Hey Network, Can You Understand Me?

Azzam Alsudais and Eric Keller
University of Colorado Boulder
Boulder, Colorado 80309
Email: {Azzam.Alsudais, Eric.Keller}@Colorado.edu

***Abstract*—In this paper, we introduce Natural Language Processing to network management by leveraging the capabilities of Natural Language Processing tools, such as speech recognition and text parsing, to extract useful information to build network tasks. We propose an intermediate network-agnostic layer that acts as the medium between natural language input (spoken or written) and different network implementations. We have leveraged the programmability that Software Defined Networks (SDN) offers to build a prototype tool that takes natural language text as an input and uses it to build abstract tasks. Such tasks are then passed to a network controller to be performed in real time. To our knowledge, this is the first work that provides such interface between network users (in the form of natural language) and different network systems.**

I. Introduction

Computer networks, and consequently network management, can be as large and complex as data center networks, or as small and simple as home networks. Despite their size and complexity, networks need to be properly managed to ensure that they operate and run as expected. Therefore, network management concerns not only network administrators and engineers, but also concerns application developers, employees, and end users at their homes, when needing to configure or debug their networks (e.g., port forwarding, testing connectivity, showing logged-in devices).

Network operators spend billions of dollars every year to manage and troubleshoot their network systems [1]. This includes the management and troubleshooting of network devices and end hosts. Although with the advances of network systems such as Software Defined Networks (SDN), network management is still a difficult task that greatly consumes the time of network engineers and users and introduces the added complexity of needing to write software for some tasks [1].

There is a large body of work on facilitating and making network management and troubleshooting more effective and feasible [2], [3], [4]. However, there is a missing link in the interaction between networks and network users[1]. That is, in order for one to manage a network, they need to learn a specific network configuration language. Whether that is in the form of configuring a specific SDN controller or configuring a modem webpage for an Internet user. We argue that network management should not be difficult, rather it should be "user-friendly", especially with the growing adoption of programmable networks and Software Defined Networks (SDN) [5], [6] since it provides programmability that can be leveraged to run network-wide applications.

What if instead of configuring a network, we could just talk to it, tell it what we want, and it understood? In this paper, we bridge the gap between network management and network users by proposing a new layer of abstraction that resides atop existing network management solutions. In particular, we propose a new abstraction for common network tasks so that a natural language input (e.g., written or spoken) can be mapped onto a network task, and eventually passed down to the network itself to be performed. And by leveraging the capabilities that Natural Language Processing (NLP) offers, we implement a tool that accepts natural text and turns it into a network task. What we propose is different than other general NLP applications (*e.g.,* a web search engine or a smartphone personal assistant). That is, we propose a general abstraction that can be used by different implementations of network systems; on the other hand, other NLP applications are specific to one system. In particular, we make the following contributions:

- We propose a novel network-agnostic abstraction for common network management tasks in which they can be encapsulated.
- We build a tool that parses natural language text and construct abstract network tasks, which then are performed in real time. To our knowledge, this is the first work that enables using natural language to manage networks.

As an initial step, we make the following assumptions:

1) A high-level information about the network is assumed available and provided to our tool (*e.g.,* mapping between logical and network-level names). This allows users to speak in a high-level tone (not being required to specify IP addresses, but instead say something like "host-1"). Of course, the user is also allowed to refer to network-level names like IP addresses.
2) An underlying set of network tasks is assumed to be already implemented (for a specific network implementation). Our tool only provides the abstraction layer between the natural language and the specific network implementation (e.g. Floodlight and Ryu).

The rest of the paper is organized as follows. Section II introduces NLP and what it can offer to help make network management easier. Section III describes common network management tasks and proposes a new task abstraction. In section IV we describe our implementation of a test tool that

[1]In this paper, we refer to one who interacts with the network, in any possible way, as a network user.

translates natural text to the abstract task, and how the abstract task is then conveyed to the network. Sections V and VI discuss future work and conclusion, respectively.

II. What NLP Can Offer to Network Management

In this section we describe what Natural Language Processing can offer to network management, and how we leverage NLP to enable a more convenient way to manage network systems.

A. NLP Overview

NLP provides the ability for humans to interact with machines without having to go through learning a specific machine language. It bridges the gap between humans and machines by processing natural human language and derive useful information from it. Eventually, the derived information helps a machine understand natural language, in the form of words and sentences, and then decides the appropriate course of action to take. As a result, NLP has made it possible to turn smart phones into intelligent personal assistants (IPAs) [7], to query a database using natural human text [8], and to manage and control one's smart home by speech or text messages [9].

The NLP Pipeline: NLP applications follow a common processing pipeline [10]. It can be summarized into five main stages in the following order: phonology (speech analysis), morphology (lexical analysis), syntax (parsing), semantics (context awareness), and application reasoning (domain-specific). By leveraging this pipeline, useful information can be extracted from any form of language (written or spoken). It is worth mentioning that following all stages of the pipeline is not required for every single application. Rather, some applications need information that can be extracted from the first stage of the pipeline (e.g., dictation software). Tools for the first four stages of the pipeline are widely available [11] since these stages focus on the natural language aspect, which is shared among different applications. However, the application reasoning stage, as mentioned, is domain-specific. For instance, if one wishes to design an NLP application for controlling their computer, they need to write the application specifically for that computer system. Likewise, our work in this paper is about designing a domain-specific application for network systems, with the abstraction of network tasks being the domain that we focus on.

B. NLP for Network Management

We envision a system that enables its users to interact with their networks by either spoken or written forms of natural language. For instance, a user can say something like *"Can A talk to B?"*, and the system would extract information such as: source and destination IP addresses, and protocol (e.g., *icmp*, *tcp*, or, *udp*) and encapsulates this information in an abstract task (described in more details in section III) and sends it to the network controller. The controller then maps the information in the abstract task onto the appropriate function calls (e.g., queries its ACL or Firewall tables), and returns the answer, which is then conveyed to the user.

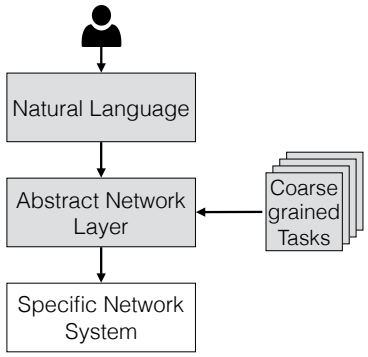

Fig. 1. The high-level processing pipeline

We argue that introducing the capabilities of NLP can greatly benefit network management in two ways, *convenience* and *speed*. First, network users can "speak their minds" in terms of what they want from the network. This makes the process of performing a network task (e.g., debugging a faulty link, querying network devices statistics, allowing or blocking two hosts to communicate) much more convenient, as compared to writing different set of commands in a specific language or format. Second, when users interact with their own networks using natural language in real time, we argue that it takes less time to perform a task, as compared to the time it takes to write in a specific format. This is different than having pre-programmed automated tasks. We propose a fully dynamic system that allows its users to perform tasks that have not necessarily been encountered by the system before, by leveraging NLP capabilities.

III. Network Task Abstraction

In this section, we describe what the common network management tasks are, and propose a new abstraction for such tasks. We believe that what we propose is not a replacement for existing network management systems, but rather is an abstraction layer that resides on top of existing systems and serves as a translation layer between natural language and such systems. As seen in Figure-1, there will be a collection of coarse grained network tasks and our tool will take in natural language and configure and use one or more of these tasks.

A. Task Abstraction

First, we need to determine what these tasks are that can be composed together to perform some network management.

1) Common Network Tasks: Common network tasks can be collected from different sources. We gathered tens of network tasks from different SDN controller APIs (e.g., Floodlight [12] and Ryu [13]), network configuration manuals [14], different papers on network debugging and management [2], [3], [4], and from textbooks [15] to help us form an intuition of what types of tasks concern network operators. In Table-I we show a sample of common network tasks and what type of information can be extracted from them. The other details of the table are explained in the following subsection III-A2. Collecting and

#	Task	Abstract information	Keyword	REST API Type
1	is h1 connected?	- end point - keyword	connected	GET
2	is h3 reachable from h2?	- end points - keyword	reach	GET
3	avoid switch-1	- end point - keyword	avoid	POST / DELETE
4	allow h4 to talk to h5	- endpoints - keyword	reach	POST / DELETE
5	rate limit h6 to 100 Mbps	- end point - keyword - extra information	rate	POST / DELETE
6	route flow x through firewall	- end point - keyword - extra information	route	POST / DELETE
7	assigne vlan #1 to 10.0.0.20	- end points - keyword	assign-to-vlan	POST / DELETE

TABLE I
EXAMPLES OF COMMON NETWORK TASKS

analyzing these tasks has helped us design an abstraction layer between natural language and different network systems.

Figure-1 shows the high-level design of what we propose. Relying on *assumption-2* in section I, our work is focused on the shaded layers of Figure-1, and we assume that the actual implementation of the network task is already implemented. However, as we show in section IV, we provide a proof-of-concept implementation using Floodlight [12] to demonstrate that our tool can be used by network developers to develop interfaces that connect to it. This modular design helps writing the NLP parts and the abstraction layer only once. Then different systems can connect to and call our tool to leverage its capabilities to convert natural language to abstract tasks.

2) Abstract Task Structure: It became clear after looking at those common network tasks that they share a common pattern. In particular, we make the following observations:

- Most network tasks can be encapsulated in an abstract form that contains: one or more *endpoints*, a *keyword*, *extra information*, and a task *type*. To put things in perspective, here is how an abstract task structure looks like, which is the building block of the second layer of Figure-1:

```
struct task
{
        endpoints[];
        keyword;
        extraInfo;
        type;
}
```

- The *endpoints* mentioned above can encapsulate other information as well, e.g. when adding a user to a group, the user can be one *endpoint* and the group can be the other *endpoint*, and also when assigning vlans to IP addresses (the IP address being one *endpoint* and the vlan-id being the other *endpoint*)

Building Abstract Tasks: Following the abstract structure described above, now we can build tasks by extracting information out of the natural language input (written or spoken). The *endpoints* are used to carry information about where the task should take place. For instance, the *endpoint* for *task-1* in Table-I is *h1*, and the *endpoints* for *task-7* are the *vlan-id* and the IP address 10.0.0.20. The *keyword* is used to distinguish different tasks and to group similar tasks together, and is also used to help provide better classification when processing a sentence provided by the user. For instance, if the user says *"allow h1 to talk to h2"*, then the *keyword* is *talk*. In section IV we describe in details how we group similar tasks together in order to provide the ability of processing as many sentences as possible. The *extra information* field contains further information about a task to help map fine-grained tasks like the rate limiting parameter of *task-5* in Table-I. In this task, the rate (100 Mbps) is stored in the *extra information* field to help provide this information to the underlying network system. The *type* field refers to the task *type*, whether there is an action required or it is simply a query. We borrow the concept of Representational State Transfer (REST) [16] from web services implementation to further help us determine what the type of the task is. That is, the *POST* and *DELETE* types are used to distinguish tasks that are adding or deleting some type of network configuration, respectively. For instance, in *task-5* if the *REST API* type is *POST*, then *h6* should be rate limited to 100 Mbps. On the other hand, if the *REST API* is *DELETE*, then *h6* would not be rate limited, and so on.

B. Example Use Cases

Different Talkers to the Network: To provide a better understanding of how our tool can be used, we list some examples and use cases of who can talk to the network [2]. These examples are by no means inclusive. We only list them to give a better sense of how the tool can be used. In general, any user with a device that is connected to the network should be able to talk to the network in one way or another. However, determining the permissions for each group of users, in terms of what they can and cannot do, is out of the scope of this paper.

Network administrators who want to perform a scheduled maintenance for a specific switch can say something like *"avoid switch x"*. Then our tool shall translate this sentence to the abstract task form by determining what the task *keyword* is and then extracting the expected information from the sentence (*e.g., endpoints*). After the abstract task is build, our tool can then call the network-specific API that eventually helps rerouting traffic around switch x (relying on *assumption-2* in section I, we assume this part is implemented by network developers).

[2]Even though throughout the paper we focus on concepts related to SDN, the abstraction we are proposing can be used to encapsulate tasks for legacy networks as well.

Enterprise **top managers** can use such tool to query about the state of their network or to even force some type of action. For instance, let us take the following scenario. A CEO suspects that one of his employees is engaged in suspicious activities. He can say something like *"route Chuck's traffic through the IPS"*, and our tool would extract the information from this sentence. In this scenario, the *endpoints* become Chuck's IP address and the IPS's IP address, and the *keyword* becomes *"route"*. Then the tool builds the abstract task, which eventually is passed down to the network controller of the enterprise.

Parents at their homes can leverage such tool to help them manage their home networks. In particular, parental control tasks can be encapsulated in our abstract task form. For instance, say a parent wants to limit her son's computer time. She can say something like *"disconnect Mike's computer"*, and then our tool would extract the required information to build the abstract task. In this scenario, the *endpoint* is Mike's device IP address, and the *keyword* is *"disconnect"*. The abstract task then is sent to the home modem to eventually execute the task through the parental control interface.

One other area that can benefit from such abstraction is the Internet of Things (**IoT**). The IoT *endpoints* in the abstract task can refer to the IoT devices (*i.e.,* things). And the tool can be modified to include *keywords* that are specific for the domain of IoT. For instance, the user can say something like *"turn on my car"*. The *endpoint* would be the user's car, the *keyword* would be *"turn"*, and the task type would be *POST* (*DELETE* when the task is meant to turn the car off).

C. Putting it Together

Here we walk through each step starting from the user's input, going through processing the request and building the task, and ending with actually performing the task in the network. Figure-2 shows the detailed processing stages involved in processing a request. We emphasize that our work is focused on the shaded parts of the figure, while the parts that are not shaded are assumed implemented.

1) The NLP Pipeline: In the NLP pipeline (as described in section-II-A), the first step is to recognize speech, which involves converting speech into text. When the input is ready in the form of a text sentence, the Part-of-speech Tagger (POS Tagger) comes in to tag words with the proper grammatical tags (*e.g.,* nouns, verbs, adjectives, etc). These tags comes in handy when extracting the required information later. Then the sentence is parsed to yield the parsing tree, which then is used to determine the relationships between different words. This relationship is useful in determining how the *endpoints* should be represented.

2) The Abstract Network Layer:

Keywords*:* After the parser yields its tree, the tree is used by the *Information Extractor* to extract information like what the *keyword* and *endpoints* are, and based on the *keyword* it determines whether to expect *extra information* or not. Since the *keywords* are provided to the tool in advance, detecting them is done by scanning the grammatical lemmas (*i.e.,* words basic forms) and looking for matching *keywords*. Upon finding a *keyword*, the tool determines what type of information to look for in the sentence. For instance, if the *keyword* is *rate*, referring to a rate-limiting task, then the tool knows that it should extract information like what the device/flow is and by how much to limit the rate of that specific *endpoint*.

Endpoints*:* Relying on *assumption-1* in section-I, a list of mappings between logical and physical names is assumed provided. Therefore, detecting the *endpoints* in a sentence is straightforward by scanning it looking for matching *endpoints*. However, determining the relationships between such *endpoints* (*i.e.,* what the source and destination are) needs to be inferred from the sentence. Such relationship can be inferred by looking at the prepositions in the sentence (*e.g.,* from, to, at, etc). These are used in determining the direction and relationship between *endpoints*. For instance, *task-2* in Table-I should have *h2* as the source and *h3* as the destination, even though *h3* precedes *h2*. But when the word *"between"* is encountered, the relationship between the two *endpoints* should be inferred.

Extra Information*:* Once the task *keyword* is detected, the tool determines whether it needs to look for *extra information* in the sentence or not. When the tool determines that the task needs *extra information*, it searches the sentence for that information. For example, given *task-6*, the tool knows that it should look for the flow information in the sentence. Therefore, the *extra information* field for that task should contain the necessary flow information, such as: source and destination IP addresses, Port numbers, and protocol.

Task Type*:* Inferring the correct task *type* is not trivial. That is, distinguishing between all three task *types* (*GET*, *POST*, *DELETE*) requires processing the sentence at a deeper level. For simplicity, when the sentence is a question, we assign *GET* as the task *type*. However, not all *GET* tasks are questions. For instance, the sentence *"get me switch X's statistics"* should be assigned *GET* as its *type*. To uncover the complexity of this issue, we provide the tool with a list of *"query"* words that refer to a *GET* task (*e.g.,* get, poll, lookup, etc). Together with the *keyword* of the task, determining whether the task should be *GET* is possible. As for the *types POST* and *DELETE*, inferring the correct *type* is done by looking for negations in the sentence (*e.g., "don't allow h1 to talk to h2"*). Once the *Information Extractor* extracts the *keyword* and *endpoints*, it looks at the grammatical lemmas of the sentence to see if there are any negation words like *"not"*, and once it finds one that is associated with the main verb of the sentence, it assigns *DELETE* as the task *type*. Otherwise, the task is assigned *POST* as its *type*.

The information extracted by the *Information Extractor* in Figure-2 is passed down to the task builder, which has a JSON formatted structure that it expects for every *keyword*. Once the task builder makes sure that it has all the information it needs, it constructs the task using the same structure described above in section III-A2. Lastly, the serializer takes the abstract task and then serializes it so that it gets sent to the network controller (we have relaxed the design decision on this as

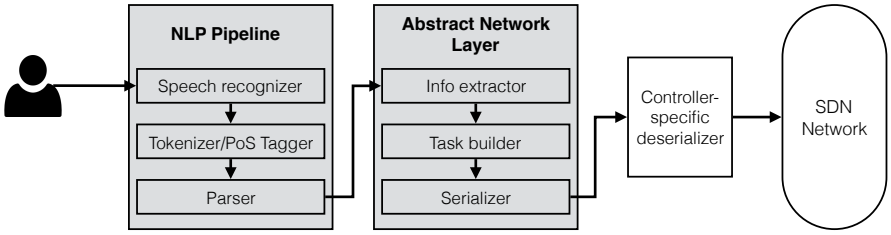

Fig. 2. The complete processing pipeline

to whether to use network sockets or *REST API* calls to communicate with the network-specific deserializer. In our prototype implementation we have used sockets).

Now, network developers come in where they implement their network-specific deserializer. We believe that this step should not take much effort. In fact, as we describe in section IV, we have implemented a Floodlight [12] module that receives abstract tasks and performs them.

IV. Implementation

In this section, we describe how we have implemented a tool that leverages the abstraction layer we proposed in section III. We show how our proposed abstraction can encapsulate much more tasks than we initially thought it would, with as few lines of code as less than 1K lines in Java.

A. Tools and Setup

For the NLP part of the implementaion, we used Stanford's CoreNLP framework [11] to take advantage of its NLP capabilities. We also used WordNet [17] to expand our *keyword* space so that it contains the synonyms for the *keywords* we provide to the tool. And for the networking part of the implementation, we used Mininet [18] with Floodlight [12] to simulate a network of multiple hosts.

B. Implementation

We wrote a Java application that reads sentences entered by the user, parses the sentence using CoreNLP Stanford's parser [11], extracts information and builds the abstract task, and sends the task to our Floodlight controller. On the controller's side, we wrote a Floodlight module that listens for connections from our tool, and then reads abstract tasks and converts them into actual API calls provided by Floodlight.

Our Java tool[3] contains three classes: *AbstractTask* to represent the abstract task fields, *InformationExtractor* whose job is to extract information from the user's input, and *NetAssistant* that interacts with the user and calls the *InformationExtractor* to help create and build an *AbstractTask* object, which eventually is sent to our Floodlight module. Next, we describe in details how the *InformationExtractor* and Floodlight module are implemented.

[3]We would like to note that the tool is at a proof-of-concept stage and we are still exploring different approaches, such as using classification to provide more flexibility in determining what task a sentence is referring to (instead of looking for specific keywords).

We have tested the tool with many sentences to see if it would construct the right abstract tasks for those sentences, and we found that the tool correctly constructed 80% of the sentences (for the 20%, the tool incorrectly assigned *POST* instead of *GET*, mainly because the sentence did not include any of the *query* words that we provided the tool with). This can be prevented by expanding the *query* words list. In addition, a validation feature could be implemented to prevent such errors, where the user is prompted to approve/disapprove any generated task before it is actually executed.

We include two examples of sentences that the tool successfully processed: *"can h1 talk to h2?"* and *"route h1's traffic through the firewall."*. For the first sentence, the tool yielded this abstract task: "{"keyword":"talk", "endpoints":{"src":"10.0.0.1", "dst":"10.0.0.2"}, "type":"get"}". And likewise, it yielded this abstract task for the second sentence: "{"keyword":"route", "endpoints":{"src":"10.0.0.1", "dst":"10.0.1.1"}, "type":"post"}". The IP addresses were obtained from the mapping information provided to the tool (based on *assumption-1* in section-I).

1) Information Extraction: This is the building block of the tool since its job is to determine what and how to extract information from a sentence, which then is used to initialize the abstract task. Currently, our tool is able to extract *keywords* and *endpoints* from a sentence, and determines the abstract task *type*. Extracting *keywords*, *endpoints*, and task *types* is described in section III-C2. Here we describe the data structures that we used to represent the the provided lists of *keywords* and *endpoints*. To improve performance, we used HashMaps to represent both lists. The *endpoints* data structure contains the mapping between logical and physical names, with the key being the logical name and the value being the physical name. Likewise, the same data structure is used to represent the list of supported *keywords*. However, it is worth mentioning that a naive approach of providing a list of supported *keywords* would limit the space of possible sentences a user can say.

To overcome this, the list of *keywords* needs to be expanded. But there is a trade-off between expanding the list of supported *keywords* and not complicating the abstract layer. In other words, when adding so many *keywords* that the abstract layer in Figure-2 can support, this results in complicating the deserializer processing (*i.e.,* having to process many *keywords* that potentially refer to the same task). Fortunately, this does not have to be the case. We optimized the *keywords* HashMap

in order to support more *keywords*, and at the same time preserve the simplicity that the abstract layer shall provide. In particular, we leveraged the capabilities of WordNet [17] to lookup synonyms of *keywords*, and had them refer to the same *keyword* in the data structure. For instance, the words *"let"*, *"permit"*, and *"allow"* all refer to the same *keyword "allow"*. Following this approach has reduced the space of possible *keywords* by 70%.

2) The Deserializer: Although in this paper we assume that implementing the deserializer is left for network developers, here we show that implementing the deserializer is not meant to be a burden, since it is written only once. Our deserializer is a Floodlight module, which currently supports 13 distinct *keywords*, and receives abstract tasks in JSON format from our Java tool, described in the previous subsection. Upon receiving a task, the module parses the JSON file and extracts the information from the abstract task fields. Then it uses the appropriate API calls to perform the task. After the task is performed, a text feedback is returned to the user. For instance, if the received abstract task contains the following information: {"keyword":"allow", "endpoints": {"src": "10.0.0.1", "dst": "10.0.0.2"},"type":"post"}, the module would call its ACL service to add a new ACL rule that allows the connection from 10.0.0.1 to 10.0.0.2.

V. Discussion and Future Work

The strength of the abstraction layer depends on its ability to support and encapsulate as many tasks as possible, while preserving the abstraction simplicity of representing such tasks. We plan on further investigating the impact of adding and supporting more network tasks and more complex composition of tasks, and analyze their effect on the abstraction layer (whether modifications are needed or not). One example of such tasks is cloud management tasks, even though that a good portion of them can still be encapsulated in our abstract task. Adding new tasks mainly results in adding new *keywords*; however, the challenge is in determining whether new tasks can be encapsulated in existing *keywords*.

In our prototype, we implemented a tool that accepts text from the user. To provide more convenience, a more natural approach would be to accept voice commands from the user, like the use of IPAs in smartphones. We plan on leveraging speech recognition tools to support such feature. In addition, we are planning on implementing the tool in different systems, including developing a mobile agent that runs on smartphones to do the same job of our prototype tool. Furthermore, in order to improve convenience, a context-aware direction is worth pursuing. That is, when processing a sentence, the tool should be aware of the context of that sentence. For example, a user can say something like *"is my computer connected to the Internet?"*, and the tool should determine what *endpoint* the user is talking about. This involves using and developing learning techniques that allows such feature to be implemented.

Of course, collaboration is a key aspect in improving any work. Therefore, open sourcing our tool is our goal once it is fully implemented and tested. We hope by making the code publicly available that the community contributes to it and takes it to a more advanced stage.

VI. Conclusion

In this paper, we proposed an abstract network layer that acts as an intermediate layer between natural language and different network management systems. By collecting and analyzing common network management tasks, we proposed an abstract structure in which such tasks can be encapsulated. In addition, we implemented a tool that accepts natural language input, as a written text, and extracts information needed to build the abstract network task. We argued that network management should be more convenient by leveraging the capabilities that Natural Language Processing (NLP) offers.

We note that we are not proposing a replacement to existing network management systems, but rather we are proposing an abstraction layer that resides atop existing network management systems to enable the encapsulation of natural language input in an abstract form of network tasks. To our knowledge, this is the first paper that proposes the use of natural language to manage computer networks.

References

[1] C. Bureau, "Network operators to spend over $1.8 bn in 2016," 2016.

[2] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, *et al.*, "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 479–491, ACM, 2015.

[3] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies (CoNEXT 12)*, pp. 241–252, ACM, 2012.

[4] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 113–126, 2012.

[5] S. Brown, "Ninth annual state of the network global study," 2016.

[6] M. Shirer, B. Casemore, and R. Mehra, "Sdn market to experience strong growth over next several years," 2016.

[7] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, *et al.*, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *ACM SIGPLAN Notices*, vol. 50, pp. 223–238, ACM, 2015.

[8] "Quepy: Transform natural language to database queries." http://quepy.machinalis.com, 2016.

[9] M. Zuckerberg, "Building jarvis." https://www.facebook.com/notes/mark-zuckerberg/building-jarvis/10154361492931634/, 2016.

[10] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python*. " O'Reilly Media, Inc.", 2009.

[11] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit.," in *ACL (System Demonstrations)*, pp. 55–60, 2014.

[12] "Project floodlight." http://www.projectfloodlight.org.

[13] "Ryu the networking operating system." http://ryu.readthedocs.io.

[14] C. Systems, *Network Management Configuration Guide, Cisco IOS Release 15.1S*. Cisco Systems, Inc, 2011.

[15] M. Subramanian, *Network management: principles and practice*. Pearson Education India, 2010.

[16] R. Fielding, "Representational state transfer," *Architectural Styles and the Design of Netowork-based Software Architecture*, pp. 76–85, 2000.

[17] C. Fellbaum, *WordNet*. Wiley Online Library, 1998.

[18] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 19, ACM, 2010.