# Lessons from Operating Systems for Layering and Abstractions in 5G Networks

Stuart Clayman and Daphne Tuncer
Dept. of Electronic Engineering, University College London, London, UK
Emails: s.clayman@ucl.ac.uk, d.tuncer@ucl.ac.uk

***Abstract*—The trend of softwarization within networks, such as SDN, NFV, and 5G, has increased the need for a network operating system and all its associated elements. In this paper we discuss how the layering and abstractions that commonly appear within computer operating systems are missing within the domain of networking. As such, the lack of abstractions means that interacting with and managing networks has already become a difficult and sometimes cumbersome task. The softwarization process is magnifying this effect. The abstractions that appear in operating systems hide the underlying features of the hardware, presenting elements to programmers and system managers that are easier to understand and to interact with. In this sense, the networking world is far behind. We present some of the lessons learned from these operating system abstractions and consider what could appear in the networking world.**

## I. Introduction

Historically the world of networks and the world of computing have developed separately from each other, but have been connected to each other – computers are connected to other computers over networks. Due to this historical separation, the management of computers is done differently from the management of networks. In recent times, the development and deployment of cloud computing [1] and Software Defined Networking (SDN) [2] and Network Function Virtualization (NFV) [3] has brought both worlds much closer together. This has created a much tighter coupling of computers and networks as one environment, but it has also highlighted the differences in approach of usability and management.

Recent efforts in industry, trade groups, and academia have been addresing multiple 5G technologies, with a particular interest in network softwarization [4], providing new software elements that can be folded into the domain of networking, creating many initiatives such as SDN, the ETSI NFV model [5], Service Function Chaining (SFC), conferences such as Netsoft and IEEE SDN-NFV, and much open source software. One of the major issues still to be addressed, is how the arenas of orchestration and higher level management can be combined and how all of these software elements can be designed and structured to create a working system. An even broader vision is how to create the equivalent of an operating system for the network, not just for one host.

To aid in this vision, this paper presents the layering and abstractions that commonly appear within operating systems and are missing within the domain of networking, but will be necessary for 5G management to be effective. Although layering is common in networks, *e.g.,* the ISO 7-layer model, it is less common when interacting with and managing devices. Some abstractions are appearing in the networking world, in the form of systems on top of NFV, SDN, OpenFlow, but more will be needed in order to address the new 5G aspects [6] of programmability and softwarization, of management and orchestration, and of network slicing.

978-1-5090-0223-8/16/$31.00 

This paper is focused towards the areas of 5G management where the role of slicing, SDN, NFV, and SFC is primary and approaches are needed for effective and appropriate solutions. The material presented is not about the right or wrong way to do management, it is more that there are opportunities to use working and well tested concepts. We present a perspective from the viewpoint of operating systems and programming languages. Overall, we wish to encourage people to design / built / utilize more in the area of abstractions, by showing the successes in other areas.

## II. Background

This work came about from discussions with networking people, telecoms operators, DevOps in recent EU projects and at the IETF. They all mentioned how difficult it is to interact with the complex systems they have, and how difficult it is to deploy a new service. Many operators have a goal to reduce service deployment from 90 days down to 90 minutes. Although NFV and SDN are showing good promise in this area [7] [8], overall there are few mechanisms to make this really happen at scale. From our perspective, there are not enough useful abstraction models and not enough abstraction layers.

The lack of layering and common abstractions in the network management means that interacting with and managing networks has become a difficult and sometimes cumbersome task. It is still common for network operators to write *scripts* that interact directly with specific devices. However, these scripts are written to send instructions to a machine – a router. If an operator has routers from Cisco and Juniper, there might be two versions of the script. Any changes will have to be made to both of the scripts. This is a non-trivial task, given the number of commands and attributes, and considering that the manual for Cisco IOS alone is over 1200 pages. The introduction of SDN by using Openflow switches [9] has exacerbated this problem [10] as there are now extra devices in the network that are controlled using yet another approach.

Networking currently has few common abstractions, where we see that nearly everyone is trying to talk to the devices directly. The appearance of Openflow has however opened up some opportunities and some realisations in this area, and has encouraged many to look at SDN in a broader way than just those features provided by Openflow. Many researchers

are looking at how SDN, Openflow, and SDN Controller can be utilized more effectively and expanded to create far more dynamic environments.

Operating Systems have many abstractions over the devices in the machine and the controller for the devices. Many of these were originally devised in the 1960s and 1970s, so there is a lot of experience as to what function to put in what location. The abstractions that appear in operating systems hide the underlying features, operations and interfaces of the hardware, by presenting elements to programmers and system managers that are easier to understand and to interact with [11]. The operations on the abstracted elements are queued, mapped, processed and multiplexed, through various layers, into control and data requests for the devices. The top level interfaces of the operating system do not interact directly with the devices.

Furthermore, there is a need to express operations on these abstractions in order to make them function. The expression of these operations is done through the use of high-level programming languages, of which there are many. They have different syntax structures and different semantics, but what they have in common is that they map to the underlying computer. This mapping is done via a compiler or an interpreter, which converts statements in the high-level language into a set of instructions for the device - namely the computer.

However, one important complexity that the domain networks has to deal with is the distributed and connected nature of the elements. To address this complexity, SDN utilizes the concept of the centralized controller. The use of single entity, centralized management system brings about some design and implementation simplicity as everything is in one place, but in a networked environment such an approach is not realistic. The side effects of centralized design include a growing code base, a limit on the amount of resources available in a single place, which bring about a stasis within the system, as well as the unpredictable delay from the remote network entities.

## III. Layering and Abstractions

In this section we present some of the layers and abstractions that an operating system has. Of particular note is that none of these abstractions are manifestations of features that are present in the hardware. Conversely, there is no, or very little, hardware support for these abstractions. Each of the layers provides a mapping from one set of functions to another. Most operating systems are designed around a set of architecture principles which guide the overall structure. In [12], the ideas of modularity and the use of software tools which can work together, the separation of concerns within each module, and the concept of *do one thing and do it well*.

In Figure 1 we see the layers and abstractions present in the Linux operating system, a UNIX like OS. It shows the 6 main functional areas of the system, namely: system, processing, memory, storage, networking, and human interface; as well as the layers, which present the interface to all of the functions of the operating system. In particular, there are 6 functional layers between the main API interfaces, the *user space interfaces*, and the *hardware interfaces* and the devices themselves.

To understand the value of these layers and abstractions we consider three of them and see the devices they eventually

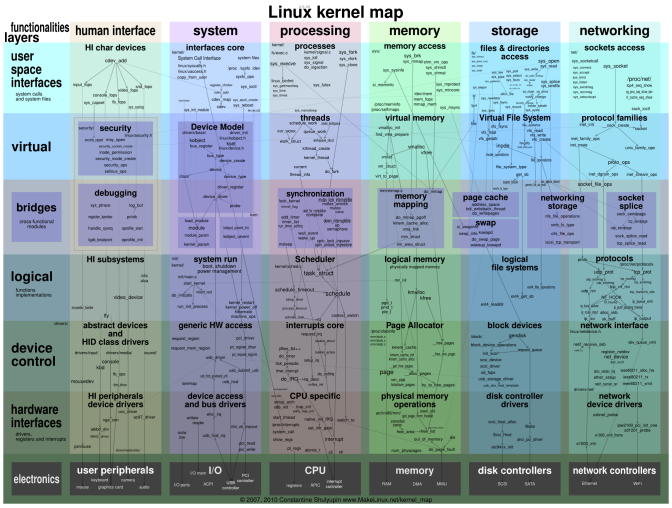


Fig. 1. Linux Operating System Layers and Abstractions

map down to. They are: *processes* which map to the CPUs & memory; *the filing system* constituting files & directories which map to the discs; and *sockets* and the high-level networking which map to the network interface cards. We also look at the abstraction that *distributed systems* present.

### A. Processes

A process is a manifestation of a program that executes on the computer. It is independent of other processes, but can interact with other processes. Processes have some support in hardware, but it is not a manifestation of a processor. Modern computers have hardware for: (i) memory management tables, which maps a process addresses (from zero upwards, for each process) into a real physical addresses of the underlying memory; and (ii) a single bit in the CPU to say whether the code running is in kernel space (part of the OS) or in user space (a process).

The process is an abstraction, independent of the hardware or CPU type. The process itself can be considered (and also written / developed) without knowing anything about the physical resources of the computer, how many other processes there are, or what state the operating system thinks the process is in. All of this is handled automatically by the *process scheduler*. This scheduler function of the OS decides which process to execute next. The operating system schedules each process depending on whether it is suitable to execute and whether is should be allocated some CPU time. From the human perspective they execute at the same time: this was devised in the 1960s and is called *time sharing* [13].

A process itself may be made up of 1 or more threads, where each thread does a function of the whole program, as in Figure 2. Every process will have some allocated working memory, and each will see its memory address space start from 0, even though it will not really be located in address 0 in physical memory. Memory allocation per process, and for all of the processes in the system, is not fixed to the maximum size of physical memory. It is done dynamically, using virtual memory and an over-provisioning strategy. This memory space of a process is split into 3 segments: code, stack, memory for data, and this pattern is applied to each thread, which will also be

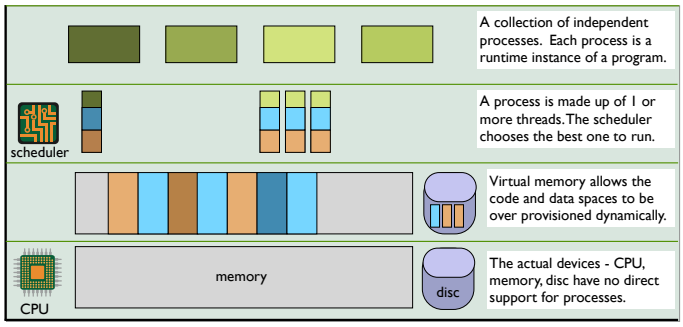


Fig. 2. Processes

composed of the 3 segments. Each segment is broken up into a set of *pages* which are smaller chunks of memory directly managed by the memory management hardware. This virtual memory and memory management technique was started in the 1950s and solved in 1960s [14].

### B. Filing System

The end-user of a computer sees their data stored in files and directories. However, the disc drives that actually store all of the relevant data, have no concept of files or directories or filing systems. Each of these files & directories are part of a big tree which is a single uniform abstraction that is mapped down to specific storage devices in the system.

This tree is the filing system that spans across all the storage devices in the system. It presents a virtual file system that is made up of individual logical file systems in underlying devices, as in Figure 3. The logical filing system will have its own different format for layout and structure of files and directories. A file, which can be viewed as a stream of bytes in the application, will be manifested in the file system and on the disc as an ordered collection of blocks. The logical file system layer will map this ordered collection to blocks with locations. The same layer is responsible for the opposite task of collecting the blocks with locations, and recreating the streams that represents a file. The locations of the blocks do not need to be consecutive, as this layer deals with that mapping.

The blocks themselves are numbered and cached within the OS. Another layer, the device driver, is responsible for interacting with each type of storage device, and dealing with the different control messages that are appropriate for the

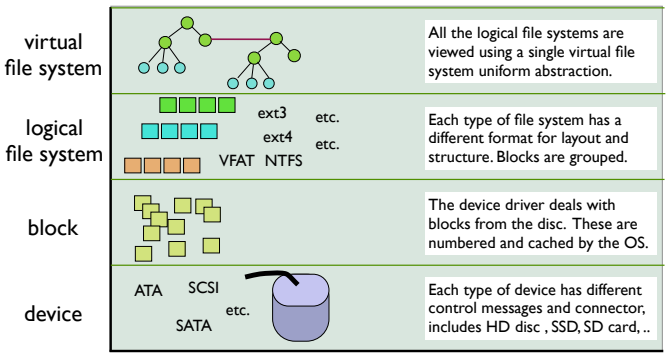


Fig. 3. Filing Systems

different kinds of disc, such as: hard disc, SSD, SD card, *etc.,* and getting the blocks on and off of the disc. It also deals with issues related to the low level device interfaces and connector type such as: SATA, SCSI, ATA. Using this layered and abstraction approach, there are various file system formats and different storage devices that can exist on the same box. As discussed earlier, there are layers where elements (blocks in this case) are queued, mapped, processed and multiplexed. This work started in the late 1950s / early 1960s, and by 1964 the notion of a file system was in general use [15].

Due to the nature of the layering and abstractions devised for managing file sytems on discs, it is actually possible to have modules that interact with any kind of element that presents a tree and have this presented as part of the overall virtual filing system view. Examples of this flexibility include modules that connect to remote ftp sites and present them as though they were local, or modules that can present a zip file as part of the filing system. This is also how network filing systems are done. NFS [16] does not talk directly to storage devices on the same computer. Rather, there is a component that uses a protocol (RFC7530) to request the blocks from a remote server. In this way, we have a distributed system, where the higher levels are unaware of the lower levels doing networking, as there is a separation of concerns.

### C. Sockets / Networking

The networking layer of most modern operating systems is presented using TCP/IP. This is accessed via the *socket* API. A socket is an abstraction over the transport layer, and supports operations for sending and receiving data. From the telecoms perspective it is not always possible to observe how this layer is put together, and how this abstraction impacts a programmer's view of the network.

The implementation of TCP/IP gives the user two kinds of network interaction: (i) UDP – an unreliable datagram delivery mechanism; and (ii) TCP – a reliable stream delivery mechanism, both accessed via two different kinds of socket.

The use of UDP, whereby each piece of data presented to a UDP socket will become one network packet, is used for particular kinds of application and is a simple model. Data sent via UDP can be lost during transmission, and there is no built in mechanism for notification or retries.

This differs from TCP, where we can consider that TCP itself is another abstraction over the network. To the user it presents a reliable stream, and to the network it sends packets – eventually. Each piece of data presented by the user to a TCP socket stream will become many packets at the network level, all of which are carefully managed.

TCP actually has 3 mechanisms:

(i) two byte streams – an input stream and an output stream which can be accessed from either end of the TCP connection,
(ii) a reliable transport mechanism – such that any data loss between the end-points is overcome through re-sending lost data packets,
(iii) a congestion control mechanism – such that TCP can adapt its sending rate, both up and down, depending on how each perceives any congestion in the network.

The packets sent via the sockets are queued both inbound and outbound, and for TCP they can be reordered and grouped as necessary. With TCP or UDP there is no need to specify the bandwidth of the connection in advance, the maximum capacity, or the length of the lifetime of a session. The packets pass from a TCP/IP handling layer down to a device layer, which interacts with the network hardware, and all the traffic is multiplexed onto the network card. The underlying networking hardware supports multiple link layer transmission mechanisms, and can operate using very different devices, including: ethernet, optical, wireless WiFi, bluetooth, etc.

### D. Distributed Systems

In a single system all of the functions and operations are directly accessible. In a distributed system, the functions and the operations within a node need to be chosen and designed. The method of doing this chopping / slicing / separation is non-trivial, but by using well known and documented design philosophies systems can be built. As stated earlier, distributed systems are themselves an abstraction and another layer of functionality. Within any distributed system there are two main factors to consider: (i) the function / operation of a node; and (ii) the interactions between the nodes. In fact, these challenges are not ad-hoc, but have also been approached from a theoretical perspective, for instance, in the Layering Optimization Decomposition framework [17], in which the different steps to modularize and distribute centralized computations are tackled using the mathematical theory of decomposition.

A good example of such a distributed system commonly used on the Internet is DNS. It has the appearance of a single system that can be accessed from anywhere. The system is built as a graph of server nodes that hold information about hostnames and host addresses. This is coupled with DNS clients that can make lookups in order to resolve names to addresses. No node in DNS holds all of the data, and any *client* of DNS can do a lookup. More importantly, DNS is utilized by other distributed systems which need to do name lookups. There is no need to create another name service mechanism. From a distributed systems perspective, it works well.

This leads us into the interactions between the elements of the distributed systems. The interactions are done using well known and well documented protocols. Protocols are an important aspect of distributed systems. Some of them are very old and are still being used: POP3, SMTP, FTP, ARP. The protocol defines interactions between the end-points. It does not define what the end-points are, how big or small they are, what other functionality they do or don't provide HTTP is a good example of this type of interaction. Anyone can write a browser and a server in any language, on any system. There is no need to know anything except the protocol, as it defines the *interactions*. To highlight how important this is, notice the massive expansion of *web services*. None of them were predefined by anyone, they had organic growth by their user base.

Alternate approaches to centralized control, using distributed systems, have been suggested by many. Distributed systems are themselves an abstraction and another layer of functionality. A lot of people in the Network Management community observe that distributed systems are difficult and manifest many new and more complex problems. This is indeed true, compared to a single system, however, it is also an area that has been investigated by computer scientists since 1970s. As one example, consider the large body of work[1] by Leslie Lamport, who has been researching this area since the early 1970s, and his first seminal paper in this area in 1978 called "Time, Clocks and the Ordering of Events in a Distributed System" [18].

*Summary:* We can see that with the right layering and abstractions, the highest level of functionality is presented to the user, and optimal performance has been superceded by reliability, stability, and scalability. Using the abstraction of a process an operating system can reliably execute thousands of processes concurrently. The user does not need to care about the number of processes or when they execute. We also see with files and directories that the filing system layer does not need to talk directly to a disc drive. It has a wide ranging power and flexibility that is not possible with simple device interactions, and can read and write 1000's of file concurrently. Furthermore, the use of the socket abstraction and TCP/IP hides all of the different networking interfaces. Not only can these interfaces exist in the same computer, in much the same way as the file system hides the different storage hardware, but it also manifests the distributed systems abstraction. Imagine how complex it would be for a programmer to specify these things for each application they write.

## IV. Languages

The purpose of programming languages is to express operations in a domain. These operations, at the lowest level, are machine instructions. Originally, assembly languages started as a 1-to-1 mapping of a text representation of an instruction to the instruction itself. A computer executes one instruction at a time, but it take considerable expertise to elaborate and understand how a sequence of machine instructions represents a higher level concept.

Programming languages have evolved from being representations of machine instructions to higher level languages that process abstract elements of the operating system domain. The syntax and semantics of these languages also vary dramatically, however they all eventually specify their high level operations as low level machine instructions. This is done through the use of a compiler, which takes the high level language and creates a sequence of machine instructions, or through the use of an interpreter, which evaluates the high level language at run-time. Both of these approaches were devised late in the 1950s, with more families of languages appearing over time.

Expressing operations instruction by instruction is low level, but this is how much network management if still done. The operations are instructions for a router or a switch. Although these router / switch instructions undertake more work than a machine instruction, in essence the situation and scenario is the same.

The expansion of the different kinds of languages, with their own syntax, semantics, and own abstract elements means that there is now many ways to write the same program. The choice of language for any job can be seen as a complexity itself. Some programmers choose a language because they

[1] http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html

already know it and are expert at it, while others will choose the language in order to participate with other programmers or to fit in with an organization. A subset will try to choose the language where the domain of the problem and the thought processes for solving the problem are highly correlated with the syntax and semantics of the language itself.

There is also a split in language styles whereby some languages, called *procedural*, are designed to be more clearly representative of a sequence of instructions, even though they have high level concepts. Other languages, called *declarative* languages, do not specify sequences of instructions. Just like mathematics, they are an expression or set of declarations in a domain. It is the responsibility of the language compiler or interpreter to take these declarative statements and convert them into a sequence of instructions.

Having the compiler / interpreter and procedural / declarative approaches to specify programs and their execution environments allowed language designers to devise many programming languages from the 1950s onwards, and these expanded dramatically in the 1960s. Diverse languages such as FORTRAN and Lisp were devised in the late 1950, and further differentiation in Algol, APL, SNOBOL, *etc.,* came in the 1960s. Further developments followed in the coming years. We are now in a position where there are many styles of languages. Some of them are presented in Table I, together with the year they were introduced.

| Language Style | Language Name (Year of Introduction) |
|---|---|
| *Procedural* | FORTRAN (1957), COBOL (1959), Algol (1960), Pascal (1970), C (1972) |
| *List* | LISP (1958), Scheme (1970) |
| *Vector* | APL (1964) |
| *Pattern matching* | SNOBOL (1962), awk (1977) |
| *Object oriented* | Simula (1965), Smalltalk (1972), Java (1995) |
| *Logic* | Prolog (1972) |
| *Stack based* | Forth (1970), Postscript (1982) |
| *Rule based* | OPS5 (1977) |
| *Functional* | ISWIM (1966), SASL (1975), Haskell (1990) |

TABLE I. LANGUAGE STYLE, NAME AND YEAR OF INTRODUCTION

Tool sets for creating new languages have been around since the end of the 1970's [12], for example *yacc* allows the creation of a parser for a new language – most of the input is the grammar of the language, and *lex* allows the creation of a syntax analyzer for a new language – most of its input is a specification of the symbols in the language. These and other useful features and processes of a high level language can be applied during the translation process to analyze the input language and also to optimize the output of instructions. There are many analysis and optimization steps that can be taken when converting a language into a set of machine instructions, as outlined in the *The Dragon Book* [19].

## V. DISCUSSION

In order to address the new 5G aspects of programmability and softwarization, of management and orchestration, and of network slicing, allowing a transition from network devises to network functions and virtual network functions, to dynamically adapt the network to meet future demands, to have a programmable network operating system with an interface to the network, and to create a dynamic, configurable, programmable, resilient, safe and cost effective E2E network, there needs to be both system layering and well formed abstractions plus the relevant. languages [6].

We have previously discussed and seen some designs of approaches to these aspects from the world of operating systems and computers. Here we present some elements of work that utilize such solutions within network management, and could be the primary contenders for utilizing and extending in the 5G management domain. To support programmability and softwarization, we need both well formed abstractions, languages to manipulate those abstractions, and a programming model to provide interfaces for network facilities and services, which will enable a high level of automation in service development and deployment processes.

The abstractions needed for 5G include new elements from the *Service layer* such as network slices, the services that run inside those slices, and a representation of the individual service elements; from the *Management and Orchestration* layer needed abstractions are the virtualized elements (the NFVs) of the service, the virtual connectivity, and the graphs that represent the service chains (SFC). There are already data models that have representations for some of the above, but these need to extended to cover all the required aspects, and work is underway in many projects to address this. The next step is the need to map such models into data structures and then formed into libraries of code to act as a basis for progamming at all levels.

We observe from computers and operating systems that using high-level languages to express operations over abstract elements is far more effective that hand coding with low level device instructions. These languages should specify 'what' needs to be done to the high level abstraction. Preferably this would be done at the abstraction level of a slice, or a service for the networking world, and not specify 'how' to drive a router or an Openflow switch. There cannot be many customers who request a service, and then go on to specify: *please set attribute X on port 14 of router R to 0.95*. The customers will express high level requirements of their service. There are many approaches to convert high level expressions into device instructions, and these have been shown to be highly performant in most cases. UNIX has been written in C since the late 1970s, except for a few hundred lines of assembler needed to control certain machine specific features [12].

The networking world can benefit from looking at the large amount of language styles, the large number of techniques for converting high-level languages into machine instructions, and the long history of language development. We can see that devising languages that are close to the instructions of the machine makes thinking about the problem domain and mapping it to the machine far more complex. Writing scripts that interact directly with a router, or having a controller that generates Openflow instructions does work to some extent, but it is clear that it is very limiting. The lesson here is that scripts driving devices should be minimised and replaced with code for high level abstractions.

It is the job of a language tool chain to take a high-level

language with high level operations and create the instructions for the devices. There are many opportunities in the area of 5G management to build such languages, and this has already started where domain specific languages are currently of interest. These are languages where the functions and symbols are specific and focused on the domain that the language is being used. Having abstractions that can be represented in a language and affected will, over time, bring about the kind of results seen in the computer domain. To enable better expressiveness in the SDN arena some languages have been created, such as Frenetic family of languages [20]. Also, efforts such as P4 [21] allow for programs that specify how a switch processes packets, and are removed from the low level instruction approach. A good survey of SDN languages is presented in [22] which highlights that some of them are specific purpose languages, aiming to solve a particular problem, providing specific operations; and others are general purpose languages that allow a more general and wider set of operations.

With respect to management and orchestration, we can observe that many of the papers which introduce new and enhanced network functionality using SDN controllers rely on a simple model of 1 layer above the data plane. Consequently it becomes difficult to mix the functions presented in more than one paper into a single system, as there is no layering, no separation of concerns, and few abstractions. The lesson here is that definitely needs to be work on models and abstractions for building platforms and adding features for SDN control. Furthermore, these abstractions should be devised to be composable from one abstraction layer to the next, so that simple building blocks can be combined in a useful way.

In any domain, a good abstraction is one whereby it can be extended and used in flexible ways not envisaged by the original designers / authors. The work on Abstractions for Software-Defined Networks [23] and vSDN [24] are good examples in networking of a useful addressable abstraction that maps to the lower layers of the network equipment. In [25] the authors provide a comprehensive survey of hypervisors for virtual SDN networks, but the architectures seen are rather simple. More elaborate work was done recently by Zhang et al. [26], where they designed and built a data plane abstraction using a fully virtualized switch. Some progress is observable, but in general we can observe that very few of the operating system features exist in the networking arena, and in this sense, the networking world is far behind.

We at UCL have used the Operating System design principles for our work on Adaptive Resource Management and Control in Software Defined Networks [27]. Its architecture is compatible with the overall SDN model, yet consists of three layers which interact with each other through a set of well defined interfaces. We have also been using these principles for designing a network slicing mechanism and tool set which has been deployed in 5G PPP projects 5GEx [28] and NECOS [29]. Using a modular structure, the frameworks makes a clear distinction between the management and control logic which are implemented by different planes, offering improved deployment advantages. Each of the layers are themselves distributed systems. As we have observed, there is no need to build one big system, distributed nodes are possible, and if you get the protocols and the interactions right, you deal with the complexities that distributed nodes may bring about.

The ONOS initiative [30], have a goal to create a software-defined networking (SDN) operating system for communications service providers. Work in the area of *intent* [31] is a separation of concerns which presents a declarative statement of requirements which are mapped down to implementation, rather than specifying how to do operations. Such approaches go towards the path that was taken in the computer domain many years ago. All of these elements come together for the goal of building a Network Operating System.

## VI. Conclusions

Some of the abstractions and layers in operating systems have been presented. Although it is possible to find potential imperfections and criticisms for each of these abstractions, the number of computers and working deployments highlight that the resulting design, tested and evaluated over 40 years, has benefits and advantages that far out-weigh the disadvantages.

In the networking world there is a concern that having abstraction and layering is too heavy, and cannot get the data rate of the underlying device, or a particular feature of the device is not exposed via the abstraction.

Consider the converse situation from the operating systems perspective: although having all of these functions and the extra code layers requires to manage the abstractions and to do the mappings means that it is difficult, if not impossible, for a single process to drive the computer hardware at it theoretical maximum performance, there is a trade-off by which we have massive gains. The operating system can support hundreds and thousands of processes and a similar number of users on the same box reliably, safely, without unintended interactions. All of these processes can read and write 1000's of files, concurrently, across multiple disc storage devices, again in a seamless and reliable manner. The same processes can also have multiple TCP/IP network connections, without unexpected interactions.

The reality is that the operating system provides a set of features, a level of scalability, a level of reliability, and a level of changeability, that would not be possible having a simple software element interacting directly with the devices. All of this is done through the use of a layer which interacts with the underlying hardware and then having layers of software which implement various algorithms and data structures to map the abstractions to the loer layers.

In summary, we can observe that the more layers that exist with different abstractions, the more opportunities there are to bind in new features. The lessons for designing systems focused on network and service management and orchestration, particularly now that softwarization and programmability is a primary driver, are: (1) devise common and well used abstractions – do not be fixated on the functionality of the device; (2) have layering – whereby different layers can provide different functions and different abstractions; (3) have a separation of concerns – so that not all the software is in one module; and (4) in each module do one thing and do it well.

## Acknowledgement

This work was partially supported by the EU projects: 5GEX – "5G Multi-Domain Exchange" [28] and NECOS – "Novel Enablers for Cloud Slicing" [29].

## REFERENCES


[1] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Cáceres *et al.*, "The RESERVOIR Model and Architecture for Open Federated Cloud Computing," *IBM Journal of Research and Development*, vol. 53, no. 4, pp. 4–1, 2009.

[2] D. Kreutz, F. M. V. Ramos, P. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *CoRR*, vol. abs/1406.0440, 2014.

[3] M. Chiosi, D. Clarke, P. Willis, A. Reid *et al.*, "Network Functions Virtualisation," White paper at the SDN and OpenFlow World Congress, ETSI, Tech. Rep., 2012.

[4] A. Galis, S. Clayman, L. Mamatas, J. Rubio Loyola, A. Manzalini, S. Kuklinski, J. Serrat, and T. Zahariadis, "Softwarization of Future Networks and Services-Programmable Enabled Networks as Next Generation Software Defined Networks," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE, 2013, pp. 1–7.

[5] ETSI-NFV, "Network Functions Virtualisation (NFV) ETSI Industry Group," http://portal.etsi.org/portal/server.pt/community/NFV/367.

[6] 5GPPP, "5GPPP Architecture Working Group - View on 5G Architecture," 2018. [Online]. Available: https://5g-ppp.eu/wp-content/uploads/2018/01/5G-PPP-5G-Architecture-White-Paper-Jan-2018-v2.0.pdf

[7] F. Hu, Q. Hao, and K. Bao, "A Survey on Software Defined Networking (SDN) and OpenFlow: From Concept to Implementation," *Communications Surveys Tutorials, IEEE*, vol. PP, no. 99, pp. 1–1, 2014.

[8] J. Batalle, J. Ferrer Riera, E. Escalona, and J. A. Garcia-Espin, "On the Implementation of NFV over an OpenFlow Infrastructure: Routing Function Virtualization," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE, 2013, pp. 1–6.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[10] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks," *Communications Magazine, IEEE*, vol. 51, no. 7, 2013.

[11] A. Tanenbaum, *Modern Operating Systems*. Pearson, 2008.

[12] B. W. Kernighan, *The UNIX Programming Environment*, R. Pike, Ed. Prentice Hall Professional Technical Reference, 1984.

[13] M. T. Alexander, "Time Sharing Supervisor Programs," May 1969.

[14] D. Sayre, "Is Automatic "Folding" of Programs Efficient Enough to Displace Manual?" *Communications of the ACM*, vol. 12, no. 12, pp. 656–660, Dec. 1969.

[15] American Data Processing, Inc., Detroit, "Disc File Applications: Reports Presented at the Nation's First Disc File Symposium." 1964.

[16] T. Haynes and D. Noveck, "Network File System (NFS) Version 4 Protocol," RFC 7530, Mar. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7530.txt

[17] M. Chiang, S. Low, A. Calderbank, and J. Doyle, "Layering as Optimization Decomposition: A Mathematical Theory of Network Architectures," *Proc. of the IEEE*, vol. 95, no. 1, pp. 255–312, Jan 2007.

[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[19] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[20] N. Foster, R. Harrison, M. J. Freedman, J. Rexford, and D. Walker, "Frenetic: A High-Level Langauge for OpenFlow Networks," December 2010.

[21] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Computer Communications Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[22] C. Trois, M. D. D. Fabro, L. C. E. de Bona, and M. Martinello, "A Survey on SDN Programming Languages: Toward a Taxonomy," *IEEE Communications Surveys Tutorials*, vol. 18, no. 4, pp. 2687–2712, Fourthquarter 2016.

[23] M. Casado, N. Foster, and A. Guha, "Abstractions for Software-Defined Networks," *CACM*, vol. 57, no. 10, pp. 86–95, October 2014.

[24] Z. Bozakov and P. Papadimitriou, "Towards a Scalable Software-Defined Network Virtualization Platform," in *NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium*, May 2014.

[25] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer, "Survey on network virtualization hypervisors for software defined networking," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 655–685, 2016.

[26] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane," in *26th International Conference on Computer Communication and Networks, ICCCN 2017*, 2017, pp. 1–9.

[27] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou, "Adaptive resource management and control in software defined networks," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 18–33, 2015.

[28] 5GEX, "EU H2020 - 5G Multi-Domain Exchange (5GEx) project," 2015, https://5g-ppp.eu/5GEx.

[29] NECOS, "EU-Brazil - Novel Enablers for Cloud Slicing," 2017.

[30] ONOS, "The ONOS™ project," https://onosproject.org.

[31] F. Callegati, W. Cerroni, C. Contoli, and F. Foresta, "Performance of intent-based virtualized network infrastructure management," *Proc. IEEE International Conference on Communications (ICC)*, 2017.