Autocompletion for Network Configurations

Ahsan Mahmood

Senior Thesis Proposal

Abstract

TODO

1 Introduction

A networks backbone is its routing control plane; a set of rules and distributed routing protocols that describe how the network should operate. A control plane is thus defined through configuration files present on every individual routing device in the network. These configurations are written in vendor specific languages (e.g. Cisco and Juniper) and describe very low level behaviours of a particular router. Network operators tasked to configure control planes may also be required to satisfy various policies that the owning organization wants to enforce: e.g certain devices should always be blocked from communicating with higher privileged devices.

Research has shown that configuring control planes can be extremely complex in modern networks [1]. Consequently, this causes configurations to be prone to errors, most of which are only uncovered during operation after a failure has already dealt significant damage [9].For example, in 2012, failure of a router in a Microsoft Azure data center triggered previously unknown configuration errors on other devices, degrading service in the West Europe region for over two hours [7]. Similarly in 2017, Google made a small error in a protocol configuration which interrupted Japan's Internet for several hours [8]. These examples highlight a need to develop highly resilient configurations that perform reliably.

Network operators thus try to minimize extraneous features by reusing existing configurations that have been known to work in the past. When creating a network, operators typically write templates containing specific configuration lines that define a base set of behaviours for different router roles [1]. These templates are then used to specialize individual routers to achieve objectives for their respective part of the network. Due to varying router specifications, the template systems used allow network operators to fill in parameters with appropriate information each time the template is used.

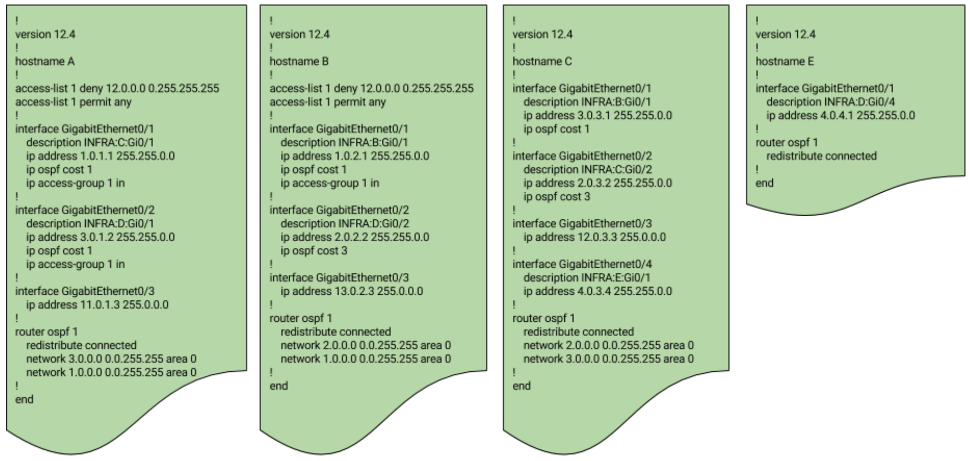
Writing templates, however, can be an inefficient solution when dealing with special cases that deviate greatly from the predefined archetypal configurations. We thus propose a different approach that can serve to complement existing techniques for writing routing configurations. We consider the problem of writing network configurations to be analogous to writing software code. Most configurations are written using vendor specific languages, that make use of rules and keywords similar to traditional programming languages. We envision an interactive system inspired by code completion engines that could be invoked by network operators as they are writing router configurations to offer them suggestions for what to put in next, or list the options available from the invocation point.

Recent research on software systems has shown that codebases tend to contain regularities, much like natural languages [2]. This has motivated further research on using traditional Natural Language Processing techniques for code completion and token suggestion, resulting in fairly accurate models [2, 5].We hypothesize a similar regularity for network configurations, especially since they tend to be homogeneous by design, reusing the same set of keywords/tokens. Some of our work over the summer tried to quantify this similarity between configurations. We analyzed router configurations from a large research university and calculated the average number of tokens shared by a particular router with the rest of the network. Our preliminary results showed that configurations shared between 85

2 Background

2.1 Network Configurations

Router configuration files are often written in a vendor specific language, the popular ones being provided by Cisco and Junyper systems. These files often exist as plain text files on the routers and can be thought of as a static rule base for the device. A configuration file is composed of different sections which are called stanzas. Stanza types could be router, access-control list, interface etc. Each stanza describes the router's particular role in relation to the stanza type. Network operators will configure these stanzas to define how the routers interact with each other. For example, operators might specify which devices the given router is connected to and what protocol it should follow when communicating with such devices. Additionally, they could enforce security measures by using access-control lists to block certain hosts from entering or leaving a network.



```
!
version 12.4
!
hostname A
!
access-list 1 deny 12.0.0.0 0.255.255.255
access-list 1 permit any
!
interface GigabitEthernet0/1
   description INFRA:C:Gi0/1
   ip address 1.0.1.1 255.255.0.0
   ip ospf cost 1
   ip access-group 1 in
!
interface GigabitEthernet0/2
   description INFRA:D:Gi0/1
   ip address 3.0.1.2 255.255.0.0
   ip ospf cost 1
   ip access-group 1 in
!
interface GigabitEthernet0/3
   ip address 11.0.1.3 255.0.0.0
!
router ospf 1
   redistribute connected
   network 3.0.0.0 0.0.255.255 area 0
   network 1.0.0.0 0.0.255.255 area 0
!
end
```

```
!
version 12.4
!
hostname B
!
access-list 1 deny 12.0.0.0 0.255.255.255
access-list 1 permit any
!
interface GigabitEthernet0/1
   description INFRA:B:Gi0/1
   ip address 1.0.2.1 255.255.0.0
   ip ospf cost 1
   ip access-group 1 in
!
interface GigabitEthernet0/2
   description INFRA:D:Gi0/2
   ip address 2.0.2.2 255.255.0.0
   ip ospf cost 3
!
interface GigabitEthernet0/3
   ip address 13.0.2.3 255.0.0.0
!
router ospf 1
   redistribute connected
   network 2.0.0.0 0.0.255.255 area 0
   network 1.0.0.0 0.0.255.255 area 0
!
end
```

```
!
version 12.4
!
hostname C
!
interface GigabitEthernet0/1
   description INFRA:B:Gi0/1
   ip address 3.0.3.1 255.255.0.0
   ip ospf cost 1
!
interface GigabitEthernet0/2
   description INFRA:C:Gi0/2
   ip address 2.0.3.2 255.255.0.0
   ip ospf cost 3
!
interface GigabitEthernet0/3
   ip address 12.0.3.3 255.0.0.0
!
interface GigabitEthernet0/4
   description INFRA:E:Gi0/1
   ip address 4.0.3.4 255.255.0.0
!
router ospf 1
   redistribute connected
   network 2.0.0.0 0.0.255.255 area 0
   network 3.0.0.0 0.0.255.255 area 0
!
end
```

```
!
version 12.4
!
hostname E
!
interface GigabitEthernet0/1
   description INFRA:D:Gi0/4
   ip address 4.0.4.1 255.255.0.0
!
router ospf 1
   redistribute connected
!
end
```

Figure 1: Here is what a set of simplified configurations for a small network employing a single OSPF protocol

2.2 Network Management Tools

Network management tools are built to assist network operators as they design and manage router configurations. Most of these tools offer some form a Command Line Interface, where the operators can use vendor specific languages to update router configurations. Often, these CLIs will offer rudimentary tab completion, where they will alphabetically suggest all the options available for a token from the invocation point. These are sometimes unhelpful as the user then has to search for the desired completion.

A recurring drawback of these tools is that they focus mostly on updating existing configurations. They do not provide any additional functionality for writing new configurations other than utilizing templates. Even in the latter case, the operators will have to fill in the templates appropriately or write their own custom templates for specialized router roles. Our work acknowledges that in practice no networks functionality can be captured by templates alone. Thus, there is a need for an engine that can distinguish itself from these existing tools by being agnostic towards where it is used in the network development life cycle. We expect our engine to perform consistently whether invoked while writing new configurations or updating existing ones.

2.3 Code Completion

Traditional completion techniques, such as those seen in IDEs, generate context aware models of program histories. In doing so, code completion engines often have to be aware of the grammar of the programming language and make suggestions based off that. These solutions offer fairly respectable accuracies but come with their idiosyncrasies. Popular IDEs, such as IntelliJ or Eclipse, use relatively simple type based inferential techniques to suggest all methods available for an object, usually sorted in alphabetical order. Researchers, on the other hand, have proposed more intelligent forms of code completion techniques in the past. Early work started by adopting rule based approaches where a database of predefined rules could be continuously queried to carry out possible completion tasks [3]. Other researchers explored how to make use of program history to offer suggestions based on what users had done in the past [6].

Eventually people started applying machine learning techniques, such as KNNs, to extract patterns from existing code bases and building models that could be used to rank possible predictions for a given input vector. All these techniques, however, require some form of context extraction, so that information about the codebase can be stored e.g. in form of a feature vector.They heavily leverage the existing code structure and require knowledge about the grammar of the programming language. A similar methodology for network configurations would require more input from our end to ensure that the context of the tokens was properly understood. However, NLP techniques can generate predictions based on token usage and do not need to be explicitly aware of the grammar. This allows us to use these techniques independent of vendor specific configuration languages.

2.4 N-gram Models

Consider a sequence of tokens in a body of text (in our case, network configurations). We can statistically model how likely tokens are to follow other tokens. We accomplish this by calculating the conditional probabilities of certain tokens appearing in the text. Given a sequence of tokens $a_1, a_2, a_3, ..., a_n$, we can calculate the probability of a_2 occurring given that a_1 has already occurred i-e $p(a_2|a_1)$. We continue by calculating the probability of a_3 given a_2, and so on. These probabilities would be estimated by counting the frequency by which a given pair occurs in our training data. Since we looked at two tokens at a time, this is called a bigram model. More generally, predicting how likely a token is to show up based on the previous $n - 1$ tokens is called an n-gram model. In our work, we plan to use bigram and trigram models.

2.5 Likelihood estimators

Likelihood ratios are one approach to hypothesis testing. The two hypotheses in our case are:

$$\text{Hypothesis 1: } P(w_2|w_1) = p = P(w_2|\neg w_1)$$
$$\text{Hypothesis 2: } P(w_2|w_1) = p_1 \neq p_2 = P(w_2|\neg w_1)$$

A likelihood estimator is simply a number that tells us how much more likely one hypothesis is than the other. They also have an added advantage of generally being more appropriate for sparse data than other tests. Our system internally uses the Manning and Schutze (5.3.4) version of likelihood estimators [4].

References

[1] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. pages 335–348, 2009.

[2] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. T. Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, 2016.

[3] G. E. Kaiser and P. H. Feiler. An architecture for intelligent assistance in software development. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 180–188, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[4] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[5] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. *SIGPLAN Not.*, 49(6):419–428, June 2014.

[6] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[7] Y. Sverdlik. Microsoft: misconfigured network device led to azure outage. `https://goo.gl/Y5sDov`.

[8] Web. Google made a tiny error and it broke half the internet in japan. `https://thenextweb.com/google/2017/08/28/google-japan-internet-blackout/`.

[9] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. pages 159–172, 2011.