Towards A Truly Autonomous Network

Pierre Imai, Paul Harvey, Tareq Amin

April 24, 2020

Abstract

The communication networks of today can greatly benefit from autonomous operation and adaptation, not only due to the implicit cost savings, but also because autonomy will enable functionalities that are infeasible today. In this paper we motivate the need and present our vision for the autonomous future of networking, the concepts and technological means to achieve it, and the architecture which emerges directly through the application of these concepts. We compare our strategy for autonomy with the efforts in academia and industry and describe the architecture which we intend to realize within the first fully virtualized telecommunication network in existence. We further argue that only a holistic architecture based on hybrid learning, functional composition and online experimental evaluation like ours will be expressive enough and capable of realizing true autonomy within computer networks.

Contents

1 Introduction

The importance and impact of networking and communication systems on our every day lives is already enormous: Virtually every minute of our waking day is in some way influenced (or even controlled) by our phones and "smart" devices[55]. Yet this influence is expected to grow dramatically in the near future, thanks to the expected proliferation of automotive[120], wearable[9,69], and many other emerging IoT-related applications[70] which will permeate every aspect of our existence[66]. This massive transformation of human society naturally depends on ever more advanced communication networks, which enable the near-instant[3,118] transport of massive[68] amounts of data from the tens of billion connected devices[1] expected in the near future[31,42].

The purpose of this paper is to present and motivate our vision for the network of the future, a network which will transform the telecommunications industry as dramatically as the the aforementioned technologies did our lives. Our goal in the long run is to make the network fully autonomous. Specifically, a network that is not only able to cope and adapt to unforeseen events, but also to improve and adapt itself to meet the challenges of the future, by integrating new technologies as they become available, with little or even no human intervention.

While there are numerous initiatives both in the telecommunications industry and academic research that cover important aspects of autonomous networking, we feel that a *holistic* approach is needed, which bridges the gap we perceive between both industry and academy, as well as between different research fields.

We start this paper by motivating the need for a truly autonomous network in section 2, and continue with a discussion of the efforts of the telecommunications industry and related standardization efforts w.r.t. autonomous networks in section 3. Afterwards we discuss the fundamental technological principles that we consider vital for achieving our vision, and outline how they are manifested in our architecture in section 4. We then detail our architecture blueprint in section 5 and explain our approach as applied to real-world use cases in section 6, before concluding with a discussion of our work and an outlook of our future work in section 7.

2 The Need for True Autonomy

The days when telecommunication networks exclusively facilitated communication between human users are long gone. Huawei is not alone in its

[1] The actual numbers seem to be difficult to predict, though[87].

prediction[138] that the Machine-to-Machine communication market will lead to billions devices being attached to telecommunication networks in the near future. The demands this imposes on the physical layers are obvious, but the complexities that need to be addressed on the transport layer and above also grow. Increasingly heterogeneous devices, such as sensors, home appliances, cars, *etc.* are introduced to the network and traditional communication devices, such as "smart" phones, become ever more powerful. Additionally, the devices themselves, the applications and services that these devices enable are rapidly diversifying and, in many cases, have become more latency critical. This is demonstrated in the growing interest of research into applications that require extremely low latency links such as remote surgery, cloud-based gaming and other applications requiring immediate visual or haptic feedback[8].

The short-term industry response is collectively known as *5G*, which includes *virtualisation*[78], *Massive MIMO*[79][2], *edge computing*[59], and *network slicing*[47]. However, although 5G promises higher throughput and lower latency, it also requires more base stations as the coverage area in which this throughput can be achieved decreases[136]. Complex interference and waveform distribution patterns that depend on building materials and shapes, humidity and air pressure might also necessitate adaptation of the antenna tilt whenever the environment changes to maintain optimal service. This issue might become even more apparent for future, even shorter bandwidth, technologies.

Another industry counter-measure is *edge compute*, which promises to drastically reduce communication latency by placing services physically close to the user. This is a radical shift from the current paradigm of providing compute power at a limited number of data centres. Providing all latency critical services at such a large number of locations at once would obviously lead to uneconomical massive over-provisioning. Thus increasingly complex decision-making becomes necessary to balance resource costs against latency demands. Since user demand changes dynamically and is very hard to predict with sufficient accuracy, providers are forced to take increasingly complex decisions *at runtime* to optimize the trade-off between resource costs and latency expectations.

All of these challenges lead to a further intensification of the existing trend towards more-and-more complex planning, adaptation, and optimization based on network load, service demand, *etc.* Such tasks are the tradi-

[2]Multiple-Input and Multiple-Output, or MIMO is a method for multiplying the capacity of a radio link using multiple transmission and receiving antennas to exploit multipath propagation.

tional domain of human engineers and go *far beyond classical automation*. However, engineers are already one of the major cost factors for today's network operators, and their numbers cannot be scaled up to meet the ever increasing demand, especially as increasing costs cannot be offloaded to the user. Without moving towards a fully *Autonomous Network*, this future will not be achievable.

The term *Autonomous Network* itself is becoming somewhat commonplace, but there seem to be different opinions about what constitutes *autonomy*. While efforts are being made by standards bodies and operators to define what the autonomous future should look like, the approaches are, on one side, vague visions for a far future, that lack a clear strategy or path to realization. Or, on the other side, they are focused on the creation or optimization of tools (such as machine learning) and processes for single use cases, *e.g.* base station tilt optimization.

We, however, maintain that one single control or *cognitive* loop (see section 4.2), which addresses a single use case, is insufficient to handle the combined effects of the many concurrent autonomy-related tasks (such as optimization or anomaly detection) that will indubitably be active in the networks of the future – and details of which might not be known at the design time of earlier deployed control loops that will still be active in the network. And since all network entities share the same playing field, they naturally influence each other. Thus a *holistic*, yet *generic* approach that can accommodate yet unknown operational and structural changes is needed. Use case specific approaches inherently necessitate the work of engineers to adapt them to each new new scenario and environment. These approaches thus only achieve *automation*, not *autonomy*, which according to the Cambridge Dictionary is *the ability to make your own decisions without being controlled by anyone else*.

We consider this ability of autonomy as a necessity for future networks. Without it the ever more rapidly changing telecommunications environment would require even more, not fewer, engineers to be employed. They would be needed to adapt and fine-tune technologies and services to the rapidly changing surroundings. This goes contrary to our *goal of a network that is generic enough to accommodate any potential future technology, and powerful enough to adapt and optimize itself for optimal performance under unforeseen (and unforeseeable) conditions*.

In other words, the concepts and realization sketch we will present in this paper are expected to not only meet, but to exceed the highest level of autonomy previously considered the goal of an autonomous network (see fig. 1). Autonomous cars, for example, can be expected to be adapted by engineers if the operational parameters drastically change, such as for flying,

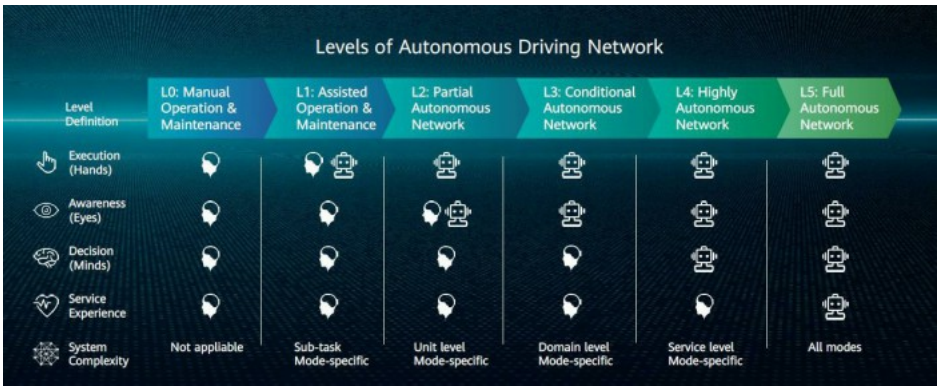


Figure 1: Huawei[60] likens the stages of an Autonomous Networks to the the levels commonly defined for autonomous driving.

submarine travel, or in the case of massive overhauls of traffic rules and conventions. We contend that autonomous networks need to be far more capable, as changes to their operational environment and their functions are more frequent and far more extreme than changes in the automotive environment.

3 Industry and Standardization Efforts

We are not alone in believing in the goal of autonomous networks. Almost every major national mobile network operator, vendor, and even standards organisation is engaged in the topic of autonomous networking. In this section we outline the major approaches and initiatives to place this work in context from an industrial perspective.

Given the scope of the above, the terminology used to discuss autonomous networking is inconsistent. Specifically, automation and autonomy are often used interchangeably and the term AI is used to refer for everything from cognition to machine learning techniques. Accordingly, in this section we shall use the definition in section 2 for automation and autonomy and appropriately define the use of "AI" as required.

3.1 Operators: Transformation Before Autonomy

As explained in section 4.2.4, autonomy necessitates the ability to effect change, *i.e.* not only to decide that a change should be made, but to actually implement it. One such "lever" is abstraction of the underlying hardware, thus enabling the free scheduling of software on arbitrary machines. This

frees us from specific, proprietary vendor hardware that requires hand-configuration. This concept manifests itself within the data centre environment through compute[37] and network[11] virtualization. Similarly, in telecommunication networks Virtual Network Functions (VNF)[41] provide the first step on the road to full Cloud Native Networks.

Currently, most network operators are engaged in the *transformation* of their networks towards fully virtualised network[5,12,17,88]. Full virtualisation is necessary to enable the flexibility and programmability required to achieve anything beyond basic levels of automation or autonomy. However, the current state of these networks is often a patchwork of proprietary technologies assembled over decades from different vendors, which are able to interoperate thanks to a meticulously woven set of standards. As a result, most industry discussion of autonomous networks is still focussed on future planning and evaluation of such autonomous ability[29,72,124], instead of implementation as their networks are not yet ready.

Luckily, we are in the exceptional situation of having access to a fully virtualised telecommunications network operated by a nationwide carrier. We will utilize this network as the testbed for our approach to autonomous networking, and gradually deploy our technology for the improvement of this network and the service to our customers.

3.2 Vendors: Stuck In The Middle

Technology vendors are actively discussing autonomous networking. As they focus on selling solutions, their descriptions of the utilized technology is overly generic and lacks detail.

For example, *Huawei* has based their approach for enabling an autonomous network on a collection of products collectively referred to as *iMaster*[61]. Within this product, virtualisation (see section 4.2.4) and autonomy technologies are combined. Extrapolating from their descriptions, their approach to autonomy is focused on "intelligent" operation and management, optimisation & resource scheduling, and evolving "AI". In each of these categories some potential use case areas are identified, but only at the conceptual level. Neither the definition of AI nor its scope is given, although they, like us, note that this AI must be able to respond to a network which is dynamically changing or *evolving*; As opposed to our approach introduced in section 4.5, their use of evolution does not refer to a concise, technical approach.

Another example is *Cisco*, whose approach to autonomy is called the digital network architecture (DNA)[24]. Like Huawei's approach, virtualisation and concepts of autonomy are mixed, but a larger focus is placed on virtu-

alisation. This is understandable as most operators are focused on transformation, as described above. Compared to Huawei, Cisco provides a more comprehensive discussion of their approach and not their products. With respect to autonomy, their system supports telemetry from the network, policy driven specification of intent, controllers with network wide visibility, and an orchestrator tasked with taking user intent and translating this to system manipulation. Primarily this document describes a conceptual landscape in which autonomy can be achieved but focuses on automation, not autonomy. There is no discussion of the "intelligence" which is capable of inferring the current state of the environment or making decisions.

One general, but important observation is that vendors are moving towards a more active role than in the past. The industry-wide move towards open source software, as well as wide adoption of machine learning frameworks and generic computing machinery, enables operators to build and deploy their own functionality to harness the data and knowledge in the network instead of having to rely on proprietary vendor solutions. Similarly for autonomous networks, operators are actively driving the change instead of relying fully on the vendors' vision.

3.3 Standards Bodies: Documenting The Visions

Before achieving autonomy, it is necessary to know what to automate and how. Effort have already been made to describe both the processes necessary to operate a network[126], as well as the tools and common terminologies to describe these processes[125]. By providing a better definition of the task of network operation, albeit in an abstract and generic manner, the identification of the use cases to be automated is simplified. The next steps are to document the approaches to autonomy. There are three main thrusts from standards bodies and industrial fora: *ETSI*, *ITU*, and the *TM Forum*, which are driven by operators and vendors.

3.3.1 ETSI: Experiential Networked Intelligence

The ETSI Experiential Networked Intelligence (ENI) System Architecture[40] describes a very conceptual approach to create an autonomous networking framework; implementation and concrete descriptions are deferred to a subsequent document. It is relevant to note that the primary author of this document has a long track record in the field[121] and has a similar mindset to ours. Like our work, the ENI approach is based on the concept of the cognitive loop (section 4.2), but describes a much finer grained decomposition of the analysis and decision phases. However, as our cognitive loops

are defined at runtime, comparison of the abilities is difficult. ENI provides an an extensive discussion on how the ENI framework interacts with the network itself, detailing multiple different scenarios and approaches, but no concrete approach or description. We address this issue through user-defined modules and their specifications (as described in section 5), and provide concrete examples in section 6. ENI adopts a similar stance on decision making technologies as us, in that it aims to be agnostic of a single approach.

A key limitation of ENI is that it does not explicitly consider self-evolution. There is a discussion on the use of reasoning to be able to adapt to new situations, however, given the lack of grounded discussion it is not clear to see how to achieve such adaption in practise. In contrast, our approach is clearly documented (section 4) and actionable (section 5).

One aspect covered by ENI, but which we do not address in this document, is the concept of inter-component on-demand negotiation to achieve some outcome, for example, resource schedulers from different domains negotiating to access a resource. While we do support the hierarchical organisation of cognitive loops which requires interaction, our approach composes cognitive loops at runtime. Therefore, the role that negotiation will play is use case dependent. As this is an interesting question to explore, we plan to do so once we start implementing our system for the actual network.

3.3.2 ITU

The International Telecommunication Union (ITU) has provided a specification for a framework to apply machine learning (ML) to future telecommunication networks[67]. Note that this is not the same as autonomous networking. Like ENI, it is also based around the concept of the cognitive loop, however, unlike ENI, the ITU framework is more concrete and grounded. The framework implicitly supports the cognitive loop, which is embodied in an ML pipeline, equivalent to our concept of a controller (section 5.2). Like our approach, there is reference to a flexible, abstract specification of both system and ML applications. Also, there is the concept of a data sharing store, however, there is no mention of the need for eventual consistency as required by ML models and network elements which operate at different time scales.

Within the ITU framework there is discussion of the need to split ML applications in response to the management and orchestration function modifying the network. This is only possible for pre-defined machine learning applications, thus limiting the scope of adaptation to previously encountered situations which models have been trained for. Our approach is capable of achieving this for ML, statistical, heuristic, or any other approach that can be expressed as one or more modules (section 5.1) and thus represents a

superset of the ITU approach. Furthermore, that approach only describes modification of the ML application and not modification of the orchestrator. As we represent the orchestrator/controller(s) as cognitive loops, our approach also provides this capability.

The ITU framework also discusses the use of security best practise in not allowing data access to unauthorised parties. While we do not explicitly discuss security here, we see it as another set of use cases to which our approach can be applied. As long as the appropriate modules are available and the domain defined, a controller can be created to address any number of security considerations applied both to the network or to itself.

A key differentiator between our approaches and that of the ITU is that the ITU framework requires human operators (via policies) to decide which ML approach to take. Conversely, our evolution-based approach (section 4.7) is in charge of such decision for ML or any other approach, thus reducing the role of human operators and providing a more flexible framework. Furthermore, the ITU approach targets one independ model per use case, whereas we believe that a holistic approach (see section 2) is needed.

Finally, the ITU framework is also a reference specification without an implementation (that we are aware of) to validate the approach, whereas our work is based on insights obtained through implementation[64,65].

3.3.3 TM Forum

The TM Forum has an active working-group investigating the topic of autonomous networking. The group is articulating their vision by means of white papers[17]. They again emphasize the importance of transformation to a fully virtualised environment (also known as the telco-cloud) as a precursor for automation. Currently, the group is working to define the concept or shape of an autonomous network. There is no concrete definition; instead they are, like others (fig. 1), defining levels of automation based on the concepts of self-driving cars or the IBM autonomic manifesto[27]. So far the TM Forum has not discussed *how* to address the problem.

Based on the their previous efforts to describe the operation process[126] and tools associated[125] with a network, they provide a break down of use cases, however, the use cases are (in the context of this document) high level and abstract. In general, they describe automation in the context of business, network, and architectural concepts, and it is difficult to understand the overall practical approach at this point. One point of note is that at the architectural layer the network is split into different autonomous domains and that these domains should coordinate. The scope of a domain is not defined, but this matches our concept of how to apply the cognitive loop.

3.4 Summary

Based on the above, it is clear that there is an active and engaged community seeking to address the topic of autonomous networking, thus showing that it is a priority topic for the global community. As autonomous networking is an unrealised dream, there is a wide range of conceptual, practical, and business approaches to the challenge, however, all seem (maybe necessarily) to be lacking concreteness, and thus finding the most promising contender is not yet possible.

4 Core Technologies for Autonomy

An autonomous network cannot just come into existence from nothing in a deus-ex-machina like approach, but instead will have to gradually *evolve* from existing technologies. After stating our motivation for *why* we need autonomous networks in section 2 in this section, we introduce *how* academic research in various fields of computer science *and beyond* will enable us to fully achieve this goal in the future, and partially today. What prevents us from achieving full autonomy today, what research is needed to fill these gaps, and why we assume that our approach will be future proof regardless, is also discussed in this section.

4.1 Autonomy in Networking

Autonomous networks are not a new idea of the telecommunications industry. The topic itself has been investigated for decades and led to several independent larger research initiatives[10,33,50,119]. One seminal work is Clark *et al.*'s knowledge plane for the Internet[25], which inherently depends on the following attributes:

Edge involvement Much of the information needed to efficiently operate a network is generated outside the network, in the devices and applications that use it, and shall therefore be made available to the knowledge plane.

Global perspective A holistic approach is needed to leverage all information that could potentially enhance its performance, including observations gathered in other parts of the network.

Compositional structure Separate entities, such as distinct sub-networks, shall be able to interact and merge their perspective and activities.

Unified approach A single, unified system, with common standards and a common framework for knowledge is required since real world knowledge is not strictly partitioned by task.

Cognitive framework Cognitive techniques serve as the foundation of the knowledge plane. Representation, learning, and reasoning allow it to be "aware" of the network and the effects of its possible actions.

Based on these and other arguments presented both in academic and industrial research, as well as our own experiences[64,65], we derived the following five principles we deem necessary for achieving true autonomy. The first three are variations of the attributes demanded for the knowledge plane.

Holistic approach In the spirit of the *edge involvement*, *global perspective* and *unified approach* required for the knowledge plane, we expect truly autonomous networks to have all-encompassing access to information about the network. We extend this requirement to encompass access to all network functionality that could benefit from autonomous control, *e.g.* network components, as well as all control, optimization and adaption functionality deployed therein. For details, please refer to section 4.3.

Abstraction and Genericity We apply the argument for a *compositional structure* not only to the interoperability of parts of the network, but extend it to encompass the interaction between distinct parts of "intelligence" in the network. For this purpose we assume that the network and all the control loops that co-ordinate its operation are composed out of generic components, which expose their functionality through abstract communication interfaces. In section 4.2 we introduce the core abstraction of an autonomous control loop, arbitrary numbers of which can operate independently or interact with each other in the network. These control loops themselves are also similarly composed out of generic components, as detailed in section 4.4.

Hybrid Intelligent System Autonomy naturally depends on the ability to make informed decisions. The more complicated these decisions become – especially if the task is perform (most of) the work currently performed by highly skilled engineers – cognition becomes a pre-requisite, just as for the knowledge plane. In addition, we expect the network to leverage the right tool for the job, which likely means that both symbolic and connectivist approaches will be utilized concurrently, as detailed in section 4.5.

Functional composition We consider a self-reflective design – especially the ability of the network to adapt and improve itself based on what it

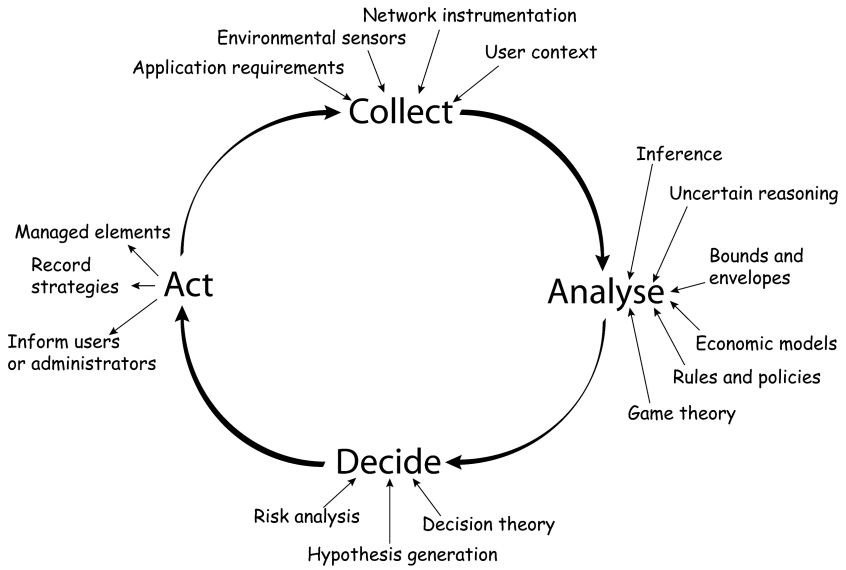


Figure 2: Autonomic Control Loop[35]

learned during its operation – vital for its success. Functional composition, the online construction of network components based on available building blocks, provides us with the means to achieve this goal: it enables the system adapt its own structure to the needs of the situation and to integrate new technology as it becomes available. This makes it future-proof and able to not only improve the underlying controlled systems' operation, but its own functionality as well.

Experimental Evaluation Since modern communication networks are too complex to appropriately reason about theoretically or by means of simulation[92,117], we state the necessity of online experimentation to evaluate the potential of new solutions or configurations in the actual environment, especially when the goal is performance optimization. In section 4.7 we make our case for application of the biological concept of evolution and other randomized search strategies to the network to tackle the complexity of the optimization case. The tradeoff between the potential gains and the costs of deviation is also discussed.

In the remainder of this section we introduce several technologies that will enable us to realize these principles, along with a brief overview of how we intend to utilize them in our framework design detailed in section 5.

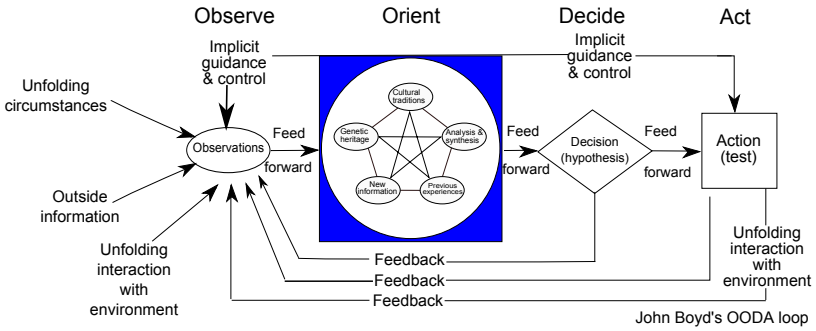


Figure 3: Col. Boyd's OODA loop, as commonly used for decision making in warfare, cyberdefence, as well as business, applications.

4.2 The Cognitive Control Loop

Perhaps unsurprisingly, the core concepts which guide autonomous behavior, cognition, reasoning, life and so on are based, are basically identical, even though the nomenclature or field of study differs. All manifestations and realization of artificial autonomy within artificial intelligence, cybernetics, biology, medicine, and so on, very closely resemble the *feedback loop* from Norbert Wiener's seminal work from 1949, *Cybernetics*[135]. The feedback loop is also known as the *autonomic control loop*[34], the *cognitive cycle*[85] (fig. 2), and by several other names[20,32], but the concept itself does not change.

The architecture of the *controller* (section 5.2) is based on the feedback loop from fig. 2. The *collection* or *sensing* stage deals with the gathering and aggregation of sensor data or other information from which, during the *analysis* stage, an understanding of the current state of the system and its environment is derived. The decision of how to act is taken based on the knowledge of the world state combined with an estimate of the likely effect of potential actions, during the synonymous *decision* stage. The corresponding actions are then applied during the *action* stage. Note that all these stages do not necessarily have to be performed strictly sequentially. Similarly to the OODA loop[19], which is depicted in fig. 3, we assume the ability for local interrupt and restart of the individual phases based on updated information. Furthermore, one control loop on its own will likely be insufficient in practice for the realization of an autonomous network. We will therefore employ a hierarchy of control loops, as explained in section 4.3.

4.2.1 Sensing: The Need for Automated Data Abstraction and Aggregation

Traditionally, any system able to adapt to its environment would be provided not only with the sensory input, but also with the means to derive state information from this data, through inherent contextual understanding that was designed into it. Just as a thermometer on its own is useless without knowing what can be considered hot or cold, a temperature sensor in a computer requires knowledge about the acceptable operating conditions of the measured component. The actions needed to *derive meaning* from raw sensor data provided from arbitrarily many distinct sensors can be very complicated, as demonstrated by the complexity of the time series aggregation capabilities of common network monitoring and analysis packages such as Google's Borgmon[129] or the open-source Prometheus[96].

In traditional systems, letting engineers supply a fixed logic to perform these steps is obviously a very sensible approach: these engineers can be expected to understand the purpose of deploying these sensors and thus also how to retrieve valuable information from them. And just as an engineer might need training to understand how to best utilize novel technologies, the same holds true for a computer system, which traditionally gets taught by means of programming, through code, scripts, or configuration data. In the case of constantly (self-)evolving autonomous systems, however, the same notion might no longer hold true. If the intention is to provide the system with the means to go beyond its programming and thus reduce the workload of the engineers, then we need to provide it with a way to derive meaning and understanding from semi-raw data on its own.

In the same spirit as W3C's use of OWL[132] to describe the data on the World Wide Web[131,133] to enable machine-based semantic understanding, we will utilize an ontology description language to give *meaning* to sensor data. Because we do not intend for the autonomous network to add its own sensors, giving the relatively small task of providing attributes for each sensor to the engineer seems to be an acceptable trade-off, considering the potential for automatic re-use of technology that has been built to utilize a sufficiently *similar* sensor[45]: For example, for any sensor classified as a packet counter and providing count c_i at time t_i in seconds, the average count per second could thus be automatically aggregated as $\frac{c_1-c_0}{t_1-t_0}$. We detail our description language and its application in section 5.4.

4.2.2 Analysis: From Metrics to Awareness

Vital information can be extracted from raw sensor data by means of both top-down (*i.e.* symbolic reasoning) and bottom-up (connectionist) approaches.

As sensors are expected to be limited in number and variation, and oftentimes already provide *ontology* information [3], we utilize semantic understanding to *reason* about their meaning. Several frameworks are already available for this purpose[54,98,140]. Such techniques enable us to, for example, automatically provide new aggregated metrics whenever sensors are added to the system, by means of *e.g.* calculating the average for temperature and summation for throughput.

Conversely, advances in machine learning have enabled the automatic *classification* of massive amounts of raw sensor data and thus enabled automatic diagnostic and mitigation of common networking issues without having to address the challenge of comprehension. For example, connectionist[128], and in particular deep learning[71,130], approaches can *perceive* issues in the network and thus led to significantly improved capabilities of network intrusion detection systems. And advantages in the field of artificial intelligence are likely to improve the ability of such systems further. We therefore also use such techniques as appropriate.

However, thorough comprehension of the state of the network environment and the controlled systems themselves enables human engineers to solve problems that are still out of reach for automated systems. As we discuss in detail in section 4.5, this ability to extract *meaning* from seemingly chaotic data is still a very active research field, and similarly general and capable solutions have yet to be found.

For the time we therefore focus on providing an approach that is powerful enough to tackle many of the current issues in networking and at the same time flexible enough to easily integrate more advanced technologies as they become available. The techniques that provide us with the flexibility to easily integrate arbitrary future technologies are discussed in section 4.4.

We utilize a hybrid approach based on both symbolic reasoning and connectionist technologies to derive meaning from sensor data. We also apply heuristics when appropriate. For example, for optimization problems, we will utilize an abstract measure of utility, also known as optimality or fitness, to direct the search for the best solution (see section 4.6). Due to high importance of this metric and the difficulty of the learning task, we expect the utility functions to be operator-defined. We provide more details about the

[3] Even venerable standards like SNMP[95] provide a hierarchy or description which explicitely or implicitly encodes such information.

techniques we intend to leverage insection 4.5.

4.2.3 Decision: The Difficult Path To True Autonomy

The decision of which actions to apply to the network, be it to self-configure, to improve the operational characteristics of the controlled system, to counter a detected intrusion, to fix a problem in a sub-component and so forth, are fully use case dependent and can become arbitrarily simple or extremely complex.

Self-Organising Networks (SONs)[2] are an example of a straight-forward decision process, as virtually all decisions a SON will take are based on simple heuristics, deterministic reaction, and defensive (resilient) design. For example, if the master instance of a component fails, the pre-defined rules will instruct the system to fall over to the backup, or if a host is unreachable to restart it. This approach requires that one (or many) human engineers designed, programmed, and tested these rules in the network before deployment, which limits the dynamicity of the resulting system.

Unfortunately, highly complex problems in the network are however very common, and the human-centric approach that worked for SON does not scale sufficiently well. Considering optimization problems on their own, numerous strategies have been exploited, such as to decide where in the network to provide a service[6,80], how many resources to allocate for it[22,77], or how to configure antenna parameters to optimize throughput and coverage[30,38,139]. Still, a practical yet optimal technique has in many cases not yet been found.

As this paper's aim is to present a generic framework that can address all potential use cases for autonomy in the network, we do not favour a particular solution for any of these problems. Instead, we show in section 4.4 how any potential approach can be integrated into our system, and in section 4.3 how these techniques enable autonomous improvement of the system itself. Furthermore, as discussed in section 4.2.2, our system will be able to address these issues using the best currently available techniques thanks to its extensibility and easy integration of arbitrary future decision and optimization techniques once they become available.

4.2.4 Action: A Lever for the Network

An autonomous network would be absolutely useless without the ability to change the systems it controls *on its own*. Consequentially, it requires the ability to affect the operation or organization of the network without the need for manual intervention. Luckily, such capabilities are gradually becoming more commonplace in telecommunication networks and beyond.

For example, the move towards end-to-end network virtualization offers huge cost saving potential for the operator. By replacing proprietary hardware with abstract software Virtual Network Functions (VNF)[41], it enables us to freely choose where to deploy such VNFs at runtime. Thus rbitrary services such as DNS, billing, or even customer edge application can be hosted in generic containers independent of the underlying hardware. VNFs together with Software-Defined Networking (SDN)[44], which provides the necessary levers to control the operation of network switches and routers online, has resulted in renewed (business) interest in network overlays in the form of network slicing[102]. Even mechanical features have become virtualized in the last decades: Thanks to phased arrays[82], modern 4G/5G base stations can electronically control the direction the antenna is facing.

The benefits of automated configuration are not limited to the world of telecommunication providers. The open-source Apache Mesos[58] and Google's Kubernetes[57] bring infrastructure virtualization to the data center, which enables software-based job scheduling and placement. All of these technologies enable us to not only automatically change the configuration and operation of the network, but also provide tools we will leverage for the purpose of autonomic operation.

4.3 The Control Hierarchy

Biological systems hierarchically control the actions of their subordinate entities. The higher-level brain functions of the frontal cortex make higher level decisions, for example, to run away from a threat, but cannot implement them without the help of the cerebellum, which translates "run towards the door" into actions to be performed by the legs. The cerebellum in turn does not control the actions performed to counteract sudden events which need immediate actions. The latter is the domain of reflexes, which are hardcoded and able to respond far quicker than the cerebellum or the even-slower frontal cortex ever could.

The rationale for this split is that higher-level functionality is more costly, as it involves more capable neurons, and takes longer. Offloading these functions to older and less complex parts of the brain not only increases reaction speed, such as when the cerebellum applies learnt operations without cognitive involvement, but frees the more advanced parts to handle more challenging tasks.

The same challenges described above had to be overcome by the first mobile robots, whether they are bipedal or wheel-based, and have lead to the application of hierarchical, layered architectures[51]. Equally, human procedure also follows the same pattern, as exhibited by military and (some) corporate

organisational structures, where long-term strategical decisions are taken at a higher rank than tactical or short-tem battle decisions. Likewise, the aforementioned OODA loop shown in fig. 3 inherently embeds this hierarchical interaction between quicker reaction and longer deliberation in the form of a feedback channel.

Our vision for an autonomous network reflects this concept of providing multiple control loops in a hierarchical structure. We intend to model the process how the cerebellum directs the actions of a leg, which can be overriden when needed by a reflex, within our network: for example, a higher-level controller assigns weights per data center, which in turn are used to distribute application instances among machines in that DC.

In our design, however, the hierarchy is not limited to optimizing operation. The structure and composition of the underlying control loops themselves can be modified and improved by higher-ranked control loops. Furthermore, we do not employ a single, linear order of layers, but instead use a directed graph. In our approach, several subordinate loops can be controlled by one or multiple higher-ranking loops, responsible for different optimization aspects (operation *vs.* evolution). We provide a more detailled description in section 5.3.

4.4 Functional Composition: Lego Pieces for Network Castles

Modular designs that allow run-time construction of functionality from small, generic building blocks have a long history in a large variety of computing applications. Dennis Ritchie[100] introduced the flexible, coroutine-based stream io subsystem for character devices into Unix back in 1984, which allows the output of one device to be connected to the input of another via pipes. The Ficus[53] file system uses *stackable layers* (in other wordsmodules) with symmetric interfaces to access services provided by other modules in the stack.

Applications to the networking field also have a long track record. The *x*-kernel[62] enables runtime construction and composition of networking protocols, as well as abstractions for common protocol functionality since 1991. Automatic reciprocal reconfiguration was explored by the Pandora[91] / C/SPAN[89] projects in 2000, which stack software components that communicate through message exchanges and utilize reflexive interfaces for reconfiguration. This, for example, enables Pandora and a collaborating web cache to reactively tune each other according to the measured disk space and traffic characteristics. Larger scale deployments of functional composition frameworks have been explored in both the ANA[7] and 4WARD[4] projects. Their

composition framework allows for transparent runtime rebinding of the communication channels between functional blocks at the sole discretion of the framework[18].

Generic building blocks are functionality- and implementation-agnostic and enable to build a future-proof and easily extensible system. Our own design will utilize them for flexible runtime (re-)composition of functionality modules. These modules communicate over strongly-typed transparent interfaces, the format of which is at the discretion of the module designers, as detailed in section 5.1.

4.5 Artificial Cognition

Research into artificial thinking and reasoning is spread across many different areas of science, from biology and medicine to mathematics and philosophy. In particular, it can be traced back two fundamentally distinct approaches, trying to develop systems that either think *rationally* or *like a human*[104].

The concept of rational thinking can be traced back to Aristotle's *syllogisms* and has been gradually mechanized over time. This process started from research into formal reasoning, such as Frege's *Begriffsschrift*[48] in 1879 and Whitehead and Russel's *Principia Mathematica*[134] in 1910, continuing with the General Problem Solver[86] in 1959, and leading to cognitive architectures such as SOAR[74] and ACT-R[101].

Artificial Neural Networks (ANN)[105], the main connectionist[83] approach to explain human-like perception using artificial neural networks, have their origins in Hebbian learning[56]. Their first practical realization was the Stochastic Neural Analog Reinforcement Calculator (SNARC) in 1951, a randomly connected network of 40 neurons, designed by Marvin Minsky[106]. ANNs had mixed fortunes over the last 70 years, with the first hiatus maybe caused by Minsky himself in 1969, when he discussed their limited pattern classification and function approximation capabilities[84]. Fukushima's Cognitron[49] inspired convolutional neural networks and consequently the advent of Deep Learning[112].

Especially thanks to the vast amounts of computational power and data that had been missing before, ANNs have enabled huge performance increases in several application areas, which lead to an enormous boost in their popularity in the last decades. Deep Reinforcement Learning has, for example, enabled machines to excel at board (go, chess) and video games and consistently best top human oppenents[116]. While the aforementioned approaches showed impressive results in certain areas, no generic approach for learning has been found so far. Lately, many researchers acknowledge the limitations of Deep Learning approaches and explore potential solutions to advance the

state of the art, for example by utilizing meta-learning of causal structures to improve the adaption speed[14].

Meta-learning[76] explores how to overcome the inductive bias inherent in learning algorithms through automatic optimization of machine learning algorithms, for example by optimizing the parameters that guides these algorithms. Of particular interest to us is the idea to employ a hierarchy of learning systems, in which each layer learns how to improve the performance of the underlying layer of the hierarchy, for example through Genetic Programming[13,113,114].

Even so, the fact remains that artificial intelligence is still extremely far from matching the cognitive abilities of a human being. Building a system which is able to reason on its own and, for example, solve a problem by applying the scientific method, can be considered the holy grail of AI research. Especially since achieving this goal seems at least as hard as acquiring the grail.

Consequently, we will not be able to fully achieve our dream of replacing all networking engineers with an autonomous system in the near future, as this would amount to the creation of an entity capable as a real engineer. This is still far beyond our abilities. The easiest to articulate of several issues that hinder general artificial intelligence's coming into existance is the lack of providing it with sufficient context knowledge to solve all problems a human engineer might be posed with: Human engineers possess context knowledge that exceeds the scope of networking and can apply it to solve seemingly unrelated or infrequent issues. Thus they can, for example, preemptively provide more resources for a service which will become more popular due to a popular sporting event or news item. However, as stated before, we designed the framework to be flexible enough to integrate any future AI technology as it becomes available, and thus utilize a manifestation of this artificial engineer, whatever it may look like.

Even partial autonomy will allow us to reduce costs as well as increase the capabilities of the network, and is therefore worth pursuing. We believe that the current state of AI research enables us to build a *semi-autonomous* network, and that it can be realized by means of a combination of both semantic / symbolic / computationalist and connectionist / emergent approaches, as appropriate for the application context. If context and logical relationships can reasonably be provided by the engineer, such as in the case of sensors, we intend to leverage it by means of symbolic reasoning. Whenever this is infeasible, due to the amount of data, variation or lack of knowledge about relationships, for example, how exactly an intrusion attempt would manifest itself in the network, then we will focus on connectionist approaches.

4.6 Metaheuristic Optimization

Many, if not most, optimization problems encountered in computer networks can be expressed by means of a utility function[107], which provides a measure of optimality for a particular potential solution, state, or configuration setting. In case of difficult (high-dimensional, non-convex) or unknown search spaces, which can not exhaustively be explored, common heuristics, such as hill-climbing[108] or simulated annealing[109], are not applicable. Here, the introduction of randomicity not only helps to traverse the potentially vast search space of the network, but also deals with uncertainty in cases were the underlying relationships between actions and outcomes are either unknown or difficult to reason about. Stochastic meta-heuristic approaches offer a flexible way of finding a sufficiently good, but not necessarily optimal, solution. Many examples of such approaches are based on or inspired by biological processes such as genetics (Genetic Algorithms[110]), evolution (Evolutionary Algorithms[21]), the behavior of ant colonies (Ant Colony Optimization[36]), or biased random exploration (Rapidly-Exploring Random Trees[75]).

Since all these approaches share common inputs (current state, (partial) history, information (utility, *etc.*) about previously visited points) and output (new state to explore or set of such states), modularization and use as building blocks for functional composition (see section 4.4) is straight-forward. By layering these approaches in inter-dependent control loops, meta-meta-heuristic optimization thus becomes possible and is applied in our framework.

4.7 The Case for Experimental Online Evolution

In 1997 the Internet was already too complicated to appropriately simulate[92] and the same claim can easily be extended to today's massive telecommunication networks. Even if the system itself was not complex, the interaction with the environment can easily make it so[117], especially considering that the traffic carried on these networks is predominantly human-initiated and -controlled.

Sentient beings inherently overcome similar problems through online experimentation and continuous learning[137]. Similar approaches have been employed for finding the best networking protocol to deploy in the network[97], to learn the best radio parameters for battle field communications in a limited number of trials[127], or to find the best bit-rate for wireless communication[16].

The trade-off between exploration and exploitation, is a major issue for online search heuristics[111,122]: more experiments lead to better results in the long run by exploring more potential improvements, but each test naturally can result in performance degradation at least as easily as it can lead

to improvements. The multi-armed bandit problem has been a particular focus of study, which led to strategies such as ϵ-greedy, Bayesian approaches, or approximate methods. In general, however, the best strategy to choose depends on the problem set and a general solution for the full reinforcement learning case has yet to be found[123].

One additional caveat is the possibly exponential growth of the search space that needs to be explored experimentally, if optimization is to be performed on multiple interdependent layers. The optimal strategy to this problem again depends on the peculiarities of the problem set.

In our framework, we will employ online experimentation of potentially better solutions for optimization problems, with the deployment and testing strategy being provided as a functional building block (FBB) and therefore subject of online optimization itself. In addition, we will perform sanity checking, simulations, and gradual rollouts to reduce the number of experiments to perform and the impact on the network operations, as detailed in section 5.8.

5 A Framework for Autonomous Network Evolution

Guided by the core concepts and related technologies we identified and motivated in section 4, we designed the following architecture for the telecommunication network of the future. Instead of presenting a complete technical design, we explain how the key elements of our approach are embodied in our framework. We will seperately provide a thorough design document to detail APIs, robustness, ontology formats, various code and metric repositories, failure and recovery handling and other engineering-focused topics.

5.1 Module: Building Block for Functional Composition

The functional building block (FBB) is a key element of our approach to autonomy (section 4.4) and realized within our framework as a *module*, (fig. 4). We define a module as consisting of both a *software* component and the corresponding *composition information*. This enables us to compose controllers out of compatible modules using the process described in section 5.5.

The software section represents the operational logic of a module and consists of three parts: *code*, *parameters* and *API*. Code represents the logical operation of the module and is expressed as software. The scope of this

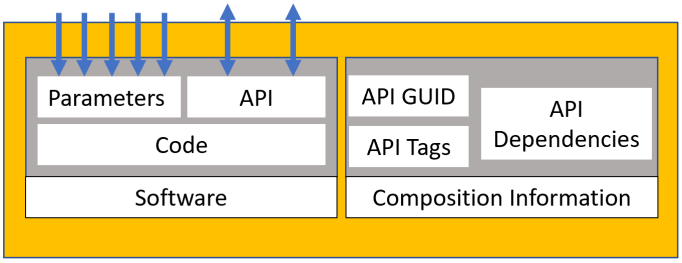


Figure 4: A Module

software is defined by the designer and may be of arbitrary size; such considerations are independent of our design. Parameters are used to initialise and configure the operation of the code, for example, how long to wait before a timeout. Finally, there is a well defined API which enables this code module to be interacted with.

This API is identified by a globally unique ID together with an arbitrary number of *optional* tags that specify the provided functionality more closely. For example, an audio codec might provide an *Encode* API, which can have the optional tags *lossless* or *lossy*, to indicate whether decoding the encoded output would return the bit-wise identical input data or not.

Importantly, in our framework the globally-unique API identifier combined with the optional tags are expected to constitute a "contract" for composability and interoperability. This contract guarantees that a module which requires a certain API can utilize any API with the same given ID. The actual API used or its calling conventions are of no importance to the composition system, and are therefore not represented within the module definition. Incompatibilities w.r.t. functionality or code-level iteraction consequently necessitate the use of a different API identifier. The inverse case, *i.e.* the use of a different name or API for compatible functionality, should also be avoided, as it reduces the freedom of choice of the composition system. This approach could be implemented similarly to the assignment process of Internet protocol numbers by IANA[63], which does not register any specifics about the protocols themselves, only assigns unique numbers to be used by them. We do however defer the decision about how to handle this process to future deliberation.

Modules can depend on the presence of APIs provided by other modules. The same API can be provided by multiple different modules, for example, sound encoding functionality could be implemented by a lossless FLAC or a

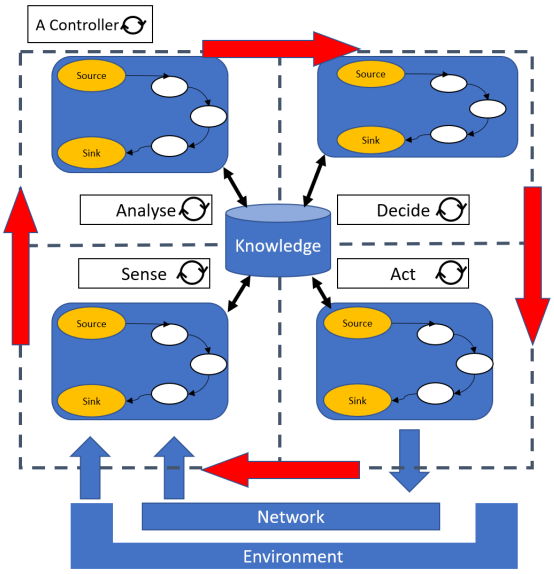


Figure 5: A Generic Controller Architecture With Cognitive Loop Flow. The theoretical "flow" of information around the cognitive loop is represented by red arrows.

lossy MP3 module. Likewise, differently parameterized instances of the same module could be utilized concurrently. Dependencies are therefore specified by means of the API UID and an arbitrary number of tags that need to be provided by the target module instance, to which this module instance will be connected. In programming terms, such a connection could, for example, be represented by a pointer to an object which provides the necessary API, or by a RPC function on a remote host.

We emphasize that we do not assume the *design* of a module requires any particular technology choice in either the software used, the overall operational purpose, or the scope of the module. This is equivalent to the concept of deciding the size and scope of a software module in an application. Also note that our framework assumes that modules are user provided instead of autogenerated. While software generated modules can be accepted by our system, we consider them out of scope for the current document.

Given these specifications, modules can now be programmatically (or automatically) composed together to create *controllers* (section 5.2).

5.2 Controller: An instantiation of the Cognitive Loop

As introduced in section 4.2, the cognitive loop is one of our fundamental concepts of autonomy. In our framework, the cognitive loop is manifested

as a *controller*, fig. 5. A controller is responsible for the control, operation, or optimisation of some task or domain within our network. Just as with modules, the size and scope of this task or domain is defined by the user.

The four phases (*sense*, *analyse*, *decide*, and *act*) of the cognitive loopa are all present within a controller. We assume that each controller element operates on an independent time scale from the others. For example, sensing can be a continuous process, where as analyse is likely to operate only from time to time to interpret collected data. Decisions are either periodic or triggered by changes in the environment, and actions are in response to decisions. Each controller phase is a composition of modules.The composition process is described in section 5.5.

Each phase of a controller can be seen as a directed graph of nodes. The nodes in this graph each reprent one module instance. The root of this graph is the *sink*, an abstract module which depends on the required inputs of the next phase in the cognitive loop and whose sole purpose is to ensure that each phase delivers everything that is needed for the next phase of the controller. The next phase contains a corresponding *source* module, which provides access to the output of the previous phase. A detailed description of what exactly is required for each phase is provided by the controller specification (see section 5.4.3), in conjunction with additional requirements derived from the modules present within this next phase. Modules can possess an arbitrary number of dependencies (see section 5.1). These identify the required APIs used by the module as defined in the module description (see section 5.4.1), and the vertices in the graph represent these dependencies. The structure of this graph, however, is not fixed. Since the dependencies of each module instance (node) guide the structure of subgraph starting in that node, arbitrarily complex graphs are possible.

It is the manipulation by means of creation from scratch, re-arrangement, replacement, and configuration changes of module compositions that enables our framework to adapt to both new and evolving situations (see section 5.5). While a controller could be expressed through one single composition, we decided to split the composition of the controller into four interconnected parts which embody the sense, analyse, decide and act phases, respectively. This separation both simplifies (human and machine) comprehension and reduces the state space of potential module configurations. It is important to note at this point that our framework currently requires the user to provide a metric for how to measure the fitness or utility of a controller by means of a utility function. We do not prohibit auto-generation of utility functions either, but due to the complexities and pitfalls associated with such approaches, we plan to thoroughly research and discuss this matter (also through practical application) separately in the future.

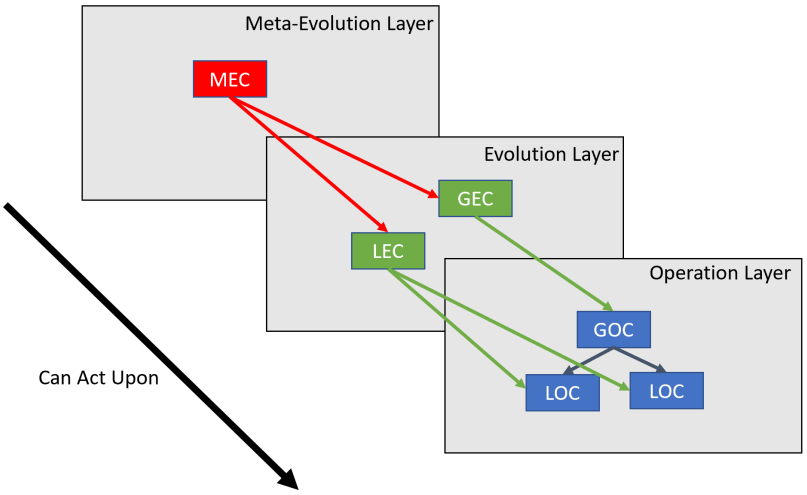


Figure 6: Conceptual controller hierarchy as separate layers.

Additionally, all controller phases share access to persistent knowledge through a knowledge base. This is necessary to understand previous choices and their consequences, changing system state, and to ease synchronisation across distinct update periods. The exact knowledge to be kept is dependent on the specific controller, but can be utilized by separate controllers if applicable. The knowledge store is an eventually consistent distributed data store, which we will also discuss separately.

5.3 Controller Hierarchy: A Layered Approach to Control

As stated in section 4.3, our approach uses flexible runtime-defined hierarchies of controllers, which consist of *operation controllers* (OC) and *evolution controllers* (EC).

Operation controllers either directly control network elements or supervise other operation controllers. They provide, for example, heuristics for network load balancing, resource distribution and job scheduling logic or anomaly detection and mitigation technologies.

Evolution controllers optimize and adapt the composition and configuration of other controllers for the purpose of achieving the best possible utility under the current operation conditions. This is an open-ended, exploratory task and refered to as evolution (see section 5.5). Evolution controllers can be under the supervision of other (meta) evolution controllers and thus evolve themselves, as we explain below.

For illustration purposes, fig. 6 separates controllers into different layers according to the roles they perform. In practice, however, all controllers are arranged in a single hierarchy graph as depicted in fig. 7. The leaves of the hierarchy graph are all operation controllers.

Operation controllers can be controlled by other higher-level operation controllers, if needed. In this case they supervise and direct the operation of their subordinate operation controllers, instead of controlling underlying network components directly. For example, the leaf operation controllers might be responsible for the optimization of a process pertaining to individual data centers. The boundary conditions within which operational adjustments are possible could then be decided by a regional operation controller. Above the regional operation controllers, a global operation controller might again be deployed. Operation controllers cannot influence the controller graph or the evolution process. The leaves in this hierarchy graph are all operation controllers.

Evolution controllers on the other hand decide when and how to evolve the controllers they are responsible for. The major distinguishing factor between the controller graphs (which define the composition of a controller phase out of module instances) and the hierarchy graph (the nodes of which are controllers) is that each evolution controller in the hierarchy graph has the liberty to define in software its own dependencies. This means that each evolution controller defines the specification of the controllers (and how many of them) it needs in the layer directly below itself, either in software or directly in its corresponding controller description (see section 5.4.3 for details). Thus evolution controllers are, for example, able to apply one or multiple independently evolved operation controller per data center, per region, or globally at their discretion, compare the results, and decide which approach is most efficient, as measured by their utility function.

The creation of a hierarchy of operation controllers enables us to separate local decisions, which might require fast reactions, from more deliberate global decisions which can be performed more slowly. For example, a single base station controller can quickly decide to adjust its tilt based on the number and conditions of the connected devices, however, the global controller can get feedback from many local controllers and provide more general policy decisions at a larger temporal granularity. Thus our framework represent and encompass the same underlying concepts centralized and distributed SON[1] provide, such as the deployment of agents in each cell to co-ordinate and minimize interference.

Whether such a separation into multiple operation controllers is sensible or not strongly depends on the use case and application environment. This is one of the reasons why we allow evolution controllers to decide the hierarchy

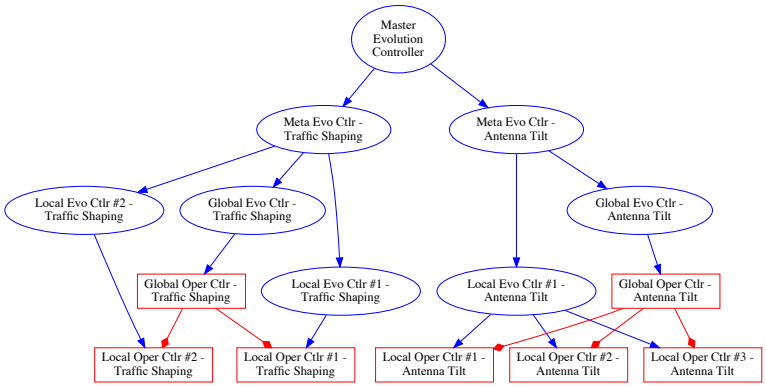


Figure 7: An example controller hierarchy. Operation control is depicted by red diamond-tipped arrows, evolution control by blue triangle-tipped arrows. Correspondingly, red boxes indicate operation controllers, blue ellipses evolution controllers.

below themselves, as it enables them to try out different potential operational separations of control and choose the right one for the current situation without having to depend on input from an actual engineer.

Evolution controllers can not only control operation controllers, but also evolution controllers, in a role we call meta evolution controller. Having a hierarchical ordering of evolution controllers enables us to apply different evolution approaches depending on the task at hand and the operational environment, as often the optimal optimization or adaptation strategy depends on these: For example, the optimization strategy for in-data center resource allocation might differ from the regional strategy (different time scale, explicit allocation to machines *vs.* weights per application group, *etc.*). The mapping of meta evolution controller to evolution controller follows a similar logic, as the ideal amount and responsibilities of evolution controllers might depend on the use case and application environment, it is the prerogative of the meta evolution controller to decide (*i.e.* evolve) the composition of the hierarchy below itself.

Figure 7 shows a somewhat complex controller hierarchy to illustrate the potential of a hierarchical separation of control. In this example, two use cases are represented, traffic shaping and antenna tilt optimization. For the traffic shaping case, a global operation controller decides the high-level weight allocations per location, and two local operation controllers shape the traffic while obeying the global weights. These three operation controllers are being independently evolved by their corresponding local or global evolution

controller. For the tilt case, all three local operation controllers are evolved by one evolution controller. All evolution controllers of the traffic shaping and tilt optimization use cases, respectively, are evolved themselves through a corresponding meta evolution controller, which in turn are evolved by the master evolution controller. Please note that we do not intend to indicate that for the given use cases the depicted separation was ideal. Our intention is solely to point out the flexibility of our approach to encompass either design.

In the first version of our framework, it is the engineers' decision to decide whether evolution controllers should be given the opportunity to try out different hierarchies of operation and/or evolution controllers, or whether it the hierarchy should remain static and evolution should be limited to the composition of the given controllers. For example, higher and lower level operation controllers can either be designed to work together to solve some use case, or evolved to do so. Whether to use a dedicated controller to supervise underlying controllers (or network entities) or to utilize a shared one depends strongly on the use case, controller design, or evolution outcome. And whether to give more freedom to experiment to the (meta) evolution controllers by *e.g.* letting them freely decide the hierarchy below them, and thus sadly also to increase the time it takes for them to come up with a close to optimal solation will be investigated by us in more detail in the future.

5.4 Description Language: Meta-Data for Symbolic Reasoning, Controller Composition and Use Case Specification

The description language steers the functional composition (section 4.4) and derivation of meaning (symbolic reasoning) from sensor data (section 4.2.1). It provides a normalisation layer that enables our system to programmatically understand and reason about the modules and sensors provided. Additionally, it allows us to specify constraints for controllers, controller hierarchy branches, as well as the corresponding utility functions, and thus to add new use case-specific controller (hierarchies), as detailled in sections 5.2, 5.3 and 5.4.3

5.4.1 Module Description

As illustrated in fig. 4, we enable our framework to sensibly compose controllers out of modules by specifying the capabilities, configurable parameters, and interface of a module using a description language. There are

```
Module LowPassFilter:
  Provides: Codec, Filter;
  Requires: Codec;
  Parameters: 0..9, 1..100, {5, 7, 9};

Module HighPassFilter:
  Provides: Codec, Filter;
  Requires: Codec;
  Parameters: 1..100, 1..10;

Module FLAC:
  Provides: Codec(lossless), FLACCodec(lossless);
  Requires: FLACFile;
```

Figure 8: Example Module Specifications

many different approaches specify such properties[26,43,94]. To maintain the genericity of this document, we defer description of our specific approach to the subsequent design document: Experimentation will help us identify the most suitable approach for the large number of domains that we seek to tackle in our telecommunications network. However, as a guiding example, we present a simple description in fig. 8 which shows the symbolic representation of three modules. Each module is identified by a unique name. The remainder of the description defines the capabilities that this module provides, its requirements, and the acceptable ranges of its configuration parameters respectively. For the latter, each module specification describes either the range of values that each (optional) configuration parameter can take, or the set of possible parameter values. These are discussed further in section 5.5.

As described in section 4.4, having each module provide a standard description enables equivalent but different modules to be interchanged programmatically. For example, a compression module designed for web server can be reused in a logging system so long as the module descriptions are compatible. This concept of module reuse is a key requirement of our system, see section 5.5.

5.4.2 Sensor Description

There are two types of description for sensors. The first is similar to the modules above and concerns the symbolic description of the sensors, such as thermistor, packet probe, energy meter, as well as the data types that they produce, *e.g.* degrees centigrade, packet loss, joules. Also like in the

case of modules, we require the developer to provide this information via a specification. To assist, our framework will support a taxonomy of sensor types and data. Existing work in sensor networks[73] and more recently IoT[103] have made efforts to classify both sensor types and data by problem domain to better exploit the right tool for the right job. Our framework will do the same; just as modules are symbolically described, so too must sensors within the context of our taxonomy. By doing so:

- sensors from one domain can be reused in another
- equivalent but different sensors can be interchanged
- classification can guide the process of "good" module compositions
- classification can help automate the process of sensor data aggregation and later reuse of this aggregation between similar sensors classes

The second description type concerns the inference of meaning from the raw sensor data (section 4.2.1). In this case, the use of taxonomies combined with *ontologies* will enable these relationships to be inferred[39].

5.4.3 Controller Description

The constraints that guide which controllers are required to be present in the framework for a particular use case are also specified by means of a description language. As explained in section 5.2, the conceptual structure of the controller is constant and always consists of the *sense*, *analyse*, *decide*, and *act* phases. The connections between the phases is provided through the corresponding *source* and *sink* modules of each stage. Figure 9 shows an example of such constraints as applied to a load balancing use case, in which one evolution controller is responsible for the evolution of two distinct load balancing operation controllers. Note that only the required outputs are specified explicitly. The required inputs are derived directly from the requirements of a composition that provides the needed outputs. These input requirements can also lead to additional output requirements for preceding phases.

The utility metric used to evaluate all controllers under an evolution controller can also be defined either in software (via a module provided for this purpose) or directly within the controller specification by means of a simple mathematical syntax, as shown in the example above. In accordance to the holistic approach (see section 4.1) we take, any available infornmation can be used for the utility estimation purpose, be it sensor data, module output provided by other controllers, or statistics gathered directly from the network.

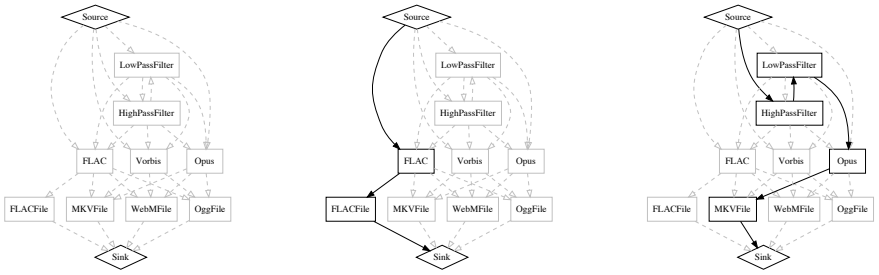
```
Controller LoadBalancer:
 ReqOutputs:
   Sense:   LinkStats,
            MachineResourceStats,
            QueryStats,
            LBPerfStats;
   Analyse: NetLoadPerSecond,
            MachineLoadPerSecond,
            QPS,
            QuerySuccessRatio,
            QueryLatencyScorePerSecond;
   Decide:  LinkWeights,
            MachWeights;
   Act:     DNSWeightAssignments,
            MachJobAssignments;
 Utility:   Product(QPS, QuerySuccessRatio, QueryLatencyScore);

Controller LoadBalancerEvoCtlr:
 ReqControllers: LoadBalancer[2];
 ReqOutputs:
   Sense:   LoadBalancer[2]->Stats,
            EnvironmentStats;
   Analyse: LoadBalancer[2]->ControllerUtility,
            EnvironmentSituation;
   Decide:  ControllerPlans[2];
   Act:     ControllerComposition(LoadBalancer(0)),
            ControllerComposition(LoadBalancer(1));
   Utility: Module:UtilityEstimator(
              LoadBalancer[2]->ControllerUtility);
```

Figure 9: Example Controller Specifications

5.5 Composition & Online Evolution

One of the key strengths of our framework is its ability to not only adapt and improve the operation of the controlled network entities, but also to *evolve* itself, *i.e.* adapt and improve its own functionality, as motivated in section 4.7. Functional composition, introduced in section 4.4, provides us with the flexibility to compose the most suitable controllers for each application and network environment. The flexible controller hierarchy (see section 4.3) employed by our framework in turn dictates which controllers are employed for what tasks and how they interact, within the boundaries defined by the controller specification. We discuss the evolution of individual controllers in section 5.5.1 and of the hierarchy as a whole in section 5.6. In section 5.7, finally, we describe the process of finding a better composition of controllers



(a) Domain (b) Composition #1 (c) Composition #2

Figure 10: Example search domain and two potential compositions within it. The graph boxes and dotted lines in this graph are not actually instantiated and are only given to highlight potential modules that could have been choosen.

and of the controller hierarchy.

5.5.1 Controller Evolution

Controllers are composed from the available modules by choosing either a new or existing instance of one module that provides a dependency of one of the nodes already in the graph and "plug" this new instance into the dependency "slot", as explained in section 5.1. This process continues until all dependencies of all modules in the controller are filled. We assume that the last requirements "layer" in this graph will be provided by the *source* module, which can be imagined as the mirror image of the aformentioned sink module: it provides all the data that the previous phase in the cognitive loop has made available.

The set of all potential valid controllers, that is all valid compositions of module instances and their configurations, defines the search space that needs to be traversed by the evolution process. The composition process is illustrated in fig. 10. The search domain of all valid compositions is represented by all possible paths from source to sink in fig. 10a. Two valid controller compositions are given in fig. 10b and fig. 10c, respectively. The evolution process can utilize any of the possible solutions within the search domain to instantiate a controller. Please refer to section 5.7 for details about this search process.

5.6 Hierarchy Evolution

As explained in section 5.3, evolution controllers are allowed to define how the subgraph of controllers below them is composed *at runtime*. The composition of the hierarchy of controllers therefore is an iterative process. The root of the controller hierarchy graph is known as the *master evolution controller*. It is instantiated first and informs the framework about its intentions regarding the composition of the sub-graph below it. The corresponding controller instance for the sub-graph layer below the master are then instantiated according to the requirements of the master.

This process is then iteratively continued for all underlying evolution controllers. As mentioned before, operation controllers do not have the liberty to choose there dependencies at runtime, instead their dependency graph is derived via an administrator-defined specification.

When an evolution controller decides to change the composition of the subgraph it manages, then the above process is applied to that subgraph alone.

The set of all potentially valid controller graphs (akin to all potential valid subgraphs of the module graph shown in fig. 10a) constitutes the search space to be traversed by the evolution process. Due to the excessive overhead incurred if all valid graphs were to be explored through instantiation, we utilize an iterative search process. Specifically, we instantiate a limited set of potential controllers, evaluate their performance, and refine our search process based on the knowledge gained by doing so. Similarly, we iteratively generate one controller hierarchy and let it operate unmodified until an evolution controller decides to change the hierarchy below it. The search space traversal itself is detailed in section 5.7.

5.7 Traversing the Composition Search Space

As introduced in sections 5.5.1 and 5.6, the composition of controllers, as well as of the controller hierarchy that is contains them, is an iterative process, and the search space that needs to be explored is a very complex and high-dimensional one. In fact, not even the number of dimensions is static. For the controller composition case, every connection between module instances, and every configuration parameter of each instance constitutes one dimension. If a different module is chosen, the number and types of parameters and dependencies changes Likewise, for controller hierarchies, each evolution controller can define the components, *i.e.* controllers, of its own subgraph and therefore the number of dimensions.

Approaches for discovering an optimal solution in such a complex envi-

ronment have already been explored in practice for the network stack composition case in our previous work[65], and we are confident that the techniques developed there can be utilized for both controller composition and hierarchy evolution. The former case is straight-forward and should pose no insurmountable issues. For the latter case, we acknowledge the need for research, and therefore defer a detailed discussion until later. We intend to provide a follow-up paper as soon as we can provide results validated in practice.

5.8 Online Experimentation

With the ability to evolve controllers programmatically, our framework can now automatically generate a large number of new controllers. However, to understand their utility or fitness as applied to some domain of control (section 4.6) we require *online experimentation*.

Within our framework, online trial-and-error experimentation is the responsibility of the *experimentation manager*. As motivated in section 4.7, we adopt a multi-layered approach to experimentation. First, new controllers are *sanity checked* to ensure that logical mistakes are not made. For example, a controller is trying to use a light sensor module where no such sensor exists. The use of taxonomies and ontologies (common sense) can be used to assist in this. Next, candidate controllers are tested in *simulation* to initially estimate their utility, optionally followed by deployment in a staging environment as needed. While not perfect, simulation tools, such as ns-3[99], iFogSim[52], or Naos[46], can serve as indicators of potential success or outright failure. Based on this information, the experiment manager can decide to move to the next stage, during which controllers will be gradually tested within the real production network.

For network trials, it is the responsibility of the experimentation manager to limit their (physical and temporal) scope, with gradual expansion based on the particular controller's measured (or estimated) utility.

As well as overseeing the experimental trials of a new controller, the experiment manager also acts as coordinator for different concurrent experimentations. This is necessary as there will be multiple ECs or MECs requesting experimentation of newly evolved controllers. Intuitively, not all experiments can be run concurrently as conflicts and false utility may be observed due to experiments interfering. Additionally, there is also the risk of instability, interference in high-gain operations, priorities of tests, *etc.* As noted in section 2, a telecommunications network is a large interconnected system meaning that interference is inevitable, however the experiment manager seeks to limit this. In this sense, the experimental manager is also acting as a *scheduler* and *resource allocator*,

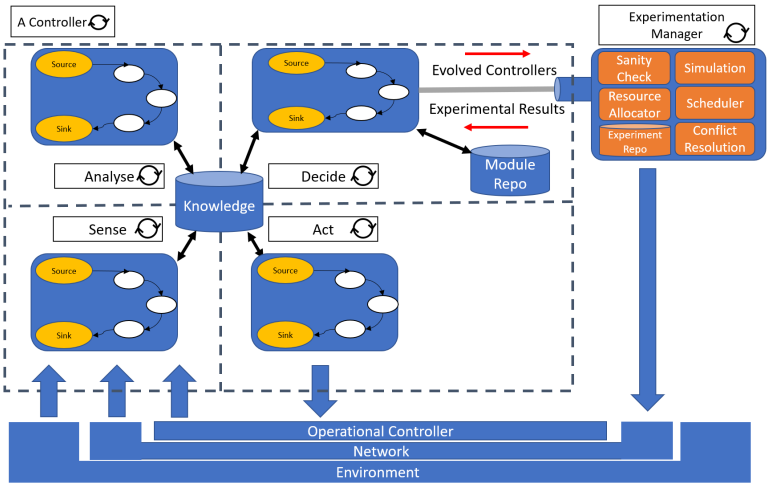


Figure 11: Evolution Controller and Experimentation Manager

More importantly, the experiment manager ensures that experiments are performed *independently* of each other. Similarly, the experiment manager needs to ensure that experiments are *fair*, *meaningful* and *representative* of the actual operation environment.

For example, consider the comparison of a messaging protocol running over TCP with another one which utilizes UDP, using a utility metric being based on throughput, latency and reliability.

As long as this experiment is performed over a reliable transport without packet loss or reordering, UDP will have an unfair advantage due to less overhead (no three-way handshake, smaller header size). However, once the transport becomes unreliable – be it through a change in the environment or another concurrent experiment affecting the transport layer – TCP's ability to retransmit and order incoming packets might put it into an advantageous position.

An experiment consists of the controller to test, its parameter configurations, utility functions, current experimental scope, and results. These results are used to determine if this controller under experimentation should replace an existing controller. Based on the provided information, potential conflicts in experimentation can be inferred.

A high level overview of how an evolution controller interacts with the experimentation manager as well as high level concepts associated with each of its phases is shown in fig. 11. Discussion of the specific concepts are deferred to the subsequent design document.

5.9 Summary

In this section we have described a high level overview of an architecture to apply evolutionary approaches and online experimentation to a modern telecommunications environment to achieve autonomous operation. By asking the user to provide well defined functional building blocks (modules) as well as the definition of "fitness" for the job (by means of a utility function), our framework can adapt to unseen situations as well as optimise known contexts.

6 Autonomous Networking Use Cases

The technologies introduced in section 4 combined with the high level architecture detailed in section 5 provide us with the means to implement a framework for autonomous networking. In this section, we provide two concrete use cases to illustrate how our framework can serve as the basis of a fully autonomous network. One concrete use case concerns autonomous network protocol stack evolution, which we already implemented and trialled in practice in the past. The other use case is related to mobile edge compute, which we are preparing to explore practically in the near future.

6.1 Network Protocol Stack evolution

Our work on autonomous networking is based on our previous experience with designing, developing and experimentally evaluating a system for autonomous network protocol stack evolution[65]. As the similarities to the framework presented in this paper might not be obvious, but are important to understand our contribution, we provide a short introduction of that work in comparison to our current approach below.

Autonomous network protocol stack evolution can be considered a subset of a fully autonomous network. In fact, finding the optimal composition and configuration of the protocol stack based on the operation environment the system is deployed in is one of the optimization use cases that our autonomy framework is expected to solve. Our approach then and now, is based on artificial evolution (using genetic algorithms, q-learning, *etc.*) based on an utility measure derived through online experimentation. We also utilized functional composition to invent and build new network stacks out of small modules that constituted either full protocol implementations or components thereof. For example, we decomposed the TCP protocol into its core features (sequentiality, reliability through retransmission, window scaling, selective acknowledgements, slow start, and so on), and thus allowed our system to derive new

versatile protocols that are similar to TCP, but outperform it under certain conditions. Likewise, we generalized all protocol implementations such that TCP or UDP could, for example, operate directly over Ethernet, omitting IP and thus overhead, if communicating with a directly reachable endpoint within the same local network segment. We encoded the *blueprints* of potential new stacks in such a way that learning algorithms could be applied and evolved separate *populations* of such stacks that were optimized for different situations (high load, low latency links, *etc.*) that would actually occur in the network. These situations were automatically classified using algorithms such as k-Means, *etc.* We did, however, not include the ability to evolve the composition evolution engine itself in our system back then. The algorithms utilized for different tasks would evolve autonomously, but the composition system itself was only applied to the network stack, not the framework.

In relation to the core technologies we present in this paper, many of the same concepts are present, have been realized, and tested in practice for the stack composition case. The cognitive control loop (controller) is manifested in the design of the evolution engine. There is, however, no hierarchy between controllers. Instead these components are a part of the framework itself. Meta-evolution is performed, but structural changes to the architecture of the evolution engine itself are out-of-scope of the implementation presented for the stack evolution framework. Functional composition is only applied to the protocol stacks themselves. Online evolution and experimentation, as well as situational classification are performed.

One of the major distinguishing factors of this work is that we obtained actual results from a practical implementation which we deployed in a real network. We developed and deployed several embedded devices in geographically separate locations, based on FreeBSD, TUN interfaces and raw sockets to take over and replace the network stack. We even implemented our own functional TCP/IP stack, since the available implementations were naturally not composable. Our results back then were promising but cut short due to lack of funding. Finally, we are now able to continue and vastly exceed our previous work.

6.2 Content Delivery Network

Content delivery networks (CDNs) geographically distribute content and services to improve latency, throughput, availability, and resilience for customers and end users that access the customers' content. Originally focussed on improving download and streaming performance through caching, CDNs nowadays provide many additional services such as mobile content acceleration, content transcoding, Distributed Denial of Service (DDoS) attack pro-

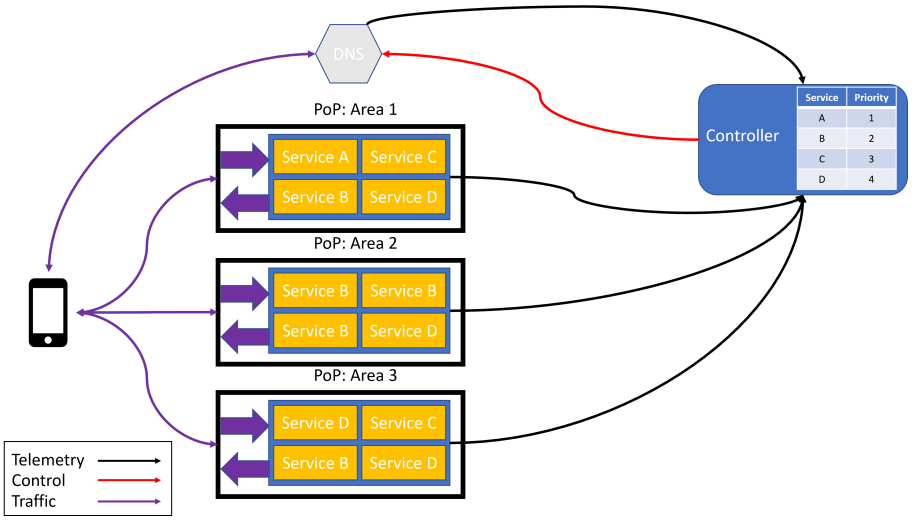


Figure 12: Simplified Traffic Load Balancing for CDN

tection, web application firewalls. Through the addition of compute functionality, CDNs are on the way to becoming mobile edge computation providers.

CDNs necessarily have to over-provision services to ensure high availability in case of failures (resiliency) and to cope with spikes in demand. Due to the inherent costs of this approach, optimization of where to place additional content copies – or to spin up software instances, for that matter – and how to dimension them is essential. We address these two topics separately in the future.

Providing the content in multiple places on its own, however, does not ensure that the load will be spread evenly – or better even, *optimally* – between the different locations. The reasons for this problem's existence, and how we solve it using our autonomy framework, is detailed below in section 6.2.1.

6.2.1 Traffic Load Balancing

Clients historically access content and services by means of transport-layer routing to the one server hosting the destination IP address. If the same content is to be provided in multiple places, this approach on its own is no longer sufficient. Common technologies to overcome this issue are IP Anycast[90] based on BGP and explicit load balancing based on the client IP via HTTP[93] or DNS[28].

Since user demand is dynamic, and in some cases even erratic, a static allocation would inherently be sub-optimal and needs to be avoided. For our

discussion below we thus assume a dynamic DNS based approach, where the approximate latency from each IP range to each serving location is known, and where each point of presence (PoP) is allocated its own IP range.

As shown in fig. 12, we utilize one global operation controller (OC), which distributes weights for each PoP and service to the distributed, anycasted DNS servers. DNS servers in turn report how many responses for each (service, PoP)-pair they have sent out during the last time cycle. Likewise, the PoPs' ingress and egress link loads and service utilization and capacities are also reported or known to the controller. Furthermore, the controller knows the importance or utility of each service, as well as its latency and bandwidth requirements.

If user demand could be reliably estimated, the logic for deciding where to redirect users to would be relatively straightforward. In reality, however, demand is hard to predict, especially since real world events, such as news, can sway users to access or stop accessing a service on short notice.

6.2.1.1 The Operational Controller An example of the requirements the global operation controller could fulfil is detailed in table 1. Based on these requirements, we specify modules in table 2 from which the OC is to be composed.

Phase	**Outputs**
Sensing	• number of incoming requests per IP range, per service • location, capacity and load of services • aggregation of results over a specified time period • measured latency between IP ranges and CDN locations • any additional data useful for predicting future user demand
Analysis	• updated latency based on measurement samples • estimated latency for unknown IP ranges • short-term predicted traffic based on incoming DNS requests • longer-term predicted traffic
Decision	• assignments (weights) per service, per location
Action	• assignments propagation to DNS servers

Table 1: Example output of the operational controller phases

For each module in table 2, a corresponding description in the specification language of its capabilities and interface has to be provided. An example set of module compositions for the sense phase is shown in fig. 13. The different example diagrams in this figure present a non-exhaustive list of possible

Phase	**Modules**
Sensing	• HTTP server request metric • Service deployment database • time series database • HTTP server request timing metric; RTT probe • news, twitter scraping, *etc.*
Analysis	• Latency estimator using mean latency over fixed period, sample length as parameter • BGP announcement based IP hole mitigator • DNS request based short-term traffic predictor • history-based long-term traffic predictor
Decision	• weight calculation heuristic employing a linear optimization solver
Action	• DNS configurator

Table 2: Example modules per phase

configurations of the modules in table 2.

It is important to note that the listed modules are not all uniquely associated with the task of traffic load balancing. Each of the sensing modules could also be used for a CDN health monitoring controller. Hence, module reuse is a key advantage of our approach.

Using this information, our framework has the basic inputs required to create and deploy controllers for the CDN traffic load balancing task. By using the analysis and sensing phases, the controller can understand how its actions (*i.e.* the weight assignments) impact the operation of the service for the current environmental state. Accordingly, the decision element can decide how to change the weights to be propagated to the DNS servers by the action phase. The utility function provides an easily comprehensible measure of how well the overall weight assignments matched the current conditions of the network. The general utility function we utilize is

$$\sum_{r \in R} u(l(r), t(r), s(r))$$

where R is the set of all requests, $l(r)$ is the latency measured for request r, $t(R)$ is the measured throughput, $s(r)$ is the service the request was destined for. u is an administrator-defined function which calculates the utility of each request based on the service value, the measured latency, and the measured throughput. To simplify the use case, we assume that the cost

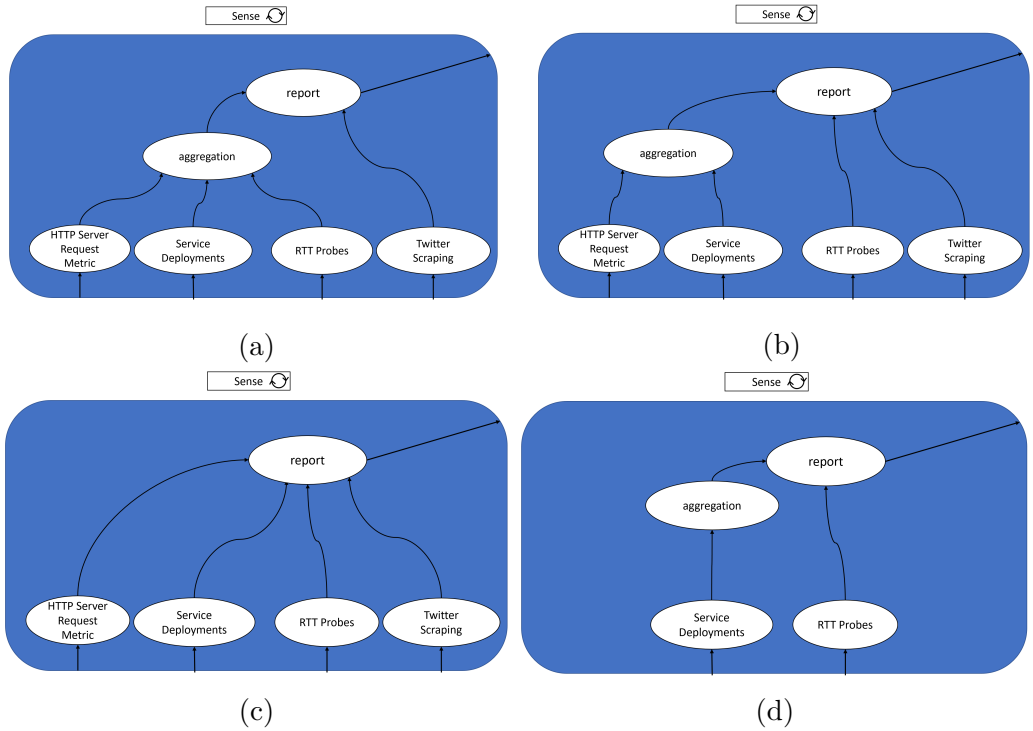


Figure 13: Example module compositions for the sense phase

of handling each request is identical and independent of the utilized bandwidth and location, which in practice might not be the case. The goal of the operation controller is to maximize the general utility function. The utility function utilized by each controller can be either expressed inline in the controller description or provided as a stand-alone code module.

Designing and re-designing the operation controller as needed based on its measured performance is the task of the evolution controller.

6.2.1.2 Evolution Controller In this use case, the evolution controller (EC) is not only responsible for ensuring that the OC is achieving its goal of efficiently load balancing the traffic, but also to come up with new OCs based on the available modules, which might outperform the existing OC. This requires current and historical data from the network, as well as the classification of the environment, the current blueprint of the OC, and a definition of the utility of the current OC. As the historical data implies, the EC will make decisions over a longer time period in this example. A list of example EC modules is shown in table 3. This table is again non-exhaustive, as the exact details of how to best achieve our goals requires further research.

Controller Element	Modules
Sensing	• general utility for each time cycle during which the current EC was active • situational characteristics at each time cycle
Analysis	• classification of situation for each time cycle • relative (as the achievable optimum is unknown) utility of EC for each encountered situation
Decision	• decide whether to come up with a new EC, which modules it should consist of, and how they should be configured using appropriate optimization techniques
Action	• evolve new controller • request controller experimentation • update knowledge base with experimental results

Table 3: Modules of the Evolutionary Controller For Traffic Load Balancing

If the OC is deemed insufficiently effective at its task for a given situation – as determined by the measured utility – the EC can decide to replace it with another existing or new OC. If the situation changes and a different controller has shown to outperform the current one under the new conditions, it might get chosen. If no sufficiently performant EC is known, a new composition is likely to be generated. A detailed discussion follows below.

6.2.1.3 Trial & Error Experimentation and Deployment For the case of online load balancing, trial & error experimentation is responsible for taking newly evolved controllers and seeing if they perform better at load balancing the traffic under the current conditions than other, existing controllers. As always, better is defined as delivering a higher utility score when measured using the same utility function as the current controllers. This stage is equivalent to continuous integration/continuous development (CI/CD)[115], acceptance testing[81], or build testing[15] found in other fields, just that the deployment decision it fully automated. We deploy new controls as follows:

First, we perform a *sanity check* to ensure that the new controller is complete. One sanity check is to ensure that all actions the controller may request to take are confined to the domain of load balancing and that actions are not being taken in other domains. This can be identified through the

action module capabilities and the network impact graph[4]. Another check can be to ensure that the new controller make progress, also known as not dead-locking.

Second, we verify through *simulation* that the controller operates as expected: We instantiate it in a sandbox environment, feed it recorded sensor data, and estimate how the weight changes it generates would impact the actual load-balancing, using the request log for the time period during which the new weights would have been applied, and calculate the utility of these changes. We also verify that sudden dramatic changes which would shift a lot of traffic from one location to another and oscillations in assignments are infrequent or non existing.

Third, we gradually deploy the new controller in the network, starting with low impact regions or for relatively unimportant services, and for a limited period of time. If the performance is poor, we roll back immediately before the allocated trial time is over. If the controller looks promising, we extend the trial to more important areas and services.

The above stages mimic the steps that a human would take, but can be fully automated. Whether we can fully remove the human from the loop, however, is a business, not a scientific decision.

6.3 Others

The above problems fit in the general category of resource allocation and scheduling. However, there are other problem areas within a telecommunications network. One such type is monitoring. This is where the autonomous network must deploy *monitor* operation controllers to collect information for either a specific or general task. Examples include network probes, inventory management, health checks, or anomaly detection. The latter is a meaningful sub category in itself and is concerned with the detection of *abnormal* activity in the network for a particular domain. Such detection can be for the purpose of security, optimisation, or troubleshooting.

Another problem type is problem solving and is the hardest challenge for an autonomous network. Having identified an anomalous situation above, the autonomous framework must identify the cause of the situation and seek to rectify it. In the telecommunications field this is known as *trouble ticketing* As noted in section 4.5, this goal is not currently achievable, and so instead we decompose the problem and seek to address it in stages.

[4]The network impact graph is a representation of the current state of the network and is a combination of network telemetry and ontological descriptions. Its discussion is deferred to future work.

Every time a problem occurs in the network, we can apply a classifier to detect the type of problem and select which graphs in the EMS[23] to present to the incident handler. We also record which other graphs this person is looking at while debugging and which actions they take to resolve them. After the problem is resolved, the incident handler is not only required to write the usual postmortem, but also to provide feedback regarding which graphs and which actions they performed where actually useful for resolving the problem. This information is then fed in to a (reinforcement) learning algorithm, which decides the actions to take in the future. Once enough data has been gathered, the next stage not only presents the most useful graph, but also suggests the potential problem cause. Once the problem is resolved, the responder has to provide feedback about how useful the suggestion was. In the final stage, the algorithm will also try to provide a resolution strategy, once confidence is high enough that the suggestion is likely correct.

Given our approach to autonomy, the above stages naturally fit into our design: the choice of learning algorithm to apply, the evaluation of the utility or fitness of the proposed solutions, and experimentation with different approaches. Accordingly, we deploy one operation controller for each stage described above. The composition of this controller will be evolved based on the feedback from the users via a machine learning approach embedded within the evolution controller associated with a particular stage. Whether a controller is actively operating the network or only learning based on feedback is decided by the satisfaction measure.

7 Conclusion

This document presents a concise and practical approach towards realizing true autonomy in the communication networks of the future. For this purpose we presented a truly autonomous, self-evolving framework for autonomous networking and telecommunications. We motivated the need for this approach and presented our vision for this autonomous future in section 2. Our vision is based on several core principles and key technologies, which we discussed and motivated in section 4. In conjunction, these core technologies clearly outline the road towards an architecture that will enable autonomy and that is flexible enough to encompass arbitrary future technologies, yet concise enough to be deployable in actual networks (summarized once more below). The architecture that directly derives from and manifests these concepts is explained in section 5.

The concept of the *cognitive loop* enables us to express arbitrary reasoning tasks in an *abstract* and *generic* way, and allows interaction and collabora-

tion between multiple independently designed control and optimization tasks. The *controller hierarchy* allows us to unify all of these tasks in a *holistic* manner, where higher-level cognitive loops supervise, control and *evolve* their subordinates. *Evolution* is realized by means of a *hybrid intelligent system*, that applies the appropriate cognition, learning and optimization strategies as needed. This feat of *online* evolution becomes possible thanks to functional composition and experimental evalution. *Functional composition* allows our framework to use small *functional building blocks* to compose and configure new and unique control entities on its own. These controllers are not limited to controlling and improving the operation of network infrastructure, but will also modify and improve the very architecture and functionality of the controlling systems themselves at runtime. It further enables our framework to seemlessly integrate new technologies and research output as they become available. *Experimental evolution* enables us to test and validate the performance of these *autonomously evolved controllers* in practice, within the actual network. The combination of all these technologies finally enables us to realize truly autonomous control and optimization in an *emergent* manner.

Our future work will consist of implementing this framework not only for the purpose of experimentation, but for production deployment with the aim of enriching the first fully deployed virtualized telecommunication network. While we expect the road towards this goal to be full of rocks and pitfalls, as we learned through our efforts on autonomous network stack evolution, we plan to leverage the experience we have gained in the past and are confident to overcome these issues collectively with our partners, to achieve the world's first truly *autonomous* network.

As telecommunication operators cannot and should not exist in isolation, but instead interact and interoperate extensively with each others, we obviously will try to integrate our own efforts with the main standardization bodies'. This way we hope be able to achieve truly autonomous networking, not just several approaches tailored to particular use cases. We believe that our own focus on a holistic and evolution-based approach that can *on its own* adapt to and optimize itself for *any* use case is superior to both a one-size-fits-all effort and to hand-crafted, per-use case designs.

References

1 3GPP. 36.902: Evolved universal terrestrial radio access network (e-utran); self-configuring and self-optimizing network (son) use cases and solutions. Technical report, 3rd Generation Partnership Project (3GPP), 2008.

2 3GPP. Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Self-configuring and self-optimizing network (SON) use cases and solutions. Technical Report (TR) 36.902, 3rd Generation Partnership Project (3GPP), 04 2011. Version 9.3.1.

3 3GPP. Public warning system (pws) requirements. Technical Report (TR) 22.268, 3rd Generation Partnership Project (3GPP), 2015.

4 The FP7 4WARD Project. `http://www.4ward-project.eu`.

5 Accenture. Digital Transformation Initative Telecommunications Industry. Technical report, Accenture, 2017.

6 Farah Ait Salaht, Frédéric Desprez, and Adrien Lebre. An overview of service placement problem in fog and edge computing. Research Report RR-9295, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, LYON, France, October 2019.

7 Autonomic network architecture project. `http://www.ana-project.org`.

8 Konstantinos Antonakoglou, Xiao Xu, Eckehard Steinbach, Toktam Mahmoodi, and Mischa Dohler. Toward haptic communications over the 5g tactile internet. *IEEE Communications Surveys and Tutorials*, 20(4):3034–3039, 2018.

9 Gobinath Aroganam, Nadarajah Manivannan, and David Harrison. Review on wearable technology sensors used in consumer sport applications. *Sensors*, 19(9):1983, 2019.

10 Ozalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen. *Self-Star Properties in Complex Information Systems: Conceptual and Practical Foundations (Lecture Notes in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 2005.

11 Md Faizul Bari, Raouf Boutaba, Rafael Esteves, Lisandro Zambenedetti Granville, Maxim Podlesny, Md Golam Rabbani, Qi Zhang, and Mohamed Faten Zhani. Data center network virtualization: A survey. *IEEE Communications Surveys & Tutorials*, 15(2):909–928, 2012.

12 Kim Baroudy, Sunil Kishore, Nitin Mahajan, Sumesh Nair, Halldor Sigurdsson, and Kabil Sukumar. Reinventing telco networks: Five elements of a successful transformation, 2019.

13 Yoshua Bengio, Samy Bengio, and Jocelyn Cloutier. *Learning a synaptic learning rule*. Université de Montréal, 1990.

14 Yoshua Bengio, Tristan Deleu, Nasim Rahaman, Rosemary Ke, Sébastien Lachapelle, Olexa Bilaniuk, Anirudh Goyal, and Christopher Pal. A meta-transfer objective for learning to disentangle causal mechanisms. *arXiv preprint arXiv:1901.10912*, 2019.

15 Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

16 John Charles Bicket. Bit-rate selection in wireless networks. Master's thesis, Massachusetts Institute of Technology, 2005.

17 Aaron Richard Earl Boasman-Patel, Dong Sun, Ye Wang, Christian Maitre, Jose Domingos, Yannis Troullides, Ignacio Mas, Gary Traver, and Guy Lupo. Autonomous Networks: Empowering Digital Transformation For The Telecoms Industry - TM Forum, 2019.

18 G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May. The autonomic network architecture (ana). *Selected Areas in Communications, IEEE Journal on*, 28(1):4–14, 2010.

19 J. Boyd, G.T. Hammond, and Air University (U.S.). Press. *A Discourse on Winning and Losing*. Air University Press, Curtis E. LeMay Center for Doctrine Development and Education, 2018.

20 R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1):14–23, March 1986.

21 Thomas Bäck. *Evolutionary Algorithms in Theory and Practice - Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996.

22 X. Cao, J. Zhang, and H. V. Poor. An optimal auction mechanism for mobile edge caching. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 388–399, July 2018.

23 Ciena. Manage, Control and Plan: Unleash the programmability of your Ciena network, 2020.

24 Cisco. The Cisco Digital Network Architecture - An Overview, 2016.

25 David D. Clark, Craig Partridge, J. Christopher Ramming, and John T. Wroclawski. A knowledge plane for the internet. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–10, New York, NY, USA, 2003. ACM.

26 Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design, IWSSD 1996*, pages 16–25. Association for Computing Machinery, Inc, mar 1996.

27 Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 31(2006):1–6, 2006.

28 C. Contavalli, W. van der Gaast, D. Lawrence, and W. Kumari. Rfc 7871: Client subnet in dns queries. `https://tools.ietf.org/rfc/rfc7871.txt`.

29 James Crawshaw. How BT Is Applying AI to Support Its Programmable Network, 2019.

30 Nikolay Dandanov, Hussein Al-Shatri, Anja Klein, and Vladimir Poulkov. Dynamic self-optimization of the antenna tilt for best trade-off between coverage and capacity in mobile networks. *Wirel. Pers. Commun.*, 92(1):251–278, January 2017.

31 G. Davis. 2020: Life with 50 billion connected devices. In *2018 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–1, Jan 2018.

32 Chrysanthos Dellarocas, Mark Klein, and Howard Shrobe. An architecture for constructing self-evolving software systems. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 29–32, New York, NY, USA, 1998. ACM.

33 P. Demestichas, V. Stavroulaki, D. Boscovic, A. Lee, and J. Strassner. m@angel: autonomic management platform for seamless cognitive connectivity to the mobile internet. *Communications Magazine, IEEE*, 44(6):118–127, June 2006.

34 Simon Dobson, Eoin Bailey, Stephen Knox, Ross Shannon, and Aaron Quigley. A first approach to the closed-form specification and analysis of an autonomic control system. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer*

Systems, pages 229–237, Washington, DC, USA, 2007. IEEE Computer Society.

35 Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, 2006.

36 M. Dorigo and G. Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation - CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477, July 1999.

37 Jun Duan and Yuanyuan Yang. A load balancing and multi-tenancy oriented data center virtualization framework. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2131–2144, 2017.

38 Harald Eckhardt, Siegfried Klein, and Markus Gruber. Vertical antenna tilt optimization for lte base stations. In *2011 IEEE 73rd Vehicular Technology Conference (VTC Spring)*, pages 1–5, 05 2011.

39 Mohamad Eid, Ramiro Liscano, and Abdulmotaleb El Saddik. A novel ontology for sensor networks data. In *Proceedings of 2006 IEEE International Conference on Computational Intelligence for Measurement Systems and Applications, CIMSA 2006*, pages 75–79, 2006.

40 ETSI. Experiential networked intelligence (eni); system architecture. Technical Report GS ENI 005, ETSI, 2019. Version 1.1.1.

41 ETSI. Network functions virtualisation (nfv); terminology for main concepts in nfv. Technical Report GR NFV 003, ETSI, jan 2020. Version 1.5.1.

42 Dave Evans. The internet of things – how the next evolution of the internet is changing everything. `www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`, 2011.

43 Oldrich Faldik, Richard Payne, John Fitzgerald, and Barbora Buhnova. Modelling system of systems interface contract behaviour. *arXiv preprint arXiv:1703.07037*, 2017.

44 Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, April 2014.

45 Lee Feigenbaum, Ivan Herman, Tonya Hongsermeier, Eric Neumann, and Susie Stephens. The semantic web in action. *Scientific American*, 297(6):90–97, 2007.

46 Forsk. Naos Overview, 2020.

47 Xenofon Foukas, Georgios Patounas, Ahmed Elmokashfi, and Mahesh K. Marina. Network Slicing in 5G: Survey and Challenges. *IEEE Communications Magazine*, 55(5):94–100, may 2017.

48 Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinenDenkens*. Verlag von Louis Nebert, Halle, 1879.

49 Kunihiko Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, 20(3-4):121–136, 1975.

50 A.G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

51 Erann Gat, R. Peter Bonnasso, Robin Murphy, and Aaai Press. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, 1997.

52 Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

53 Richard G Guy, John S Heidemann, Wai Mak, Thomas W Page Jr, Gerald J Popek, Dieter Rothmeier, et al. Implementation of the ficus replicated file system. In *USENIX Conference Proceedings*, volume 74, pages 63–71. Citeseer, 1990.

54 Amelie Gyrard. An architecture to aggregate heterogeneous and semantic sensed data. In *Extended Semantic Web Conference*, volume 7882, pages 697–701. Springer, 05 2013.

55 M. Harris. *End Of Absence: Reclaiming What We've Lost in a World of Constant Connection*. Harper Perennial, 2014.

56 Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, June 1949.

57 Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 1st edition, 2017.

58 Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, USA, 2011. USENIX Association.

59 Cheol-Ho Hong and Blesson Varghese. Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms. *ACM Comput. Surv.*, 52(5), September 2019.

60 Huawei. Moving towards autonomous driving networks, 2019.

61 Huawei. Moving towards autonomous driving networks, 2019.

62 Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng.*, 17(1):64–76, 1991.

63 Internet Assigned Numbers Authority (IANA). Internet protocol numbers. `https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml`.

64 P. Imai and C. Tschudin. Practical online network stack evolution. In *Self-Adaptive and Self-Organizing Systems Workshop (SASOW), 2010 Fourth IEEE International Conference on*, pages 34–41, 2010.

65 Pierre Imai. *Exploring Online Evolution of Network Stacks*. PhD thesis, University of Basel, Switzerland, 2015.

66 S. M. R. Islam, D. Kwak, M. H. Kabir, M. Hossain, and K. Kwak. The internet of things for health care: A comprehensive survey. *IEEE Access*, 3:678–708, 2015.

67 ITU. Architectural framework for machine learning in future networks including IMT-2020. Technical report, ITU, 2019.

68 ITU-R. Imt traffic estimates for the years 2020 to 2030. *Report ITU-R M. 2370–0, ITU-R Radiocommunication Sector of ITU*, 2015.

69 Jesse V. Jacobs, Lawrence J. Hettinger, Yueng-Hsiang Huang, Susan Jeffries, Mary F. Lesch, Lucinda A. Simmons, Santosh K. Verma, and Joanna L. Willetts. Employee acceptance of wearable technology in the workplace. *Applied Ergonomics*, 78:148–156, 2019.

70 M. Jalasri and L. Lakshmanan. A Survey: Integration of IoT and Fog Computing. In *2018 Second International Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 235–239, Aug 2018.

71 Ahmad Javaid, Quamar Niyaz, Weiqing Sun, and Mansoor Alam. A deep learning approach for network intrusion detection system. In *Proceedings of the 9th EAI International Conference on Bio-Inspired Information and Communications Technologies (Formerly BIONETICS)*, BICT'15, page 21–26, Brussels, BEL, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

72 Juniper Networks. The Self-Driving Network.

73 Raja Jurdak, Cristina Videira Lopes, and Pierre Baldi. A framework for modeling sensor networks. In *Proceedings of the Building Software for Pervasive Computing Workshop at OOPSLA*, volume 4, pages 1–5, 2004.

74 John E. Laird. *The Soar Cognitive Architecture*. The MIT Press, 2012.

75 Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Iowa State University, 1998.

76 Christiane Lemke, Marcin Budka, and Bogdan Gabrys. Metalearning: a survey of trends and technologies. *Artificial intelligence review*, 44(1):117–130, 2015.

77 Pei-Ying Lin, Hsiao-Ting Chiu, and Rung-Hung Gau. Machine learning-driven optimal proactive edge caching in wireless small cell networks. In *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*, pages 1–6, 04 2019.

78 Lorena Isabel Barona López, Ángel Leonardo Valdivieso Caraguay, Luis Javier García Villalba, and Diego López. Trends on virtualisation with software defined networking and network function virtualisation. *IET Networks*, 4(5):255–263, 2015.

79 Lu Lu, Geoffrey Ye Li, A Lee Swindlehurst, Alexei Ashikhmin, and Rui Zhang. An overview of massive mimo: Benefits and challenges. *IEEE journal of selected topics in signal processing*, 8(5):742–758, 2014.

80 A. M. Maia, Y. Ghamri-Doudane, D. Vieira, and M. F. de Castro. Optimized placement of scalable iot services in edge computing. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 189–197, April 2019.

81 Roy Miller and Christopher T Collins. Acceptance testing. *Proc. XPUniverse*, 238, 2001.

82 Thomas A. Milligan. *Modern Antenna Design, 2nd ed.* John Wiley & Sons, Ltd, 2005.

83 Marvin Minsky. Logical versus analogical or symbolic versus connectionist or neat versus scruffy. *AI Mag.*, 12(2):34–51, April 1991.

84 Marvin Minsky and Seymour Papert. Perceptrons: An essay in computational geometry. *MIT Press*, 1969.

85 J. Mitola. Cognitive radio architecture evolution. *Proceedings of the IEEE*, 97(4):626–641, April 2009.

86 Allen Newell, John C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *Proceedings of the International Conference on Information Processing*, pages 256–264, 1959.

87 Amy Nordrum. Popular internet of things forecast of 50 billion devices by 2020 is outdated (2016). *IEEE Spectrum: Technology, Engineering, And Science News*, aug 2016.

88 Nuance. The Telco industry's quest for digital transformation. Technical report, Nuance, 2017.

89 Frédéric Ogel, Simon Patarin, Ian Piumarta, and Bertil Folliot. C/span: a self-adapting web proxy cache. In *Autonomic Computing Workshop. 2003. Proceedings of the*, pages 178–185. IEEE, 2003.

90 C. Partridge, T. Mendez, and W. Milliken. Rfc 1546: Host anycasting service. `https://tools.ietf.org/rfc/rfc1546.txt`.

91 Simon Patarin, Simon Patarin, Mesaac Makpangou, Mesaac Makpangou, and Simon Pat. Pandora: A flexible network monitoring platform. In *In Proceedings of the USENIX 2000 Annual Technical Conference*, pages 200–0, 2000.

92 Vern Paxson and Sally Floyd. Why we don't know how to simulate the internet. In *Proceedings of the 29th conference on Winter simulation*, pages 1037–1044. IEEE Computer Society, 1997.

93 A. Petersson and M. Nilsson. Rfc 7239: Forwarded http extension. `https://tools.ietf.org/rfc/rfc7239.txt`.

94 Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. {REX}: A development platform and online learning approach for runtime emergent software systems. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 333–348, 2016.

95 R. Presuhn, J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Rfc 3418: Management information base (mib) for the simple network management protocol (snmp). `https://tools.ietf.org/rfc/rfc3418.txt`.

96 Prometheus. Prometheus open-source monitoring solution. `https://prometheus.io/`.

97 Juan J. Ramos-Munoz, Lidia Yamamoto, and Christian Tschudin. Serial experiments online. In *ACM SIGCOMM Computer Communication Review*, volume 38 (2), pages 31–42. ACM, April 2008.

98 Fano Ramparany and Quyet H. Cao. A semantic approach to IoT data aggregation and interpretation applied to home automation. In *2016 International Conference on Internet of Things and Applications (IOTA)*, Prune, India, 2016.

99 George F. Riley and Thomas R. Henderson. *The ns-3 Network Simulator*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

100 Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63:311–324, 1984.

101 Frank Ritter, Farnaz Tehranchi, and Jacob Oury. Act-r: A cognitive architecture for modeling cognition. *Wiley Interdisciplinary Reviews: Cognitive Science*, 10:e1488, 12 2018.

102 P. Rost, C. Mannweiler, D. S. Michalopoulos, C. Sartori, V. Sciancalepore, N. Sastry, O. Holland, S. Tayade, B. Han, D. Bega, D. Aziz, and H. Bakker. Network slicing to enable scalability and flexibility in 5g mobile networks. *IEEE Communications Magazine*, 55(5):72–79, May 2017.

103 Vitor Rozsa, Marta Denisczwicz, Moisés Lima Dutra, Parisa Ghodous, Catarina Ferreira da Silva, Nader Moayeri, Frédérique Biennier, and Nicolas Figay. An application domain-based taxonomy for iot sensors. In *ISPE te*, pages 249–258, 2016.

104 S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd edition*, chapter Introduction, pages 1–5. Prentice-Hall, 2003.

105 S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd edition*, chapter Learning, pages 736–748. Prentice-Hall, 2003.

106 S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd edition*, chapter Introduction, page 17. Prentice-Hall, 2003.

107 S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd edition*, chapter Artificial Intelligence, page 51. Prentice-Hall, 2003.

108 S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd edition*, chapter Problem Solving, pages 111–112. Prentice-Hall, 2003.

109 S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd edition*, chapter Problem Solving, page 115. Prentice-Hall, 2003.

110 S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd edition*, chapter Problem Solving, pages 116–119. Prentice-Hall, 2003.

111 S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd edition*, chapter Learning, page 772. Prentice-Hall, 2003.

112 J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

113 Jürgen Schmidhuber. Evolutionary principles in self-referential learning. Master's thesis, Technische Universität München, 1987.

114 Jürgen Schmidhuber. Ultimate cognition à la gödel. *Cognitive Computation*, 1(2):177–193, 2009.

115 Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

116 Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944*, 2019.

117 Herbert A. Simon. *The Sciences of the Artificial*, pages 51–52. MIT Press, Cambridge, MA, 3 edition, 1996.

118 Meryem Simsek, Adnan Aijaz, Mischa Dohler, Joachim Sachs, and Gerhard Fettweis. 5g-enabled tactile internet. *IEEE Journal on Selected Areas in Communications*, 34(3):460–473, 2016.

119 Mikhail Smirnov. Autonomic communication—research agenda for a new communication paradigm. company whitepaper. *Fraunhofer Institute for Open Communication Systems, Berlin, Germany*, 2004.

120 R Stahlmann, A Festag, A Tomatis, I Radusch, and F Fischer. Starting european field tests for car-2-x communication: the drive c2x framework. In *18th ITS World Congress and Exhibition*, page 12, 2011.

121 John Strassner, Nazim Agoulmine, and Elyes Lehtihet. Focale: A novel autonomic networking architecture. *Latin American Autonomic Computing Symposium (LAACS)*, 2006.

122 R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction, 2nd ed*, chapter Introduction, pages 1–3. MIT Pressl, 2017.

123 R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction, 2nd ed*, chapter Multi-armed Bandits, pages 19–35. MIT Pressl, 2017.

124 Takafumi Tanaka, Akira Hirano, Shoukei Kobayashi, Takuya Oda, Seiki Kuwabara, Andrew Lord, Paul Gunning, Oscar Gonzalez De Dios, Victor Lopez, Arturo Mayoral Lopez De Lerma, and Antonio Manzalini. Publisher's note: Autonomous network diagnosis from the carrier perspective [Invited] (Journal of Optical Communications and Networking (2020) 12 (A9-A17) DOI: 10.1364/JOCN.12.0000A9), mar 2020.

125 TMForum. Application Framework (TAM).

126 TMForum. Buisness Process Framework (eTOM).

127 G.D. Troxel, Eric Blossom, Steve Boswell, Armando Caro, I. Castineyra, Alex Colvin, Tad Dreier, Joseph B. Evans, N. Goffee, K.Z. Haigh, Talib Hussain, V. Kawadia, David Lapsley, Carl Livadas, Alberto Medina, Joanne Mikkelson, Gary J. Minden, R. Morris, C. Partridge, Vivek

Raghunathan, Ram Ramanathan, Cesar Santivanez, Thomas Schmid, Dan Sumorok, M. Srivastava, Robert S. Vincent, D. Wiggins, Alexander M. Wyglinski, and Sadaf Zahedi. Adaptive dynamic radio open-source intelligent team (adroit): Cognitively-controlled collaboration among sdr nodes. In *Networking Technologies for Software Defined Radio Networks, 2006. SDR '06.1st IEEE Workshop on*, pages 8–17, September 2006.

128 Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. Intrusion detection by machine learning: A review. *expert systems with applications*, 36(10):11994–12000, 2009.

129 Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

130 R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, A. Al-Nemrat, and S. Venkatraman. Deep learning approach for intelligent intrusion detection system. *IEEE Access*, 7:41525–41550, 2019.

131 W3C. W3c data activity. `http://www.w3.org/2013/data/`.

132 W3C. W3c owl. `http://www.w3.org/TR/owl-features/`.

133 W3C. W3c semantic web activity. `https://www.w3.org/2001/sw/`.

134 Alfred North Whitehead and Bertrand Arthur William Russell. *Principia mathematica; 2nd ed.* Cambridge Univ. Press, Cambridge, 1927.

135 Norbert Wiener. *Cybernetics*. Technology Press, 1949.

136 D. Wisely, N. Wang, and R. Tafazolli. Capacity and costs for 5g networks in dense urban areas. *IET Communications*, 12(19):2502–2510, 2018.

137 David Wood, Jerome S. Bruner, and Gail Ross. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry*, 17(2):89–100, 1976.

138 William Xu. Huawei Global Industry Vision. Technical report, Huawei, 2018.

139 J. Yang, M. Ding, G. Mao, Z. Lin, D. Zhang, and T. H. Luan. Optimal base station antenna downtilt in downlink cellular networks. *IEEE Transactions on Wireless Communications*, 18(3):1779–1791, March 2019.

140 Zhiwen Yu, Xingshe Zhou, and Yuichi Nakamura. Semantic learning space: An infrastructure for context-aware ubiquitous learning. In FrodeEika Sandnes, Yan Zhang, Chunming Rong, LaurenceT. Yang, and Jianhua Ma, editors, *Ubiquitous Intelligence and Computing*, volume 5061 of *Lecture Notes in Computer Science*, pages 131–142. Springer Berlin Heidelberg, 2008.