accessible within the defined context scope. When a query returns multiple records, the template code within the context is repeated, producing a loop. For example, SELECT * FROM LAN WHERE ROUTER=THIS_ROUTER has the effect of configuring multiple LAN-facing interfaces.

4.3 Iteration

The PRESTO template language simulates iteration by executing database queries that return multiple records. The template designer creates a data driven loop by defining a new context name, an SQL-like query, and a scope. Each row returned by the query produces an iteration. For example:

```
[INT:SELECT (*) FROM (WAN_INTERFACE) WHERE
 (WAN_INTERFACE.HOSTNAME=<ROUTER.HOSTNAME>)]
interface serial0/<INT.SLOT>/<INT.PORT>
 bandwidth <INT.BANDWIDTH>
 ip address <INT.IP> <INT.MASK>
!
[/INT]
```

Here, `INT` is the name of the new context. The statement associates the `INT` context with the record returned by querying the `WAN_INTERFACE` table of the provisioning database for all fields (`(*)`) related to the current router hostname (note the use of `<ROUTER.HOSTNAME>` in the query). The text within the `INT` context scope, i.e., all text between the query statement and the context closing statement, `[/INT]`, is repeated for each returned record. Field names from each record are accessible as variables within the context, as shown. Note that new context definitions can be arbitrarily nested, but they cannot define scopes spanning multiple parent scopes. That is, the nested context's closing statement must occur before its parent closing statement. This constraint is consistent with loop structures in common programming languages.

4.4 Conditional Logic

Configuration statements are commonly dependent on router properties. For example, E1 (a standard widely used in Europe) line cards required slightly different interface specification than T1 (a standard widely used in the US) cards. The PRESTO template language supports the inclusion and omission of configuration options with conditional statements. All conditionals have a label, condition and scope, in general:

```
[COND LABEL CONDITION]
... template text
[/LABEL]
```

`COND` indicates a conditional statement; `LABEL` defines a label; and `CONDITION` contains relational operators that dictate if the template text between the condition statement and the closing statement, `[/LABEL]`, is included. The template text can contain static strings, new contexts, or even more conditionals. The `CONDITION` itself supports arbitrary complexity of Boolean logic. Statements can be simple:

```
("<ROUTER.HAS_FEATURE_X>" eq "YES")
```

or more complex logic:

```
(("<ROUTER.HAS_FEATURE_X>" eq "YES") &&
(("<ROUTER.HAS_FEATURE_Y>" eq "YES") ||
  ("<ROUTER.FEATURE_Z>" ne "BASIC")))
```

4.5 Data Transformation

Configuration statements commonly require a transformation of an input variable. For example, an interface IP address may be specified as IP and mask, i.e., *x.x.x.x/y*, but the router configuration language requires the IP and mask coded separately, i.e., *x.x.x.x z.z.z.z*. In another case, the template designer may need to configure the network address corresponding to the input value. To accommodate such requirements, the PRESTO template language provides a mechanism for arbitrary extension.

A function added to the language interpreter module is referenced within a template as a context, variable, function, and argument:

```
<CONTEXT.NEW_VARIABLE:function(args)>
```

Upon execution, arguments are evaluated (if they are variables) and passed to the function. The function performs a specific manipulation and returns the result to a new variable in the specified context. The new variable's value is inserted into the template text, and it's value is retained for later use within the context.

To aid template design, the PRESTO template language contains a core set of application agnostic functions. Some functions provide generic computation abilities, e.g., `calc()` performs simple arithmetic, `sbsstr()` returns a substring specified by an offset and length, and `matchre()` provides regular expression substring matching. Other core utility functions perform useful conversions on common network values such as IP address. For example:

```
<INT.NETIP:computeOffsetMaskIP(<INT.IP>,0)>
```

computes the network address of an IP specified in *x.x.x.x/y* form. The function, however, can calculate any offset of the IP, a useful feature when network policy dictates devices on specific offsets, e.g., the gateway is commonly `.1`. Realizing a new PRESTO function involves including its code in the PRESTO language interpreter.

4.6 Hidden Evaluation

Configuration policy occasionally requires values resulting from complex computations. While additional domain specific functions provide ample mechanism, template designers are encouraged to keep domain knowledge within the templates themselves. The motivation is twofold. First, this reduces bloat of the core language. Second, as function definitions require programming, and most template designers do not possess the necessary skills, or are simply unwilling, to create new functions. Therefore, we have added only a small number of generic primitive operations to the core language in an application.

As described to this point, the template language is not conducive to performing complex computation within the templates themselves. All functions return text that is inserted into the end router configuration. Multi-step computations therefore become difficult, if not impossible. To overcome this issue, the language supports hidden evaluation:

```
[EVAL LABEL noprint]
... template statements
[/LABEL]
```

Statements within the `LABEL` scope produce no output.

Computation within `EVAL` blocks is not limited to simple multi-step functional transformations. In practice, we leveraged the hidden evaluation interface to provided a multitude of features. For example, database `SELECT` queries were used to lookup values in supplemental data tables. Values were assigned to higher level contexts, e.g., `ROUTER`, and used throughout the template. The `EVAL` blocks also proved useful to determine values that depended on multiple conditionals. The conditional logic was performed once, and the value was used many times thereafter.

4.7 Template Assembly

Managing one large template becomes unwieldy. Software engineering experience recommends modular code. Templates are no exception. Using many small templates, or `configlets` provides many beneficial side effects. It allows a template designer to concentrate on one feature at a time. For example, a configlet can be written for each network access type used for the WAN interface of a router. Later, depending on the router provisioning data, the correct configlet is chosen. By including configlets on demand, complicated conditional logic is avoided. Additionally, as configlets are only inserted where applicable, they can be written with certain assumptions in mind. This reduces complexity within the configlets themselves. Finally, as configlets can include other configlets, the template designer can exploit commonality between configlets.

PRESTO stores all configlets in a template library. Configlets can be included at any point. The language provides a special syntax for including configlets:

```
[INCLUDE FROM (FEATURE) WHERE
  (FEATURE.TYPE=SOME_TYPE)]
```

In our above example, the correct WAN interface configlet is included using the `<ROUTER.ACCESS_TYPE>` variable:

```
[INCLUDE FROM (WAN) WHERE
  (WAN.ACCESS_TYPE=<ROUTER.ACCESS_TYPE>)]
```

4.8 Example Configlet

Once the data model and configlet organization are determined, writing the configlets themselves is straightforward. We now provide a quick example to show how each of the language primitives come together. Consider a network topology where the Internet edge has two access lines, each connected to one router, and the two routers establish load sharing of in and outbound traffic. The most complex configlet will define the WAN routing protocol. Figure 2 provides an example.

The example begins by including the interface configlet defining the back to back connection between the two routers. Multiple connection types may be supported. Instead of using a large conditional statement to pick the right interface definition, the `INCLUDE` statement allows the relational database to perform the conditional logic and simplify the code the domain expert must specify.

The BGP block defines the WAN routing protocol configuration. Here, a `SELECT` queries for network addresses specific to the peer router. The configlet also performs a domain specific sanity check that warns the enabler if the two routers' back to back IP address are in different networks. This check is placed within and `EVAL` block to keep warning text from leaking into the end configuration. Finally, the `SELECT` query is also used to determine information specific to the WAN connection. Here, the data model specifies that WAN interface specifics be placed in a separate table. The configlet selects the correct table row using the `HOSTNAME` foreign key. The Cisco IOS `network` and `neighbor` commands require a translation of the available information, therefore the `computeOffsetMaskIP()` function is used. In this case, the remote peer is always the second IP in the network, therefore the domain expert is able to code the neighbor's IP directly using the offset function.

```
%% Configure Back-to-Back Interface to Peer Router
[INCLUDE FROM (B2B) WHERE (B2B.TYPE=<ROUTER.B2B_TYPE>)]
%% Define the BGP configuration
router bgp <ROUTER.LOCAL_ASN>
 network <ROUTER.LOOPBACK0> mask 255.255.255.255
[PEER:SELECT FROM (ROUTER) WHERE (ROUTER.HOSTNAME=<ROUTER.PEER>)]
%% Ensure the peer is in the same network
[EVAL B2B_CHECK noprint]
[COND WRONGNET ("<ROUTER.B2BNET:computeOffsetMaskIP(<ROUTER.B2B_IP>,0)>" \
               ne "<PEER.B2BNET:computeOffsetMaskIP(<PEER.B2B_IP>,0)>" ) ]
<ROUTER.WARNING:templateWarning(<ROUTER.HOSTNAME> and <PEER.HOSTNAME> different B2B Net)>
[/WRONGNET]
[/B2B_CHECK]
 network <PEER.NETIP:computeOffsetMaskIP(<PEER.B2B_IP>,0)> mask 255.255.255.252
 neighbor <PEER.B2B_IP> remote-as <ROUTER.LOCAL_ASN>
 neighbor <PEER.B2B_IP> next-hop-self
[/PEER]
[WAN:SELECT FROM (WAN) WHERE (WAN.HOSTNAME=<ROUTER.HOSTNAME>)]
 network <WAN.NETIP:computeOffsetMaskIP(<WAN.IP>,0)> mask <WAN.MASK:computeMask(<WAN.IP>)>
%% The gateway is the second IP in the subnet (for this example)
 neighbor <WAN.GW:computeOffsetMaskIP(<WAN.IP>,1)> remote-as <WAN.REMOTE_ASN>
[/WAN]
 no auto-summary
!
```

Figure 2: Example configlet defining the WAN routing protocol of a two-line two-router configuration

5 The PRESTO System

The template language and data model provides a mechanism to describe configuration policy; however, it must be incorporated into a usable system. In an ideal world, an engineer receives a request for a group of related routers with all required input information available and correct. This would allow a straightforward application of the data model and templates to act upon inputs. As shown in Figure 1, the per-request input data is parsed and merged with tabular supplementary rules to create a one time database. Then, for each router in the request, a master active template is executed by the provisioning generator. This master template includes and executes appropriate configlets according to the input data, resulting in a completed configuration text file.

Unfortunately, the information required to configure a router is not always readily available. In large operational networks, the input data for the configuration task spans the outputs of multiple upstream workflows, which may arrive at different points in time. It is therefore important to be able to work with such partial information flows and to able to handle any inconsistencies across the flows. In such a scenario, ubiquitous flow-through or full automation becomes extremely difficult to realize. Accordingly, PRESTO provides hooks to overcome these difficulties, when and where they arise.

5.1 PRESTO Architecture

PRESTO achieves nearly full automation using a 2-step process, see Figure 3. The goal of automation is to minimize user actions. PRESTO minimizes manual processes in two ways. First, it handles bulk requests. This streamlines the process of creating the initial router configuration code. Second, it requires only one point of user interaction at which point users provide the most minimal effort to allow automation to complete. In PRESTO, data processing proceeds in two steps, with user integration capabilities made available between step 1 and step 2.

Specifically, PRESTO uses a 2-step architecture to request user input at the most ideal moment. The process begins with the submission of a batch request to step 1. Step 1 pulls together and parses information from available input data sources for the batch request. The output of step 1 is the complete and unambiguous information needed to configure all routers in the batch. The role of step 1 then is to normalize and tabulate the input information and, if possible, apply defaults or inference rules to fill in missing information. Where defaults or inference are applied, the step 1 output will flag or annotate the output, for (optional or mandatory) user inspection. In practice, we found that inference rules include many types of calculations, for example, ranging from assigning incremental BGP AS prepend values to selecting interface ports for network connections. The result is presented to the user for validation in tabular form. The user is then given the option to change certain of the data to meet customer requirements (which we found sometimes change faster than the ordering information systems can be updated), e.g., changing interface cards, and the batch request is submitted to step 2. Step 2 executes the PRESTO engine described in Figure 1, combining complete input information with templated policy information as described above, to produce the configuration file for each router in the batch request. By dividing the

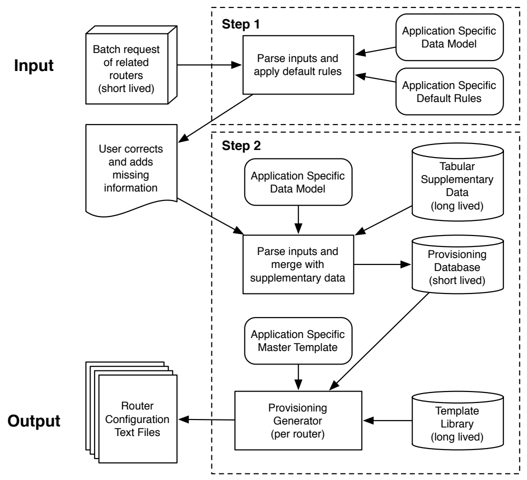
Input
Batch request of related routers (short lived)
Step 1
Parse inputs and apply default rules
Application Specific Data Model
Application Specific Default Rules
User corrects and adds missing information
Step 2
Application Specific Data Model
Tabular Supplementary Data (long lived)
Parse inputs and merge with supplementary data
Provisioning Database (short lived)
Application Specific Master Template
Output
Router Configuration Text Files
Provisioning Generator (per router)
Template Library (long lived)

Figure 3: 2-step PRESTO Data Flow

processing in these two steps, we isolate fallout due to inadequate information to step 1, and provide the system and its users opportunities to repair fallout and resume automation before proceeding to step 2, which then produces the desired output.

PRESTO was designed to be input agnostic, and has been adapted to be invoked in a number of different modalities (via web services, via database invocations, or via stand-alone interfaces), and to operate on inputs of various forms and origins (database extracts, Excel spreadsheets, XML forms). PRESTO is essentially stateless. By design, the persistent data in PRESTO is limited to a a repository of configuration policies; the persistent database of associated router configurations is external to PRESTO.

Accordingly, PRESTO applications are built on a *provisioning data model*, describing short-lived data, and a *policy data model*, describing long lived data (see Figure 3). We do not have space here to describe the details or the precise representations of these data models, and so shall describe salient features. Short lived inputs are limited to those specific to a given batch request and contain information used to populate the active templates. Long lived inputs contain configuration policy. These inputs exist in a variety of forms, including the default rules and data models used in both steps 1 and 2. The policy data model captures rarely changing information, such as the number of ports on an line access card, as well as stable configuration parameters, e.g., domain wide network access lists. The policy data model also encompasses the library of templates for configlets and whole configurations. As noted above, the templates describe the comprehensive configuration policy logic in the native device configuration language. Typically, new templates are released after significant scrutiny and test on a release schedule (albeit at a rapid pace), accompanied in parallel with customer or user notification – as template change leads to change in network and service behavior.

All data models eventually require maintenance, including the policy data model. PRESTO simplifies maintaining tabular supplementary data on hardware configuration rules, and PRESTO language templates by storing both in a database. We found that tabulating hardware configuration data only allows for easy additions of new hardware options, but it also allows non-technical domain experts to update tables by maintaining and submitting corresponding spreadsheets. As noted in Section 4, the active templates are also broken down into configlets (sub-templates). The configlet concept allows logical components to be easily updated without affecting the entire library.

5.2 Validating Step-2 Inputs

Step 1 cleans up and normalizes short lived input data. PRESTO provides an interface between steps 1 and 2 to allows users to update and validate values to ensure all required data is available and correct. Users, however, can make mistakes. Therefore, step 2 must perform sanity checks; both syntactic and semantic checks are required. Syntax checks occur in the parsing phase and test for data formats, e.g., an IP addresses are valid "dotted quads," of the form *x.x.x.x/y*. Factoring parsing checks to the parsing phase of step 2 simplifies the system and improves template readability – as the templates are written under the assumption that domain agnostic inputs such as IP addresses are syntactically correct. Semantic checks are more implemented naturally in the configlets within the PRESTO language, as these checks are domain or application specific, e.g., a check may assure that an IP address is neither a network or broadcast address. To support these various forms of checks, the PRESTO language provides capabilities to emit errors and warnings, via functions `templateError()` and `templateWarning()`, each taking a string argument.

5.3 Handling Unknown Information

We found warning messages to be of remarkable utility, as PRESTO users insisted that the process of generation router configuration files proceed in spite of missing information. Users often preferred to obtain configurations with warnings that some information might be missing or inferred, rather than obtaining just an error message. As this eventuality may appear surprising given PRESTO's two step architecture, some explanation may be in order. Routers for a given project or customer are not always ordered or provisioned at the same time. As routers are related to one another through their configuration, situations sometimes arise where information needed to configure one set of routers may depend on the ordering and configuration of a second set of routers, and the information flow for this second set may be missing at the time that PRESTO is invoked to provision the first set. In other scenarios, essential parameters such as IP addresses which must be written into the configuration may be unknown at the time of initial configuration generation.

PRESTO accommodates these scenarios by allowing the active templates to perform a sort of due diligence. Active templates perform conditional checks to see if all non-mandatory inputs are available. To allow data availability tests of values that drive iteration or looping constructs, the PRESTO template language was extended to support query checking by allowing the standard SQL syntax `count(*) as COUNT`. Using such a query, the new context has access to a `COUNT` variable, on which conditional logic is performed, allowing for the detection of missing information. Where these non-mandatory inputs are missing, configuration lines are still generated with dummy or inferred values, but language specific comments ensure the produced code is still of some utility (e.g., the router will boot and provide basic connectivity), leaving to automation downstream of PRESTO to complete the configuration task.

5.4 Implementation

The core PRESTO system was implemented in approximately 3,000 lines of Perl code. The code is divided into two modules, `PROVGEN.pm` and `PROVDB.pm`, which implement the language interpreter and database interface, respectively. To accommodate a new application, we found the application specific adapters to deal with step 1 inputs can easily grow to thousands of lines. Fortunately, however, many identical components or parsing patterns can be easily ported from one application to another.

6 Experience with Real Services

The development of the PRESTO system has benefited from the insights of network designers and engineers responsible for configuring network elements for large commercial connectivity services. A key measure of the value of such a tool ultimately is how useful and usable it is in practice for this target user community. The PRESTO system is currently being used to automate configuration generation for a number of different commercial network services. In addition to the clear pressing benefits of a successful configuration tool for network management, such an exercise is important for the following reasons:

- It helps us understand the type and amount of effort needed in the process of taking the PRESTO tool and customizing it for a new service.
- It allows us to evaluate and if required evolve the design of the tool in the context of a real world application and its particular requirements.
- It provides valuable feedback for improving the tool.

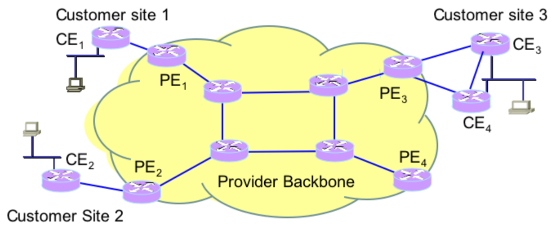

Figure 4: Provider VPN

In this section we present our experiences with, and lessons learned from automating configuration for provider-based VPNs via PRESTO. We first describe the service, then outline our experiences with customizing PRESTO for this service.

6.1 Customer edge router configuration for provider VPNs

Enterprise networks are increasingly using provider-based Virtual Private Networks (VPNs) to connect geographically disparate sites. In such a VPN, each customer site has one or more customer edge (CE) router connecting to a one or more provider edge (PE) routers in the provider network (see Figure 4). Incoming customer traffic from a CE is encapsulated at the PE and carried across the provider network, decapsulated by a remote provider edge router and handed off to the customer CE router at a remote site of the same customer. Traffic belonging to different customers is segregated in the provider backbone, and the provider network is opaque to the customer. The predominant method for supporting provider-based VPNs uses MPLS as the encapsulating technology across the provider backbone.

From a provider perspective, a critical part of supporting the VPN service involves configuring the the CE routers. The tasks include configuring ACLs, interfaces, WAN (Wide Area Network) and LAN (Local Area Network) links and routing (e.g., OSPF and BGP) . The key challenges pertain to heterogeneity and scale. VPN services enjoy a large and growing number of customers. A single customer can have hundreds to thousands of different sites. There are a wide range of features and options a customer can select that impact the configuration.

There are different hardware elements (router models, interface cards), different access type options for the CE-PE connection (e.g., Frame Relay and ATM), different interface speed options, and so on. Customers can opt for a range of resiliency options for each of their sites, where the resiliency option determines the number of routers at the site, the number of distinct links from the site towards the provider network, whether there is a last-resort dial backup to be used if the data service fails, etc. An example resiliency option is 2 CEs with 2 different links to the provider network running in load sharing mode. In addition to being already large, the features and options also change as the service offering improves and evolves. For instance, newer router models and line cards are being constantly added to the list of supported options.

The CE configuration task embodies many of the challenges outlined in Section 2. VPN services involve fast growing demand, a large and increasing volume of router orders to be configured, a huge space of feature combinations, and the need to support a steadily increasing slew of new features. All these factors make the overheads with a manual-intensive configuration workflow unacceptably high, and make these services prime candidates for PRESTO automation.

6.2 A PRESTO tool for CE configuration

Developing a PRESTO-based CE configuration tool involved knowledge engineering (codifying expert knowledge, initially only partially documented) and data modeling (identifying service-specific information and business rules), as well as front end user interface and back end PRESTO template development. The main tasks involved were:

- identifying all the service-specific information required for building the CE configs, and developing the resulting VPN-specific configuration data model.
- understanding the workflow surrounding the provisioning process and available data sources and determining how the information in the above data model can be extracted.
- collating the service-specific provisioning rules and building the service-specific templates based on engineering guidelines from the service designers.
- defining the workflow of the PRESTO-based tool and developing service-specific code around the core service-agnostic PRESTO system.

Recall that PRESTO requires an application to define two types of data models—a provisioning data model for short lived data, and a policy data model for supplemental data and templates. The provisioning data model provides a normalized repository of data specific to the current router request set. The VPN instantiation of the provisioning data model placed as many fields as possible in a central `ROUTER` table indexed by the router hostname. This contained router specific information such as model number, software version, and available customer information. Whenever multiple instances of any type of data was required to build template iterations, a new table was created – that is the data model was highly normalized, as replication has risk in provisioning tasks. For VPN, this led to the creation of tables for WAN interfaces, dial backup information, and the logical interfaces that define VPN connections. In total, the resulting provisioning data model consisted of one main `ROUTER` table and nine secondary tables, each containing foreign keys to reference the `ROUTER` table.

The longer lived policy data model was split into supplemental data and template data sub-models. Supplemental data consisted mainly of data already naturally expressed in tabular form, e.g., mappings from card names to interface type and number of ports, and mappings from strings describing interface speeds to the actual value to code in the configuration. The template data model proved much more interesting, as it contained the configlets used to create the actual router configuration. Configlets were grouped by logical features, specifically `BASE`, `LAN`, `WAN`, `RESILIENCY`, `DAC`, and `B2B`. as follows. The `BASE` table consisted of the configlet required for all routers, e.g., hostname, password, loopback, and `motd` commands. The `LAN` and `WAN` tables contained configlets for types of interfaces, e.g., frame relay and ATM interfaces. The `RESILIENCY` table contained configlets defining the different resiliency options required by the VPN service. The `DAC` table contained configlets specific to various parts of the dial backup configuration. The `B2B` table defined special interface definitions used where CE routers are organized in back to back configurations. Defining these new feature tables as primitives or building blocks allowed specialized configlets to be easily composed and promoted knowledge and code reuse. A total of 44 configlets containing 5414 lines of statements were created.

6.3 Benefits and Experiences

Various existing processes such as accounting, customer service, provisioning, and network management interact with configuration management. For the PRESTO tool to be practically usable, it was critical for it to operate within the confines of the surrounding existing configuration management processes and systems. This requirement to operate in pre-existing existing system and tool environments significantly impacted the PRESTO design, and the extent to which the configuration generation could be automated. Indeed, the 2-step PRESTO architecture and the accommodations for potential human intervention/oversight between the two Steps were key design elements that resulted from this requirement.

We next revisit the requirements criteria introduced in Section 2.2 and discuss to what extent the PRESTO realization achieves those.

• *Support existing configuration languages:* The PRESTO template language achieved this goal completely. A PRESTO template consists of active template code and configuration statements in an existing configuration language. The template language was used to extract the particular configuration context, determine the control of flow in a configlet, specify rules for combining configlets, and achieve variable substitution and functional substitution. However, the actual configlet was specified in the configuration language that network engineers are familiar with, e.g., Cicso CLI (Figure 2).

• *Scale with options, combinations, and infrastructure:* Several aspects of the PRESTO design made it possible to write a configlet once and reuse it for many different configuration scenarios. These included the capability to dynamically extract data as needed, the approach of writing small configlets for specific features of a configuration, and the support provided for combining configlets together deterministically or conditionally. Reuse of configlets was critical in ensuring that the authoring of templates for the VPN service was tractable, despite the large number of feature combinations in this service. Our experience demonstrated that the effort required to author templates the first time was acceptable – any significant lags were attributable to legitimate debate on the nuances of the precise design intent and associated router capabilities. The incremental effort in updating the system to handle new features was also low. Any updates to the supplementary data such as a new interface card were realized by simply updating the relevant table in the database, without any additional coding effort. For supporting a new router model, we were able to reuse all the existing templates for common features, and only needed to write templates for features that were not yet covered (e.g., a new interface type) or that were router model specific (e.g., rules for numbering interfaces). In fact, the ability to reuse existing templates has proven to be a key strength of the PRESTO approach.

• *Support heterogeneous and diverse data sources:* A key task involved modeling the information needed for service configuration and determining how that information could be obtained from existing data sources. For the VPN service, there were multiple sources of input information: (i) a customer order document that contained details about a customers request for the service, such as a list of sites, and the selection of choices for that site (as listed above, the choices included the number of routers, router models, number of access links, access types, and resiliency option); (ii) other documents that listed required auxiliary information, e.g., a list of the supported router models and cards and the type and number of interfaces on them, (iii) engineering policy documents that specified the configuration rules for all combinations of customer order options. While a large subset of the required information could be directly gleaned from the various data sources described before, there were other important information pieces that for different reasons could not be pulled automatically from an external source. Some of this information required application of service-specific business rules and computations to available input data, while other information needed to be manually assigned. We found that the 2-step PRESTO architecture (see Section 5.1) was well suited to handle this data-heterogeneous environment. In addition to getting available information from a variety of input sources, Step 1 marshalled additional required values and choices in the data model by applying service-specific rules to the available input information. The resulting partially populated provisioning data model was exposed to the user at the end of Step 1 for validation and for filling in missing values. Step 2 of the system then successfully drove the task of actually creating the CE configurations.

One measure of the benefit an automation tool is the reduction in the amount of human mediated effort. While an exact apple to apple comparison is not easy, we compared the traditional VPN CE configuration workflow to the PRESTO workflow. In the traditional manual configuration workflow, engineers proceeded through a manual time consuming process where they collated the different data sources, ran complex computations to derive additional necessary information, and then navigated the complex options in the customer order to create the router configuration. In a customer order with many sites and routers, the process had to be repeated many times, once for each router. If dependencies existed between multiple routers, engineers had also to be careful to reflect the dependencies and build consistent configurations. For instance, if 2 routers had a connection between them, the configuration of the interfaces on both sides should be consistent. In contrast, the PRESTO tool for CE configuration has a significantly more automated workflow, where:

1. The configuration engineer uploaded the customer order, and begins executing the first step of PRESTO.

2. PRESTO created a document at the end of the first step that contained for each router in the customer

order a list of all the information fields needed for configuring the router. Of the total of 64 relevant fields, about 42 were auto-populated with values parsed directly from the various input sources, and another 14 were auto-populated by applying default engineering rules coded into PRESTO. A small number of fields (around eight), had to be manually filled in.

3. The engineer reviewed this document, filled in the missing field values, overrode auto-populated values if required, and initiated the execution of the second step of PRESTO.

4. The tool then created and returned the configurations for all the routers in the customer order.

The manual effort with PRESTO was reduced to filling in a small number of values per router, reviewing auto-populated fields, and sometimes overriding them. Feedback from user trials indicated that engineers found the approach of auto-computing, and auto-populating field values based on default rules to be very useful, even though manual intervention was sometimes needed to override the values.

The PRESTO CE configuration tool was put through user trials involving engineers from across the world. This helped uncover and normalize certain configuration rules that showed regional variations under the manual process, reinforcing the need for the PRESTO tool. One lesson from the user trials was that the system must not only create correct configurations, but also must support a streamlined and sparse user interface. Though we do not describe the details here, designing a suitable user interface proved non-trivial, and required several iterations before it passed the acceptance threshold of users.

7 Related Work

Several industrial products (for example, [6, 16, 20, 21, 7]) have emerged that offer support for configuration management. Many of these efforts have focused on developing abstract languages to specify configurations in a vendor neutral fashion, e.g., IETF standard SNMP5D MIBs [5], and the Common Information Model (CIM) [9]. These information models define and organize configuration semantics for networking/computing equipment and services in a manner not bound to a particular manufacturer or implementation. An example of the success of such an approach is the DSL Forum's TR-069 effort for DSL router configuration [10]. Yet, general router configuration via this approach is challenging given rapid technology evolution, forces driving network operators and vendors towards competitive and differentiated advantage, feature proliferation, and the need to continuously expand networks and features while maintaining backwards compatibility.

Boehm et. al. [3] present a system that raises the abstraction level at which routing policies are specified from individual BGP statements to a network-wide routing policy. The system includes support to generate the appropriate pieces of router configuration for all routers in the network. An approach to automated provisioning of BGP-speaking customers is discussed in [15]. These efforts focus on BGP, just one component of router configuration. Narain [18] seeks to bridge the gap between end-to-end network service requirements, and detailed component configurations, by formal specification of network service requirements. Such specification could aid synthesis of router configurations. In contrast to these efforts, our focus in PRESTO is on the synthesis of complete, precise, and diverse network configurations that are readily deployable.

Several initiatives have explored configuration management systems for desktop, and server environments [1, 4, 2, 12]. Networked and router environments often involve more complex options and interdependencies than desktop environments, and these solutions do not directly apply. That said, there is much potential benefit from cross-fertilization between these domains. Further, many of these works have emphasized deployment of configurations, and have placed relatively little effort on deciding what the configuration of a node should be [2].

While the focus of PRESTO is the synthesis of configuration files, others have looked at important orthogonal issues related to configuration management. The Network Configuration Protocol (NETCONF) [19, 11] effort provides mechanisms to install, manipulate, and delete the configuration of network devices. The NESTOR project [22] seeks to simplify configuration management tasks which requires changes in multiple interdependent elements at different network layers by avoiding inconsistent configuration states among elements, and facilitating undo of configuration changes to recover an operational state. Others [13, 14] have looked at detailed modeling and detection of errors in deployed configurations.

8 Conclusions

The PRESTO system presented throughout represents a step toward realistic automation of massive scale configuration management. While its genesis was mandated by the specific needs of a single provider, the approach and insights are universal. Central to the success of PRESTO are the satisfied mandates for the treatment of complex and evolving service definitions and customer requirements, dealing with the hugely diverse and sometimes unreliable data sources, and communication within the *lingua franca* of its user community.

PRESTO attempts to balance these requirements by providing malleable and composable *configlets* that encode configuration business logic directly in the target language. Integration of external data sources is performed by simple code embedded in templates and specialized database query interfaces. This approach provides network engineers with rigorous tools to clearly define the workflow and content of service configuration while maintaining the flexibility of enablers to manipulate the resulting configurations to suit the customer or environment in which they will be used. Additionally, configlets are not just applicable for routers; other devices with text-based configuration can also benefit.

Our experiences with PRESTO in the VPN and other services are promising. We learned much that led us to adjust the language and the way it is used in practice, but also confirmed that PRESTO approach is viable. However, we found one of the greatest challenges of automating router configuration at a massive scale is the ability to gather and reconcile necessary input data.

Our future work extends PRESTO in two key directions. First, we are deploying the tool in a wider range of services. Our goal here is to demonstrate that much of the PRESTO design is general, and new services supported with relatively little effort. The second major initiative is to evaluate PRESTO as a platform for "brownfield" configuration in full generality, where updates may be made by a set of systems and tools, with PRESTO among this set. Such tools tackle the more complex problem of projecting a configuration into a live system without negative consequences, e.g., causing performance, security, or connectivity problems. These efforts may require significantly more intelligence than the smart templates currently supported, and may require the introduction of techniques that reason about the consequences of configuration with respect to the services that are already present. It is through these efforts that providers will begin to ease the burden of costly and error-prone configuration management.

Notes

[1]The specification may in fact be split across more than one file, or modality of description of the command set

[2]*Enablers* are the personnel who implement a given service, either staff on-site or within a provider's Network Operations Center.

[3]Note, this use of *context* is different than in context-based evaluation of programming language literature.

References

[1] P. Anderson. Towards a high-level machine configuration system. In *Proc. of the 8th Large Installations Systems Administration (LISA) Conference*, 1994.

[2] P. Anderson and E. Smith. Configuration tools: Working together. In *Proc. of the Large Installations Systems Administration (LISA) Conference*, December 2005.

[3] H. Boehm, A. Feldmann, O. Maennel, C. Reiser, and R. Volk. Network-wide inter-domain routing policies: Design and realization. April 2005.

[4] M. Burgess. Cfengine: a site configuration engine. In *USENIX Computing systems, Vol 8, No. 3*, 1995.

[5] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (snmp). `http://www.ietf.org/rfc/rfc1157.txt`, May 1990.

[6] Cisco IP solution center. `http://www.cisco.com/en/US/products/sw/netmgtsw/ps4748/index.html`.

[7] Cisco Systems Inc. Cisco works small network management solution version 1.5. `http://www.cisco.com/warp/public/cc/pd/wr2k/prodlit/snms_ov.pdf`, 2003.

[8] Cisco Systems, Inc. *Cisco IOS Configuration Fundamentals Command Reference*, 2006. Release 12.4.

[9] Distributed Management Task Force, Inc. `http://www.dmtf.org`.

[10] DSL forum TR-069. `http://www.dslforum.org/aboutdsl/tr_table.html`.

[11] R. Enns. NETCONF configuration protocol. `http://www.ietf.org/internet-drafts/draft-ietf-netconf-prot-12.txt`, February 2006.

[12] N. Desai et al. A case study in configuration management tool deployment. In *Proc. of the Large Installations Systems Administration (LISA) Conference*, December 2005.

[13] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[14] A. Feldmann and J. Rexford. IP network configuration for intradomain traffic engineering. In *IEEE Network Magazine*, September 2001.

[15] J. Gottlieb, A. Greenberg, J. Rexford, and Jia Wang. Automated provisioning of BGP customers. In *IEEE Network Magazine*, December 2003.

[16] Intelliden. `http://www.intelliden.com/`.

[17] T. Li, B. Cole, P. Morton, and D. Li. RFC 2281, Cisco hot standby router protocol (HSRP). *Internet Engineering Task Force*, March 1998. `http://www.ietf.org/rfc/rfc2281.txt`.

[18] S. Narain. Network configuration management via model finding. In *Proc. of the Large Installations Systems Administration (LISA) Conference*, December 2005.

[19] Network configuration (netconf). `http://www.ietf.org/html.charters/netconf-charter.html`.

[20] Opsware. `http://www.opsware.com/`.

[21] Voyence. `http://www.voyence.com/`.

[22] Yechiam Yemini, Alexander Konstantinou, and Danilo Florissi. NESTOR: An architecture for network self-management and organization. *IEEE Journal on Selected Areas in Communications*, 18(5):758–766, May 2000.