App2net: A Platform to Transfer and Configure Applications on Programmable Virtual Networks

Ricardo Luis dos Santos*, Oscar Mauricio Caicedo Rendon†,
Juliano Araujo Wickboldt*, Lisandro Zambenedetti Granville*
*Institute of Informatics – Federal University of Rio Grande do Sul, Brazil
†Telematics Engineering Group – University of Cauca, Colombia
*{rlsantos, jwickboldt, granville}@inf.ufrgs.br – †omcaicedo@unicauca.edu.co

***Abstract*—In programmable virtual networks, simple tasks, like installing software, can be extremely complex. This complexity occurs mainly because the code transfer and initial functional settings in network execution environments are not automated. In addition, the same tasks have different requirements in each service lifecycle stage. In this sense, we propose the App2net platform for enabling the transfer and configuration of network applications in programmable virtual networks that use heterogeneous execution environments. We also propose a taxonomy for grouping code transfer techniques and, based on such techniques, we develop models for code transfer. A prototype has been implemented and tested on realistic network topologies commonly found on the Internet. Results allow us to identify which models improve code transfer consuming fewer resources, regarding service lifecycle stages and network topologies.**

Keywords*—Network Programmability; Network Virtualization; Code Transfer; Network Applications.

I. Introduction

Network programmability arose to introduce the ability to rapidly build, deploy, and manage new services in the network [1]. Since then, several programmability-related technologies emerged, such as Active Networks [2] and Mobile Agents [3], allowing developers (a.k.a *end-users*) to implement and deploy programs into the core network. However, none of these technologies ended up being adopted because they lacked of convincing guarantees about the settings and applications deployed on devices, which could interfere or collapse the production network.

Today, network virtualization technology enables virtual network nodes to be created, moved, and destroyed. Granting virtual nodes to end-users allows them to deploy and run their programs in an isolated way [4]. Thus, in case of misbehaving code, these programs do not affect the production network. In addition, the administrator can easily destroy a set of devices running such code, preventing the network from collapse. Several companies and standardization bodies have carried out efforts to define standards and solutions for networks that support virtualization and programmability (a.k.a *programmable virtual networks* - PVNs) [4], such as Juniper's Junos SDK [5], Cisco's onePK [6], OpenFlow [7], ETSI's NFV [8], and IETF's ForCES [9]. These solutions provide an Execution Environment (EE) that supports the deployment of novel network services as diverse as dynamic configuration [10], multicast [11], and video transcoding [12].

Although the aforementioned solutions enable the development of novel network services, the end-user needs to have extensive knowledge on device instructions and network applications to manage such services. As a consequence, simple tasks, such as transferring or configuring a new network application, become extremely complex and repetitive. First, because none of the solutions automate the code transfer in heterogeneous EEs. Second, because the same tasks have different and conflicting requirements (*e.g.*, minimal network interference or distribution time) in each service lifecycle stage (*e.g.*, design, transition, and operation). And, third, the initial settings must be manually replicated in each EE to set up the logic for delegating data flows to each new service. All this impacts the innovation process in the whole network.

In this paper, we introduce the App2net platform to tackle the problem of transferring and configuring network applications in the context of PVNs. In particular, App2net focuses on enabling end-users to transfer and configure the code among heterogeneous EEs. We also analyze a set of code transfer techniques and propose a taxonomy for grouping them together. Moreover, we develop and analyze models for code transfer based on the previously identified techniques. A prototype has been implemented and tested on realistic network topologies commonly found on the Internet. The results allow us to identify the models that improve the code transfer and consume fewer resources, according to the requirements of service lifecycle stages and realistic network topologies.

The rest of this paper is organized as follows. In Section II, we discuss key research efforts to transfer code to multiple nodes. In Section III, we detail the App2net platform. In Section IV, we introduce the taxonomy of techniques and the models for transferring code. In Section V, we detail the implemented prototype and discuss the results obtained. In Section VI, we present final remarks and future work.

II. Related Work

Technologies like Juniper's Junos SDK [5], Cisco's onePK [6], OpenFlow [7], and ETSI's NFV [8] introduced PVNs with multiple EEs. These technologies focused mainly on making network programmability feasible but did not intend to address code transfer problem. In addition, none of them automate the configuration of network applications. Below, we discuss some of the most important studies about the code transfer.

Zabolotnyi *et al.* [13] proposed a framework for dynamic code transfer in clouds. The framework transfers the code to several virtual machines, which are responsible for executing diverse functions of an application, by three strategies: (*i*)

978-1-4673-7194-0/15/$31.00 ©2015 IEEE

complete, when all application code is provided on request, (*ii*) class-based, if only the requested class is provided; and (*iii*) smart batching, when the code is delivered in highly related batches. These strategies divide the code in small pieces to optimize the distribution time. An important shortcoming is that such framework allows only to transfer Java code.

In OpenFlow networks, HotSwap [14] proposes a disruption-free mechanism to upgrade controllers, which avoids disabling the controller and therefore stopping the network. The mechanism consists in parallel executions of the old and new version of the controller in different network segments. When the new controller is fully initialized and finishes resuming the network state, the update process is completed by replacing the old controller. However, HotSwap is constrained to scenarios with a unique EE.

Olteanu and Raiciu [15] proposed a solution for migrating stateful middleboxes, which provides functionalities ranging from security to performance optimization. This solution executes in parallel both the source (A) and the destination (B) processing for a short time. Next, A copies the code and invariant global state to B. After, two overlapping phases are carried out: idle and freeze-and-copy. In idle, all traffic is forwarded to A, but all packets that do not have any matching state are forwarded to B. In iterative freeze-and-copy phase, when a threshold is reached, A freezes its remaining states and transfers them to B. This solution considers just the migration of processing from one EE to another similar EE.

Although research studies addressing issues such as event propagation and replica updates are not directly linked to the code transfer problem, these studies provide ideas to overcome it. In this context, HyperFlow [16] uses a publish/subscribe system for event propagation among OpenFlow controllers. In such system, all controllers must subscribe to a channel. When an event occurs, it is advertised in that channel and received by all the subscribers. In another study, Nawaf and Torbey [17] proposed a hybrid push/pull strategy for file updates propagation among replicas in mobile ad-hoc networks. This strategy divides network nodes in two groups. The first group subscribes to the source to directly receive updates via push-based communications. The second group receives updates via pull-based communications from the nodes in the first group.

To sum up, some shortcomings of the studies focusing on code transfer are: (*i*) dependence on a programming language, (*ii*) limitation to homogeneous EEs, avoiding the use of distinct technologies; and (*iii*) lack of automation to configure initial functional setting, making the configuration process extremely complex and manual. To overcome these shortcomings, we introduce the App2net platform as follows.

III. App2net Platform

App2net aims to transfer, configure, and manage network applications (*i.e.*, NetApps) in PVNs that use heterogeneous EEs. These EEs can be hosted in virtual or physical devices. We assume, first, that there is a traditional best-effort, TCP/IP network intermediating the communication between App2net and EEs. Second, the end-user uses a Virtual Platform Manager (*e.g.*, OpenStack [18], Eucalyptus [19], or Aurora [20]) to build up a PVN over several Infrastructure Providers. It is worth mentioning that such set up is outside the scope of App2net.

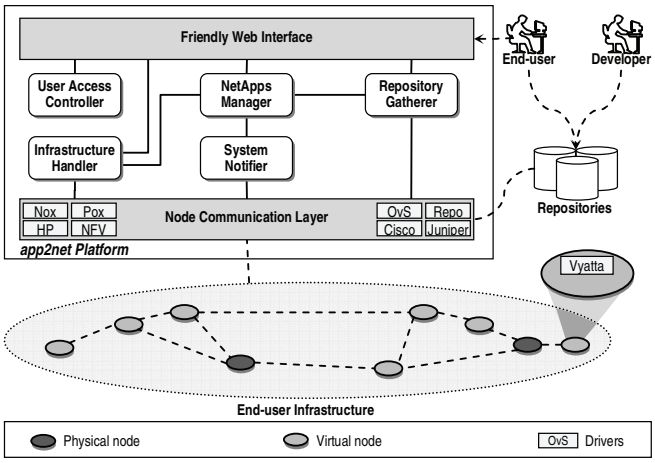

Fig. 1: App2net conceptual architecture.

Let's suppose that the end-user needs to release a service provided by a NetApp such as deep-inspection firewall, video encoding/transcoding, or trust-based broadcasting. For this purpose, the end-user interacts with App2net through a friendly Web interface. First, the end-user provides information about the PVN (*e.g.*, nodes addresses and programmable technologies). Afterward, the end-user selects the NetApps to be distributed. These NetApps and their initial functional settings are stored in Repositories. Then, App2net interacts with PVN nodes to install and configure the selected NetApps. Finally, the end-user can start, resume, pause, and stop NetApps as well as collect data and statistics of them.

Figure 1 depicts the conceptual architecture of App2net and its five main components: User Access Controller, Repository Gatherer, Infrastructure Handler, NetApps Manager, and System Notifier. In addition, Figure 2 shows the main actions performed by the end-user in the App2net. Prior to interacting with the platform, the end-user's credentials need to be checked (action 1). In this regard, the User Access Controller performs actions related to user management, like authenticating and granting permissions. Such component also limits access and actions in networks according to permissions assigned per user, avoiding the improper handling of PVNs and nodes.

After logging into App2net, the end-user configures the Infrastructure Handler with the nodes that compose the PVN, informing just nodes addresses, supported programmable technologies, and access credentials (action 2). This configuration is carried out manually or imported from a Virtual Resources and Interconnection Networks Description Language (VXDL) [21] file. We choose VXDL because it represents the infrastructure elements in a simple XML notation. The Infrastructure Handler is also in charge of: (*i*) installing drivers in nodes (action 3), (*ii*) collecting data about resources in PVN nodes, for example, network interfaces or programming languages (actions 4 and 5), (*iii*) maintaining the list of available devices per PVN; and (*iv*) storing the credentials for accessing devices.

When desired NetApps are available at the local repository, the end-user only requests the installation of them by NetApps Manager. However, if the desired NetApps are hosted in an online repository, then, the end-user must register such repository by the Repository Gatherer (action 6). This Gatherer configures, communicates, and manages the repositories as

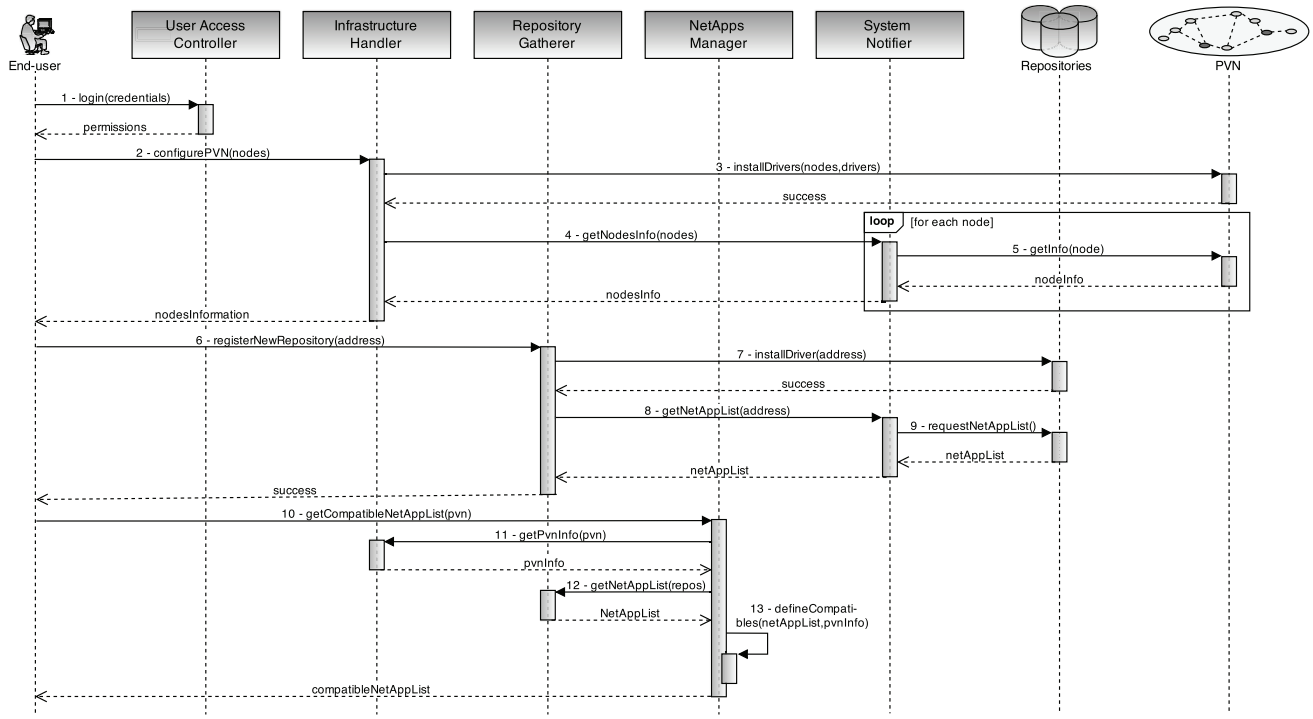

Fig. 2: Sequence diagram about main actions performed in App2net platform.

well as maintains an updated list of available NetApps. To create a new repository, the end-user requests the installation of the corresponding driver (action 7). This driver collects information about available NetApps (actions 8 and 9) and comunicates with the App2net and PVN nodes. It is noteworthy that a repository can be maintained by end-users, companies, or third-party developers interested in providing NetApps.

The NetApps Manager is in charge of installing, uninstalling, configuring, and managing (*e.g.*, start, stop, and resume) NetApps. In this sense, this Manager acts as an orchestrator, communicating and requesting information from other components (action 10). Initially, the Manager retrieves information about PVN nodes and their resources via the Infrastructure Handler (action 11). Then, the Repository Gatherer provides a list of available NetApps (action 12). Afterward, the Manager conducts a matching between requirements and resources in nodes; thus, it only allows to install compatible NetApps (action 13). The NetApps Manager is formed by five modules: Node Selector, Package Transfer, Package Installer, NetApps Configurator, and Data Collector. Figure 3 presents the actions performed by NetApps modules to install and configure a NetApp in a PVN. Note that all messages to repositories or PVN are sent via System Notifier, but these interactions were hidden for the sake of readability.

For installing a new NetApp, the Node Selector module identifies the *compatible nodes* and the repository that holds the corresponding package (action 1). A compatible node is able to install a specific NetApp, meeting the minimum requirements in both software and hardware. Thereafter, the Selector defines which compatible nodes must install the NetApp, according to requirements and the *location flag* (action 2). This flag takes in consideration the application-network interaction and the location of nodes into PVNs. Therefore, it is possible to install and run NetApps at strategic places of PVNs. For instance, a deep-packet inspection firewall uses border value and, thus, it is only installed in the border nodes, avoiding additional and unnecessary computing.

The location flag may take on five values: (*i*) *border* regards to border nodes and all data flows in a particular PVN, (*ii*) *ingress* considers border nodes and input data flows, (*iii*) *egress* denotes that border nodes and output data flows will be used, (*iv*) *custom* allows an end-user to select which nodes and data flows will be used by the NetApp; and (*v*) *all* refers to all nodes and all data flows. Note that, when specifying the location flag with custom value, the NetApp developer may set up the rules to handle the data flows; however, rules for choosing the nodes cannot be specified.

The Package Transfer module receives information about nodes and repositories from Node Selector (action 3). Initially, the Transfer validates the authenticity of the repository using assimetric keys and SHA hashes (actions 4 and 5). Next, this module defines the guidelines (*e.g.*, order and initial transfers) required for network nodes to have a copy of the NetApp package (action 6). Afterward, Package Transfer informs these guidelines to drivers of repository and PVN nodes, which perform the package transfer (actions 7 and 8). Finally, the driver validates the package integrity, using SHA hashes. For the sake of simplicity in presentation, we sum up the notification of guidelines to drivers and hide the package validation in Figure 3.

Once a NetApp has been transferred (actions 9 and 10), the Transfer notifies the Package Installer in order to install the NetApp (action 11). After, Package Installer gets required commands from the repository (action 12), then, it sends these commands and parameters to each compatible PVN node (action 13). Using the NetApps Configurator, an end-user may issue input commands to adapt and tune a NetApp execution to a specific PVN environment (actions 14, 15, and 16). It is important to highlight that input commands and parameters are provided by NetApp developers. The Data Collector requests

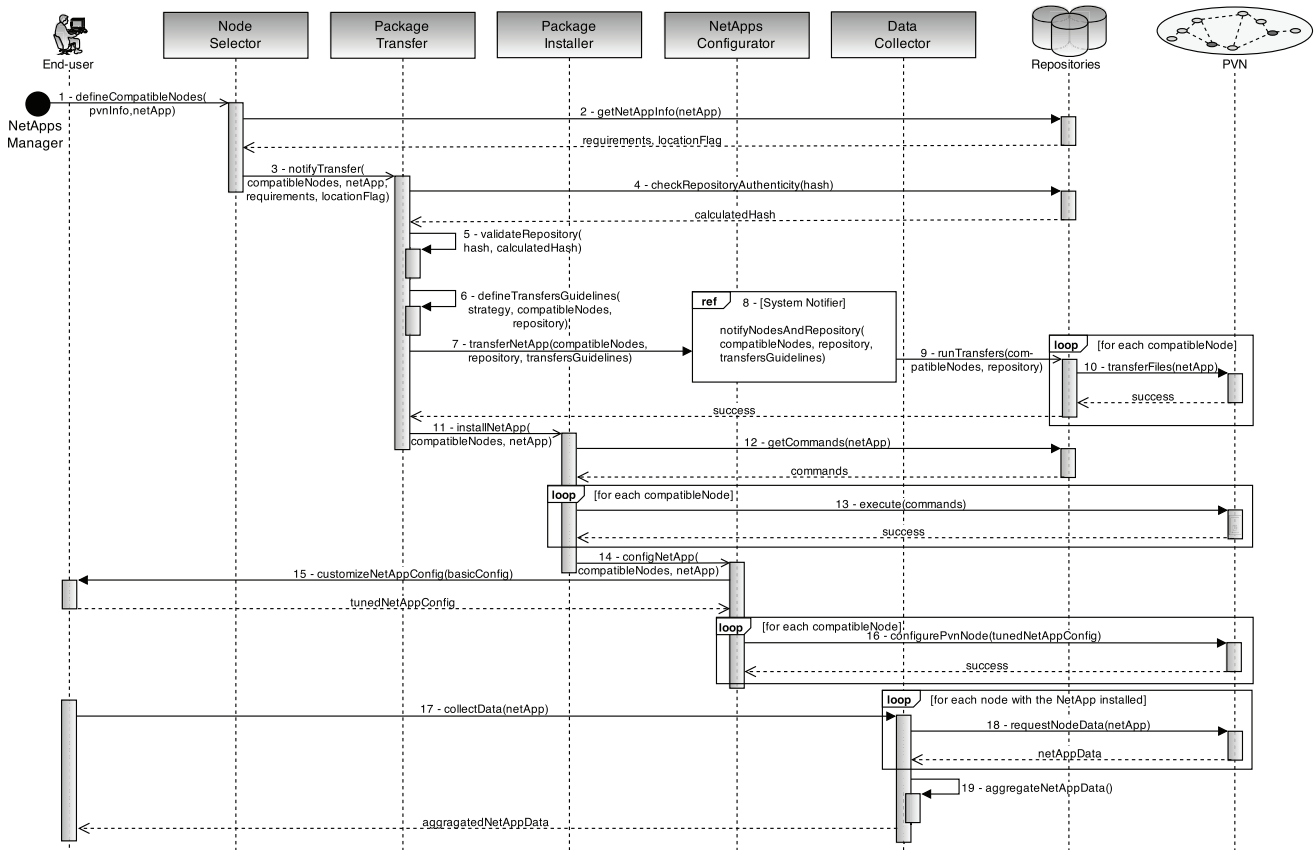

Fig. 3: Actions performed by NetApps modules in order to install and configure a novel NetApp.

(action 18) and aggregates (action 19) the NetApps data from all nodes, allowing the end-user to check the correct NetApps execution and performance statistics (action 17).

The System Notifier encodes and sends all messages from App2net components to PVN nodes. For instance, the exchanged messages contain data about: (*i*) package transfers, (*ii*) priorities of nodes, (*iii*) used strategies, (*iv*) NetApps management; and (*v*) repository and package validation. These messages are encoded in the JavaScript Object Notation (JSON) format. We use this format because it is lightweight and simple, allowing quick exchange of messages.

The PVN nodes and repositories must have a *driver* installed, which is coded and adapted to the features of each device. Thus, different drivers are needed per specific technology, such as Vyatta, Cisco, Juniper, and POX. The set of used drivers forms the Node Communicator. This Communicator carries out the following functions on PVN nodes: (*i*) executing commands to install, uninstall, and configure NetApps, (*ii*) collecting data from NetApps, (*iii*) executing commands to manage NetApps execution, (*iv*) conducting the communication among nodes, (*v*) answering to requests from App2net; and (*vi*) managing nodes registration and lists of transferences. Note that if a PVN node is unable to host a driver, App2net can host the driver on behalf of such node. Thus, communication between App2net and nodes will be guaranteed, even though delays might be significantly longer.

IV. Taxonomy and Code Transfer Techniques

Different research areas have used several techniques to perform the code transfer [13] [14] [17] [22]. However, these techniques receive several names and, there is no study that classifies and compares them. Therefore, we propose a taxonomy that clusters code transfer techniques according to their purpose, allowing to compare and analyze them in the context of PVNs. To elaborate the taxonomy, we identify three main groups: ranking criteria, mechanisms, and strategies.

A. Ranking Criteria

Ranking criteria are metrics and rules used to calculate priorities and sort the PVN nodes, aiming nodes with a high priority receive the NetApp at the first transfers. To establish the ranking of nodes, we consider three criteria: available bandwidth, average delay, and node degree. In average delay, the node with the lowest absolute value has the highest priority. Instead, in available bandwidth and node degree, the node with the highest absolute value has the highest priority.

The *available bandwidth* of a link is the serviceable capacity for data transmission in a time interval [23]. This value is obtained applying Equation 1, in which: (*i*) $A_i^\tau(t_0)$ represents the available bandwidth of link i during a time interval $(t_0, t_0 + \tau)$, (*ii*) C_i is the capacity in link i; and (*iii*) $u_i^\tau(t_0)$ is the average utilization of link i during a time interval $(t_0, t_0 + \tau)$, with $0 \leq u_i^\tau(t_0) \leq 1$. Using this criteria, initial transfers may become faster because bottlenecks are avoided at the beginning of the NetApp distribution process.

$$A_i^\tau(t_0) \equiv C_i[1 - u_i^\tau(t_0)] \quad (1)$$

Delay is the time for a packet to travel across the network from one node (source) to another node (destination). Here,

the *average delay* criteria is the average time (in milliseconds) obtained through the Round-Trip Time (RTT) of packets between source and destination nodes during a time interval. In that way, the NetApp distribution occurs earlier close (short RTT) to the source and/or in idle nodes. Afterward, the NetApp is sent to distant and busy nodes.

According to the graph theory, *node degree* represents the numbers of links connecting a node. Using such criteria, nodes with high degree are privileged and receive the NetApp at initial stage. Subsequently, these nodes are able to send the received NetApp to a greater number of neighbors, making NetApp distribution more agile. We consider that two nodes are neighbors when there is a direct link among them.

B. Mechanisms

Mechanisms define the behavior of NetApp sources, called *repositories*. In PVNs, we consider three mechanisms: push-based, pull-based, and hybrid-based. In the *push-based*, repositories are "*active*", which means that they start the NetApp transferences to nodes. The Package Transfer module creates a list with the initial transfers and priorities of PVN nodes, called *transfers list*. Then, the System Notifier sends this list only to involved repositories. Next, repositories send the NetApp to nodes in the transfers list, according to the priorities of nodes. It is worth noting that nodes with high priority (in the top of list) receive the NetApp at initial stage.

In turn, *pull-based*, repositories are "*passive*", in other words, they just answer requests from PVN nodes. The System Notifier component sends messages to nodes about which NetApps must be installed or upgraded. Also, this component informs the calculated priority for the nodes. Thereafter, each node explicitly requests to the repository a given NetApp. If the repository cannot send the NetApp immediately, the requests from these nodes are stored in a *standby list* ordered by priorities. Then, as soon as possible, the repository sends the NetApp to nodes in the top of the standby list.

The *hybrid-based* uses active and passive repositories. Thus, the repositories manage and use both standby list and transfers list. First, in the initial transfers (*push stage*), repositories send the NetApp directly to nodes in the transfers list (active). In subsequent transfers (*pull stage*), nodes request the NetApp from the repository (passive). Next, if the requests are not answered, they are stored in the standby list and, as soon as possible, the repositories will send the requested NetApp.

C. Strategies and Models

We define strategies as guidelines to control the behavior of PVN nodes during the NetApp distribution process and the number of supported simultaneous transfers. Based on techniques found in the literature, we propose four strategies to transfer NetApps in PVNs: sequential, parallel, gossip, and groups. Once defined and grouped the code transfer techniques (ranking criteria, mechanisms, and strategies), we merge them to build up *models*. A model is composed of one strategy applied in one particular mechanism and using only one criteria for ranking the nodes. In App2net, such models were implemented in the Package Transfer module.

1) Sequential: This strategy simulates a simple script sending a NetApp to the PVN nodes, thus, there is only one transfer at a time. For instance, the repository only sends the NetApp to node B, after completing the transfer to node A. This is repeated until all nodes have the NetApp. The implemented models in this strategy have the following specifics. In push-based mechanism, after receiving the transfers list, the repository sends the NetApp to each node, starting a new transfer only when finishing the previous one.

In the pull-based, the App2net informs the nodes about the NetApps needed as well as the priority of each node. Then, the nodes request such NetApps to the repository, which answers only one node at a time. As previously described, when the repository cannot immediately answering, the requests from nodes are storing in a standby list ordered by priorities. When a transfer finishes, the repository selects the node with the high priority to send the NetApp.

In the hybrid-based, the App2net sends a transfers list to the repository. This list includes just half of the nodes, which have higher priority. In the push stage, the repository sends the NetApp directly to each node in transfers list. Meanwhile, the App2net informs the remaining nodes about the NetApp needed. These nodes request the NetApp to the repository, which stores all download requests in the standby list. After completing the push stage, the repository starts the pull stage and, thus, answers the download requests from nodes in the standby list according to the priorities.

2) Parallel: In this strategy, the repository simultaneously sends the NetApp to multiple nodes until reaching a threshold called *quota* that is calculated by Equation 2, in which: (*i*) n is the amount of PVN nodes, (*ii*) $d(node_i, node_j)$ denotes the length of the shortest path in hops between $node_i$ and $node_j$; and (*iii*) $node_i$ and $node_j$ are nodes of a connected graph. This equation calculates the average shortest path length, a classic network topology measure. Such measure allows adapting the quota to amount of nodes and distances. Thus, there will be a high quota on PVNs with many distant nodes.

$$quota = \sum_{i \neq j} \frac{d(node_i, node_j)}{n(n-1)} \tag{2}$$

In the push-based, after receiving the transfers list, the repository sends the NetApp to *x* nodes at the same time. Note that the *x* is the limit defined by the quota. Otherwise, in the pull-based, the repository accepts *x* simultaneous resquests from nodes. Thereby, *x* nodes download the NetApp in parallel, meanwhile the remaining requests are stored in the standby list.

The hybrid-based is a mix of the two previous behaviors. In the push stage, the App2net sends to repository the transfers list including just half of the PVN nodes. Then, the repository simultaneously sends the NetApp to *x* nodes. After completing a transfer, the repository sends the NetApp to the node with the highest priority from the transfers list. Meanwhile, the repository waits for download requests, storing them in the standby list. When the pull stage starts, the repository sends the NetApp towards up to *x* nodes in parallel. Again, when finishing a transfer, the repository answers the request from the node that has the highest priority in the standby list.

3) Gossip: This strategy initially behaves as the parallel one, in which, simultaneous downloads are limited by the quota. However, nodes that receive the NetApp become *ghost repositories*. A ghost repository sends the NetApp to the remaining nodes, working like a extra source of the NetApp. Thus, the number of sources is incremental, optimizing the NetApp distribution over time. Also, this strategy introduces a shared list in the original repository and local lists on ghost repositories, allowing multiple sources at the same time.

In the push-based, the repository sends the NetApp to nodes in transfers list, limiting the parallel transfers by quota. When receiving the NetApp, the nodes become ghost repositories. These nodes check which of their neighbors are in the shared transfers list, located in the original repository. Then, each ghost repository includes in its local transfers list all neighbors that still have not received the NetApp. Next, the ghost repository sends a deletion request to the original repository to remove its neighbor from the shared transfers list, which must be confirmed. Posteriorly, the ghost repository sends them the NetApp, observing the quota computed.

In the pull-based, after becoming a ghost repository, the nodes create a local standby list, considering only neighbors still remaining in the shared standby list. Ghost repositories then send a message to nodes in their local standby lists informing about the NetApp available. If a node answers with a download request, the ghost repository sends the NetApp away. Meanwhile, the node sends a deletion request to the original repository to remove it from the shared standby list. However, if a node answers with a busy notification, it is removed from the local standby list. A busy notification means that a node is downloading the NetApp or the transfer was completed.

The hybrid-based has minimal variations regarding to previous models. The essential difference is that the original repository only sends the NetApp via push-based calls, whereas the ghost repositories only send the NetApp through pull-based calls. It is worth noting that ghost repositories do not send deletion requests to original repository. Only nodes send such request to original repository. In addition, the push and pull stage happen simultaneously.

4) Groups: In this strategy, PVN nodes are divided into groups, in which the group members must be interested in the same NetApp. The number of groups is equals to the quota (see Equation 2). Each group has a unique master that is selected according to the ranking criteria. After the Package Transfer chooses master nodes, the System Notifier informs remaining nodes, called *slaves*, about this selection. Regardless of the ranking criteria used, slaves register in the nearest master, identified by the lowest average delay (short RTT).

Once groups are formed, the original repository sends the NetApp to master nodes that become ghost repositories. Then, slave nodes can download the NetApp only from their selected masters (parallel downloads are limited by the quota). Pull-based and push-based mechanisms of Groups and Gossip behave similarly. The main difference is that slave nodes only receive the NetApp and will not become another ghost repository. In the hybrid-based, the original repository transfers the NetApp to master nodes by push-based calls. After that, masters receive requests from slaves to send the NetApp using pull-based calls.

V. Prototype and Experiments Analysis

In this section, we detail the App2net prototype and the features of the test environment. Subsequently, to demonstrate the feasibility of prototype, we present and discuss the results obtained for code transfer using App2net.

A. App2net Prototype and Test Environment

The App2net prototype was implemented using the Django 1.4.3, Python 2.7.3, and PostgreSQL 9.1.6. Such prototype was installed on a server with four AMD Opteron 6276 processors (16 cores per processor) 2.30 GHz and 64 GB of RAM, running Ubuntu Server 12.04.3. The experiments were performed on 51 virtual machines (VMs) hosted on the server described above. These VMs were created with two 2.3 GHz cores, 1 GB of RAM and 1 Gbps network interface, using different EEs: Vyatta Core 6.6 R1 and VyOS 1.0.4. Furthermore, the hypervisor QEMU-KVM 1.0 and Open vSwitch 1.10.2 was required to host and connect these VMs. The Aurora [20] was used to manage the virtual infrastructure.

We organized virtual nodes according to the three representative Internet topologies formalized by Kamiyama *et al.* [24]: *Ladder*, *Star*, and *Hub & Spoke*. For these topologies, we generated samples using the IGen tool. Each sample has 50 EEs and one repository, which stores only one NetApp. Also, we divided the VMs into three groups simulating their geographic distribution. Thus, the connections among groups were configured with limited bandwidth rate of 100 Mbps and delay of 16 ms, 24 ms, and 50 ms with a variation of 5 ms (as observed in tests among main network spots in our country).

To be realistic about the size of NetApps, we analyzed a sample of applications implemented with different network programmability technologies. In such analysis, we concluded that the size of most of NetApps varies between 100 KB and 1 MB. However, some network services and firmware present sizes up to 12 MB. Thus, we considered the following NetApps sizes: 100 KB, 500 KB, 1 MB, 5 MB, and 10 MB.

B. Experiments and Analysis

In this and the following experiments, we took 30 measurements with 95% confidence level. Therefore, all the following graphs present the average values and standard deviation. Based on service lifecycle stages [25], we analyze three main requirements: good distribution time, minimal resource consumption, and minimal communication overhead. First, we analyze the performance of the App2net in each topology. Then, we employ as evaluation metric the time elapsed until all EEs get the NetApp, which we call NetApp distribution time.

Figure 4 depicts the results obtained by models using hybrid mechanism and delay criteria in Ladder. These results reveal that Parallel has a good distribution time for small NetApps (100KB), but, this time increases for large NetApps (5MB and 10 MB). Meanwhile, Gossip gets the shortest times for almost all file sizes. The different results of Gossip and Parallel occur because for small NetApps the central repository is enough and transfers so fast that EEs have no time to act as repositories. However, as the package size increases, Gossip offers additional *ghost repositories* that minimize delay and

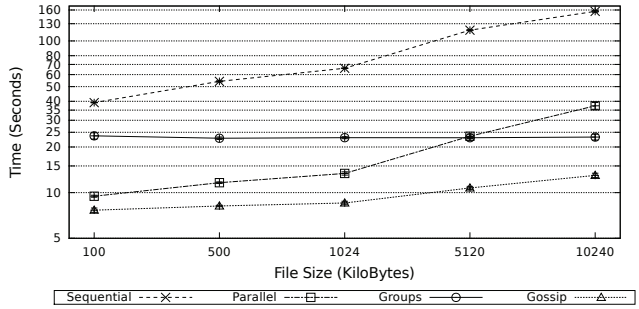

Fig. 4: Strategies in Ladder Topology (hybrid-delay).

prevent the central repository from overloading. Still, Groups obtains similar times in all package sizes, mainly because each master must sends in parallel the package only to the nearest devices, preventing congestion of links and delays.

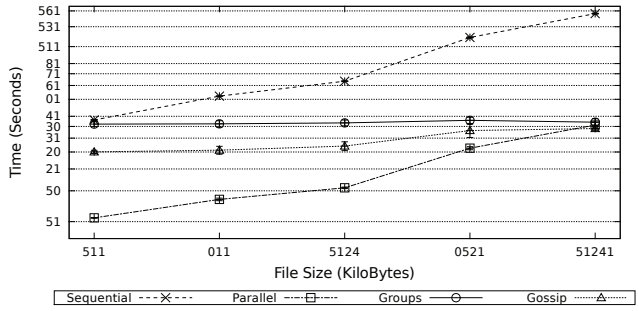

Fig. 5: Strategies in Star Topology (hybrid-delay).

Figure 5 depicts the results obtained by models using hybrid mechanism and delay criteria in Star. These results reveal that NetApp distribution times in Star are longer than or equal to the ones obtained for Hub & Spoke and Ladder, corroborating that the PVN topology influences the NetApp distribution process. In Star, the low number of links and the centralization, in which only one node interconnects various branches, cause congestion and delay incurring poor performance. Analyzing the results, the Gossip strategy loses the advantage it showed over others. As there is only one path to reach most EEs Gossip cannot avoid delays and congestion. Similarly, topology issues also hinder other strategies explaining the similarities among the Parallel, Groups, and Gossip for large NetApps (10MB).

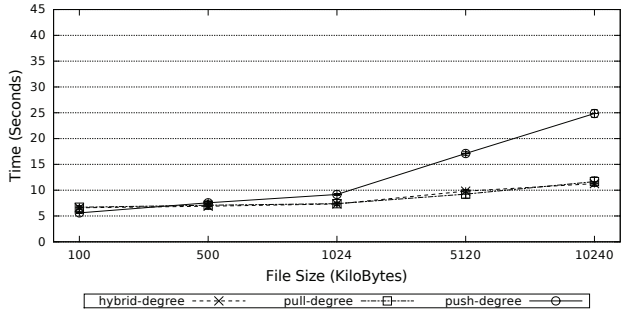

Fig. 6: Gossip Strategy in Hub & Spoke Topology (degree).

Figure 6 depicts performance results for Gossip in Hub & Spoke with degree criteria, revealing that all mechanisms attain similar distribution times for small files. As the file size gets larger than 1MB, the *push-based* behaves increasingly worse while the *pull-based* and *hybrid-based* show a moderate time increase. Figure 7 reveals that the *push-based* in Star performed pretty well for small files and poorly for larger ones. This happens because the nodes actively transfer the file to their neighbors. In contrast, *pull-based* and *hybrid-based* started with a longer time to transfer small files but show good performance for the larger ones. This occurs because devices now have to act as repository and communicate changes in the standby list in addition to their regular routing tasks.

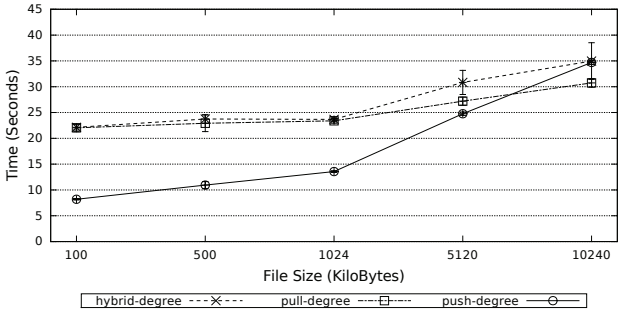

Fig. 7: Gossip Strategy in Star Topology (degree).

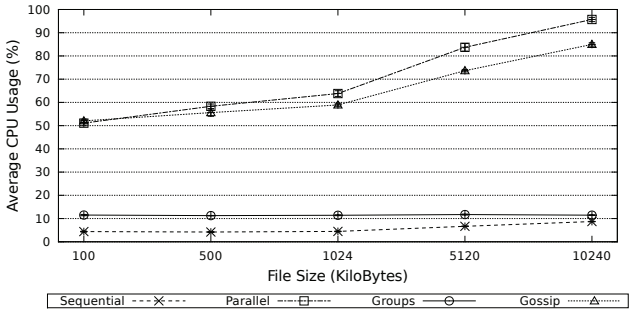

Fig. 8: Average CPU Usage of the Repository (push-delay).

We also analyze the resource consumption (CPU usage), considering push-delay (repository) and pull-links (PVN nodes) models. In the original repository, Parallel and Gossip (Figure 8) show a great and incremental average CPU usage. Whereas Groups and Sequential have a small and steady CPU usage for all NetApp sizes. It results out of required computing to control and perform parallel transfers for PVN nodes. Groups CPU usage is smoother because the original repository transfers the NetApp just to masters. In PVN nodes (Figure 9), Gossip obtains a higher CPU usage than others strategies, because the PVN nodes become ghosts repositories after receiving the NetApp. In Groups, the small CPU usage of slaves hide the high consumption of masters, minimizing the average. In Sequential and Parallel, each PVN node performs its transfer and interacts only with the original repository, resulting in a small average CPU usage.

We measure the communication overhead on the total network traffic generated by each strategy, considering 50 nodes and the NetApp of 100KB. Charts were omitted for traffic overhead measures due to space limitations. As expected, the overhead remained almost the same for most of the analyzed models, being observed some variation only by changing the used mechanisms. Gossip registered the highest observed overhead 2085.75 KB, which represents next to 5.07% of the total traffic generated. Next, Groups obtained

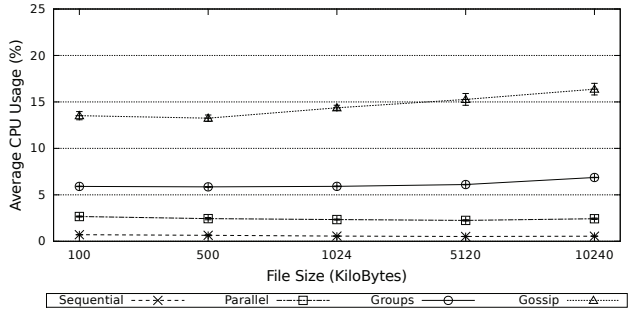

Fig. 9: Average CPU Usage of PVN Nodes (pull-links).

1875.31 KB (≈ 4.44%) of overhead, whereas, Sequential and Parallel achieved 1567.87 KB (≈ 2.87%) and 1458.46 KB (≈ 2.72%) of overhead, respectively. Although Gossip and Groups have obtained a greater overhead, these strategies showed a reduction in total traffic generated. This is result of the new repositories created by these strategies along the NetApp distribution process.

VI. Final Remarks and Future Work

In this paper, we have introduced App2net that empowers end-users to transfer, distribute, and configure NetApps in PVNs with heterogeneous EEs. We also analyzed a set of code transfer techniques and proposed a taxonomy for grouping them. Futhermore, we presented models for code transfer merging some of these techniques.

We evaluate App2net considering two different EEs (Vyatta and VyOS). The evaluation results clearly disclose the influence of end-user's network topology in the distribution times of NetApps over PVNs with heterogeneous EEs. We found that the behavior of strategies was similar in both Hub & Spoke and Ladder topologies. In these topologies, Gossip with hybrid and pull mechanisms obtains the best distribution times in most of the tests. With degree as criteria, nice distribution times were achieved for Hub & Spoke, whereas, for Ladder available bandwidth performs better. In Star, the best performance is obtained with the Parallel strategy and push mechanism using degree and available bandwidth criteria.

We also evaluate the average CPU usage and network overhead generated by each strategy. Although Gossip obtains good distribution times, it requires a significant CPU usage of both original repository and PVN nodes. Moreover, in one hand, Gossip and Groups strategies show a high network overhead. On the other hand, these strategies reduce the total network traffic. Parallel and Sequential strategies both require low CPU utilization from PVN nodes. However, despite Parallel achieving the best distribution time for Star, it still imposes a high average CPU usage at the original repository. In summary, there is no single best strategy for good distribution time, minimal overhead, and minimal resource consumption. It is on the account of the end-user to choose the best solution and trade-offs for each environment.

As future work, we intend to: (*i*) bring concepts from the graph theory for ranking nodes, such as eccentricity, closeness, and betweenness, (*ii*) automate the choice of distribution models based on topology supplied by the end-user; and (*iii*) verify the dependencies and conflicts among NetApps.

References

[1] A. T. Campbell *et al.*, "A survey of programmable networks," *Comput. Commun. Rev.*, vol. 29, no. 2, pp. 7–23, apr 1999.

[2] D. Tennenhouse *et al.*, "A survey of active network research," *Communications Magazine, IEEE*, vol. 35, no. 1, pp. 80–86, Jan. 1997.

[3] D. Chess, C. Harrison, and A. Kershenbaum, "Mobile agents: Are they a good idea?" in *MOS'97*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1997, vol. 1222, pp. 25–45.

[4] N. Chowdhury and R. Boutaba, "Network virtualization: state of the art and research challenges," *Communications Magazine, IEEE*, vol. 47, no. 7, pp. 20–26, July 2009.

[5] A. Clemm, R. Wolter, and J. Kelly, *Network-embedded management and applications understanding programmable networking infrastructure*, 1st ed. NY, USA: Springer, 2013.

[6] S. Kiran and G. Kinghorn, "Cisco Open Network Environment: Bring the Network Closer to Applications," 2013, Available at: https://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9902/white_paper_c11728045.pdf. Accessed: August, 2013.

[7] N. McKeown *et al.*, "Openflow: enabling innovation in campus networks," *SIGCOMM*, vol. 38, no. 2, pp. 69–74, mar 2008.

[8] NFV ISG, "Network Functions Virtualisation: An Introduction, Benefits Enablers, Challenges and Call for Action," European Telecommunications Standards Institute, Tech. Rep., 2012.

[9] R. Dantu, T. Anderson, and R. Gopal, "Forwarding and control element separation (forces) framework (rfc 3746)," United States, 2004.

[10] H. Yeganeh, M. Shakiba, and M. Samie, "Optimal resource allocation in next generation network services using engineering optimization with linear constraint particle swarm," *IJCSNS*, vol. 8, no. 11, p. 328, 2008.

[11] B. M. Edwards *et al.*, *Interdomain Multicast Routing: Practical Juniper Networks and Cisco Systems Solutions*. Addison-Wesley Prof., 2002.

[12] H. Egilmez, S. Civanlar, and A. Tekalp, "An optimization framework for qos-enabled adaptive video streaming over openflow networks," *Multimedia, IEEE Transactions on*, vol. 15, no. 3, pp. 710–715, 2013.

[13] R. Zabolotnyi *et al.*, "Dynamic program code distribution in infrastructure-as-a-service clouds," in *ICSE*, 2013, pp. 29–36.

[14] L. Vanbever *et al.*, "Hotswap: correct and efficient controller upgrades for software-defined networks," in *2nd SIGCOMM Workshop HotSDN*, NY, USA, 2013, pp. 133–138.

[15] V. A. Olteanu and C. Raiciu, "Efficiently migrating stateful middleboxes," *Comput. Commun. Rev.*, vol. 42, no. 4, pp. 93–94, aug 2012.

[16] A. Tootoonchian and Y. Ganjali, "Hyperflow: a distributed control plane for openflow," in *INM/WREN*, 2010.

[17] M. M. Nawaf and Z. Torbey, "Replica update strategy in mobile ad hoc networks," in *MEDES*. NY, USA: ACM, 2009, pp. 474–476.

[18] OpenStack, "Open source software for building private and public clouds," 2011, Available at: http://www.openstack.org/. Accessed: August, 2013.

[19] Eucalyptus, "Open Source Private Cloud Software," 2008, Available at: http://www.eucalyptus.com/. Accessed: August, 2013.

[20] J. A. Wickboldt *et al.*, "Resource management in IaaS cloud platforms made flexible through programmability," *Computer Networks*, vol. 68, pp. 54 – 70, 2014.

[21] G. P. Koslovski, P. V.-B. Primet, and A. S. Charao, "VXDL: Virtual Resources and Interconnection Networks Description Language," in *Networks for Grid Applications*, 2009, vol. 2, pp. 138–154.

[22] J. Payne, S. Shaio, and A. Van Hoff, "Method for the distribution of code and data updates," Jul. 6 1999, uS Patent 5,919,247. [Online]. Available: http://google.com/patents/US5919247

[23] M. Jain and C. Dovrolis, "End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput," *SIGCOMM*, vol. 32, no. 4, pp. 295–308, aug 2002.

[24] N. Kamiyama *et al.*, "Impact of topology on parallel video streaming," in *NOMS*, 2010, pp. 607–614.

[25] OGC, *Information Technology Infrastructure Library 3.0*. London, UK: OGC, 2007.