NaaS: Network-as-a-Service in the Cloud

Paolo Costa[†] Matteo Migliavacca[*] Peter Pietzuch[†] Alexander L. Wolf[†]

[†]*Imperial College London* [*]*University of Kent*

Abstract

Cloud computing realises the vision of utility computing. Tenants can benefit from on-demand provisioning of computational resources according to a pay-per-use model and can outsource hardware purchases and maintenance. Tenants, however, have only limited visibility and control over network resources. Even for simple tasks, tenants must resort to inefficient overlay networks.

To address these shortcomings, we propose *Network-as-a-Service* (NaaS), a framework that integrates current cloud computing offerings with direct, yet secure, tenant access to the network infrastructure. Using NaaS, tenants can easily deploy custom routing and multicast protocols. Further, by modifying the content of packets on-path, they can efficiently implement advanced network services, such as in-network data aggregation, redundancy elimination and smart caching.

We discuss applications that can benefit from NaaS, motivate the functionality required by NaaS, and sketch a possible implementation and programming model that can be supported by current technology. Our initial simulation study suggests that, even with limited processing capability at network switches, NaaS can significantly increase application throughput and reduce network traffic.

1 Introduction

Most cloud applications are *distributed* by nature and often involve significant network activity to perform their operations. Yet, in today's cloud computing offerings, tenants have little or no control over the network. Tenants can accurately select the number and types of computational and storage resources needed for their applications, but they cannot directly access and manage the network infrastructure (i.e., routers or switches). This means that all packet processing must occur at the end hosts. Even some relatively common communication operations, such as a multicast, are not supported, requiring tenants to implement them using inefficient application-level overlays.

Triggered by lowering costs and increasing performance, there has been a growing interest in software- [20, 21, 26, 32, 33] and FPGA-based [30] programmable routers. These proposals aim to replace the traditional switch and router operations (e.g., IPv4 forwarding) with a custom implementation. We argue to go one step farther: *the flexibility provided by these implementations should be offered to tenants to implement part of the application logic in the cloud network.*

In this paper, we motivate and describe *Network-as-a-Service* (NaaS), a new cloud computing model in which tenants have access to additional computing resources collocated with switches and routers. Tenants can use NaaS to implement custom *forwarding* decisions based on application needs, for example a load-balancing anycast or a custom multicast service. They can *process* packets on-path, possibly modifying the payload or creating new packets on the fly. This enables the design of efficient in-network services, such as data aggregation, stream processing, caching and redundancy elimination protocols, that are *application-specific* as opposed to traditional application-agnostic network services.

In the next section we consider some existing applications that would benefit from NaaS. Based on this, in Section 3 we discuss the functional requirements that we see for NaaS and sketch a possible NaaS architecture and programming model. In Section 4, using large-scale simulations, we show that a modest processing capability at switches is sufficient to significantly increase performance and overall data centre throughput. Finally, in Section 5 we conclude with brief remarks on the future of NaaS.

2 Applications

The traditional model offered by cloud providers rigidly separates computation in the end hosts from end-to-end routing in the network. We argue that this separation hurts both *performance* in the data plane and *flexibility* in the control plane of the network fabric.

For example, applications that try to disseminate or

collect information using a black-box network can waste bandwidth. *Broadcast/multicast* [14, 34], *content-based networking* [13] and *content-centric networking* [23] services must send multiple copies of packets to all interested parties. By instead operating within the network, they can conserve bandwidth by duplicating packets for multiple destinations as late as possible. Applications that perform *complex event processing*, *stream processing* [16] or specialised *data aggregation* [4, 36] can compute in-network aggregates incrementally, avoiding the collection of all data at the destination end host, thereby greatly reducing network traffic. Similarly, applications that use caching mechanisms, such as *redundancy elimination* [8] and memcached [3], can improve their efficiency by operating at internal network nodes instead of at end hosts.

Some applications address these issues by inferring the physical data centre network topology and creating an overlay network that is optimised for the communication patterns of the application [14]. Discovering the network topology at the application level, however, is imprecise and introduces probing overhead. More fundamentally, a perfect mapping cannot be achieved. For instance, rack-level aggregations cannot be computed directly by top-of-rack (ToR) switches, but instead require computation by an end host in a rack, leading to increased latency and lower throughput due to the limited network capacity.

The current cloud model also struggles to provide application-specific services that require flexibility in the control plane. It forces tenants to circumvent limitations, thus increasing development effort. For example, *firewalls* and *intrusion detection systems* cannot be customised to the set of applications deployed by tenants; *traffic engineering* requires control over the scheduling of packets; and *multipath* flows require steering of packets across different paths. Implementing such features in current data centres is challenging. For instance, Orchestra [14] uses multiple TCP flows to mimic traffic reservation policies, and multipath TCP [31] relies on VLAN tags or ECMP to exploit multiple paths between end hosts.

Table 1 summarises the applications mentioned above. The second column lists the current solutions used by those applications to work around network limitations. In the remaining columns, we analyse the requirements that these applications have in terms of their in-network memory footprint, packet-processing order dependencies, and required functionality.

Applications vary as to the amount of in-network memory that they require. *In-network caching*, *complex event processing* and *stream processing* typically have a large memory footprint (MBs or GBs) because they must maintain application data for long periods of time. *Data aggregation*, *firewalls*, and *content-based networking* need less memory, only storing temporary results or rule policies (MBs). *Packet scheduling*, *multipath routing* and *multicast* have the lowest need for memory because they only maintain comparatively small routing tables and per-flow statistics. To support the most demanding applications a NaaS facility must be provided with adequate memory resources and the ability to manage their allocation.

Application	Current approach	Mem	Per-packet state	Packet operation
Broadcast/multicast [14, 34]	overlay	KBs	✗	duplicate
Content-based netw. [13]	overlay	MBs	✗	duplicate
Content-centric netw. [23]	overlay	MBs	✗	duplicate
Complex event processing	overlay	GBs	✓	modify
Stream processing [16]	overlay	GBs	✓	modify
Data aggregation [4, 36]	overlay	MBs	✓	modify
Deduplication [8]	end-to-end	GBs	✓	modify
Distributed caching [3]	end-to-end	GBs	✓	modify
Information flow control [29]	middlebox	KBs	✗	forward
Stateful firewalls/IDS	middlebox	MBs	✓	forward
Packet scheduling [14]	network	KBs	✓	forward
Multipath [31]	network	KBs	✗	forward
Load-aware anycast	various	KBs	✓	forward

Table 1: Taxonomy of data centre applications that would benefit from in-network processing.

NaaS implementations should exploit hardware parallelism when processing packets within the network. Applications that make use of stateless packet operations, such as *multicast*, *multipath routing*, or filtering (i.e., *content-based networking* and *stateless firewalls*) are easily parallelised by processing packets on multiple cores. On the other hand, applications that update state on each packet are more challenging to support.

Applications pose various demands on the network in terms of supported packet operations, as shown in the last column of Table 1. A first class of applications wants control over packet *forwarding*, potentially deciding on the prioritisation of the output packets. Content dissemination (i.e., *multicast* and *content-based networking*) must *duplicate* packets. The most general class of applications needs the ability to process incoming packets arbitrarily by modifying them or creating new packets. An architecture for realising a NaaS model must therefore support these different in-network functions.

3 NaaS Overview

The goal of the NaaS model is to enable tenants to use the network infrastructure in a data centre (DC) more efficiently, addressing the shortcomings discussed in the previous section. In this section, we first motivate the functionality that the NaaS platform should offer, then outline the requirements that the implementation should fulfil, and finally discuss our proposed architecture and programming model.

3.1 Functionality

For simplicity, we present the three functions of NaaS separately, although in practice they are used together.

F1: Network visibility. Many of the applications presented in Section 2 are built on top of overlay networks. To achieve high performance, great effort must be made to optimise the mapping between the logical and physical topologies. Since existing DCs are characterised by high degrees of over subscription [10], taking into account rack locality in the overlay layout can have a significant impact on performance. Several solutions for inferring network locality have been proposed. For example, Orchestra [14] uses a sophisticated clustering protocol to discover the DC topology and leverages this information to efficiently lay down its tree-based overlay.

While black-box approaches are necessary in an open environment such as the Internet, the DC provider has an accurate knowledge of the topology and could make this information available to tenants at no extra cost. This would allow tenants to efficiently allocate overlay nodes to VMs, without requiring expensive and often inaccurate probing solutions.

F2: Custom forwarding. Network visibility would yield a significant performance improvement for overlay-based applications. However, there are some fundamental limits to the performance achievable using overlay networks. Since servers have usually only one NIC, even a simple multicast tree with a fan out greater than one cannot be optimally mapped to the physical network.

Therefore, the second functionality that NaaS should provide is the ability to control packet forwarding at switches. This would allow the implementation of custom routing protocols. All the applications in Table 1 that are tagged as *duplicate* or *forward* would greatly benefit from this functionality. Examples include content-based and content-centric networking, but also tenant-specific firewalls, packet scheduling and load-aware anycast.

F3: In-network processing. The main benefits of NaaS come from providing in-network packet processing capabilities as part of the cloud computing platform. For instance, distributed computing platforms, such as MapReduce [18] and Dryad [22], as well as real-time streaming systems and search engines [1, 2, 4, 11] operate on large amounts of data that are often aggregated between stages. By performing in-network aggregation, it is possible to reduce the overall traffic sent over the network, thereby significantly reducing execution times [15]. Note that these aggregation functions are application-specific and, hence, could not be provided as a traditional network service.

Another application that would benefit from this functionality is a distributed caching service, similar to memcached [3]. For example, by leveraging the ability to intercept packets on-path, it would be possible to implement opportunistic caching strategies based on how many times a packet (or a collection of packets representing a given item) has been seen by a given switch.

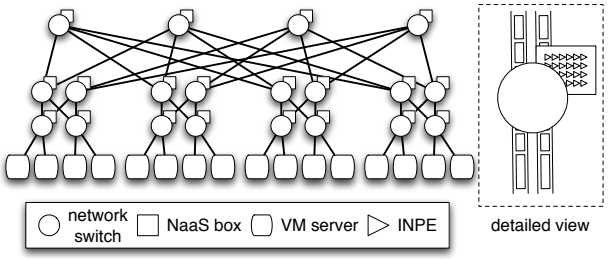


Figure 1: NaaS architecture in a data centre.

3.2 Requirements

For a NaaS model to be used in DCs, we believe that the following requirements must be satisfied:

R1: Integration with current DC hardware. Existing DCs constitute a significant investment. The use of commodity networking equipment, which typically lacks programmability features, reduces the cost of large DC deployments. For NaaS to become successful, it must not require expensive, non-commodity, networking hardware.

R2: High-level programming model. NaaS should expose a programming model that is natural for software developers to use, hiding low-level details of network packet processing and not exposing the full complexity of the physical network topology in the DC.

R3: Scalability and multi-tenant isolation. Compared to existing software-based router solutions [20, 21, 26, 32, 33], NaaS must be able to support a multitude of different applications, written by different organisations and running concurrently, unaware of each other. Therefore, to be successful, a NaaS model requires strong isolation of the different network resources offered to tenants.

3.3 Architecture

Figure 1 shows an example of the NaaS architecture. Compared to existing cloud data centres, NaaS requires that network devices be able to (efficiently) execute tenant code. To logically separate the functionality of traditional packet switching from the more advanced NasS functionality, we colloquially refer to the component responsible of executing tenant code as a *NaaS box*. NaaS boxes can be implemented as separate devices connected via high-bandwidth links to the switches or could be integrated into the same switch hardware.

NaaS boxes host instances of *in-network processing elements* (INPEs) that carry out application-specific processing of the packets that flow through them. For a given application run by a tenant, each INPE executes the same application logic on a subset of all NaaS boxes, i.e., the ones along the routing paths between the tenant's VMs in the physical network topology.

Implementation. A seemingly natural choice to implement NaaS boxes would be to use OpenFlow switches. However, OpenFlow typically assumes that only a tiny fraction of the packets (e.g., only SYN packets) are processed in software. Therefore, most commercially available switches support very low bandwidth to the OpenFlow controller (e.g., the HP 5406zl supports only 17 Mbps [17]). Also, the expressiveness of OpenFlow filters is too limited to implement complex matching such as that required by content-based networking [13] and their usage is typically restricted to packet headers.

NetFPGA [30] is at the other extreme in terms of packet processing performance. By taking advantage of hardware programmability, NetFPGA can support fast packet processing (up to 40 Gbps in the latest release). However, the programming languages used by NetFPGA, e.g., VHDL or Verilog, are quite low level and require considerable implementation expertise in order to achieve high-performance custom packet processing. Furthermore, sharing FPGAs among tenants is currently not supported and remains an open research problem [28].

A more promising technology is represented by software routers [20, 21, 26, 32]. They have reportedly been able to support rates of tens of Gbps and could be used either alone, as a replacement for traditional hardware-based switches, or in combination with them as in SideCar [33]. The NaaS model, however, introduces new challenges. First, rather than a handful of services as is typically the case with software routers [19], NaaS-devices must be able to support up to two orders of magnitude more tenant applications. Second, these applications are competing with each other for resources and are written by different parties. This is critical because, contrary to previous proposals on software routing, which assume *cooperating* and *trusted* services, a NaaS model exposes in-network processing to tenants. Hence, we must be able to execute malicious or poorly written code without impacting the performance of other tenants. Given the high scalability requirements, traditional virtualisation technologies cannot be used but more lightweight and scalable solutions must be developed, possibly coupled with network-level isolation mechanisms, e.g., [9]. Addressing these challenges is part of our current research agenda.

Network topology. We assume that network switches are interconnected in a standard fat-tree topology [6]. We chose the fat-tree because it provides full bisection bandwidth and has reportedly been widely adopted in data centres [5]. In addition, this topology requires only 1 Gbps switches with a limited port count (e.g., for a 27,648-server cluster, only 48-port switches are needed). This is important for our design because it means that the worst-case processing rate that the NaaS boxes must support is limited to tens (instead of hundreds) of Gbps (e.g., 48 Gbps for a 27,648-server cluster).

3.4 Programming Model

To implement INPEs we are investigating different programming models, ranging from rather low-level and highly expressive languages (e.g., Click [24] and those based on the Active Network model [12, 35]) to higher-level languages (e.g., based on declarative [25] or functional [27] models), which may be easier to use.

Our current approach is to expose a constrained programming model to tenants. By limiting the programming model, we simplify implementation (R2), while supporting efficient execution of INPEs on NaaS boxes (R3). In particular, our goals are (1) to improve *concurrency* when executing an INPE by dividing processing into distinct pipelined *phases*; (2) to increase *parallelism* by executing multiple instances of an INPE on different processor cores, while permitting out-of-order processing of data chunks; and (3) to reduce the memory footprint of an INPE so that the efficiency of L2/L3 cache of a processor core can be improved.

NaaS API. VMs and NaaS boxes exchange data streams, which are split into application-specific *data chunks*. For legacy applications, which are unaware of NaaS, data chunks may simply map to standard Ethernet frames. Each NaaS box and VM have a unique *NaaS ID* per tenant. An application running in a VM can use a `send_chunk` NaaS API call to send a data chunk of an arbitrary size either to a specific NaaS box or to another VM.

A tenant registers an INPE on all NaaS boxes for a specific application by calling the `reg_chunk_inpe` function, passing a filter expression and the callback to be asynchronously executed when a data chunk is received.[1]

All the traffic that does not match any of the above filters is forwarded directly by the switch as in the current, non-NaaS setup. Therefore, *no additional overhead is introduced for non-NaaS flows*.

NaaS topology. At deployment time, the cloud provider passes the network topology that interconnects NaaS boxes and VMs to INPEs. This allows INPEs to change their behaviour based on their location within the network.

Since the topology is only exposed to INPEs at run time, the implementation of an INPE cannot depend on a particular topology. This gives the cloud provider the flexibility to change the topology or expose a simplified network topology to tenants, with only a subset of all available NaaS boxes in the DC. We assume that the topology does not change at run time and therefore INPEs do not support updates to the topology once instantiated.

INPE API. To reduce contention and enable efficient pipelining, we propose to divide the processing logic of an INPE into a set of phases. When a new data chunk is received in a network stream bound to an INPE, the NaaS box triggers a callback to the `process_chunk` func-

[1] Data chunks can be batched to improve performance, similar to what is done with interrupt mitigation.

tion of the INPE instance. This function is implemented by application developers as a sequence of four phases: (1) In the *Read* phase, for each received chunk, the INPE can inspect the NaaS ID of the source that originally sent the data chunk and the ID of the previous NaaS box that processed the chunk. (2) INPEs can use a fixed amount of memory that is maintained by the NaaS box as a key/value store in the *State* phase. For example, to support aggregation across multiple chunks, the INPE can read and update an aggregation value. (3) In the *Transform* phase, the INPE can modify the received chunk data or create new chunks. (4) Finally, the INPE forwards the chunks on the network to the next NaaS ID using a call to `send_chunk` in the *Forward* phase.

The INPE processing state is explicitly managed by the NaaS box through a key/value store and can only be accessed during the *State* phase. This allows scheduling phases to reduce synchronisation costs for memory accesses to shared structures and to benefit from prefetching, allowing INPEs to maximally exploit L2/L3 caches.

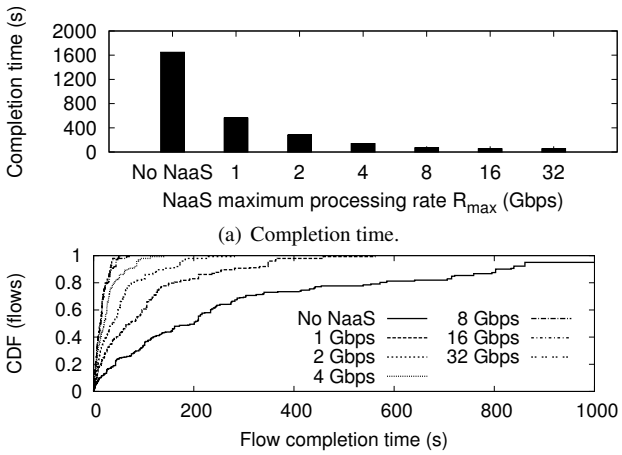
If the application tolerates out-of-order processing of data chunks, multiple data chunks can be processed in parallel by instantiating several INPEs instances on different processor cores.

4 Feasibility Analysis

We use simple flow-level simulations to evaluate the benefits and the performance requirements of the NaaS boxes. Albeit preliminary, our results suggest that a relatively modest processing capability of the NaaS boxes is already sufficient to significantly improve tenant applications performance. Further, we show that, since the overall traffic is reduced, the performance of non-NaaS users increases too, due to the higher bandwidth available in the cluster.

Simulation setup. We adopted a setup similar to the one used in [31]. We use a 8,192-server fat-tree topology [6], which uses only 32-port 1 Gbps switches. Each switch is equipped with a NaaS box. In this experiment, we want to quantify the trade-off between the performance increase and the processing resources needed by a NaaS box. Therefore, in our simulator, we assume that the rate at which the NaaS boxes can process packets is bounded by a parameter R_{max}. Depending on the actual implementation of the NaaS box and its bottleneck, R_{max} can be interpreted in different ways, e.g., the bandwidth of the bus connecting the switch to the processing unit like in ServerSwitch [26], the capacity of the links connecting the NaaS server to the switch like in SideCar [33] or, ultimately, the maximum processing rate achieved by the CPU. Here, we are not interested in comparing different implementations but just in understanding the minimum requirements that *any* NaaS implementation must fulfil.

We use a synthetic workload, consisting of approximately 1,000 flows. We assumed that 80% of these



(a) Completion time.

(b) CDF of flow completion time for different R_{max}.

Figure 2: Total completion time and individual flow completion time against different values of R_{max}.

belong to non-NaaS aware applications (e.g., standard TCP flows), and, therefore, their packets are routed only through switches, not through NaaS boxes. Based on the traces presented in [10], we modelled the flow inter-arrival times using a Poisson process with mean of 1 ms. The size of each flow is drawn by an exponential distribution with mean equal to 500 MB. The rest of the traffic is a mix of NaaS-aware applications, which include a multicast service [34], an aggregation service, inspired by the partition/aggregation pattern used by search providers [7], and a caching application, akin to the one used in [26].

We varied R_{max} from 1 Gbps to 32 Gbps. Since we used only 32-port 1 Gbps switches, 32 Gbps is the highest rate at which packets can be received by the NaaS box. We also run an additional configuration, *No NaaS*, which we use as baseline. In this configuration, no NaaS boxes are used and the multicast and aggregation traffic is routed using tree-based application-level overlays.

Simulation Results. The graph in Figure 2(a) shows the total (simulated) completion time of our experiment. This is measured from the start of the simulation until when the last flow completes. Across *all* values of R_{max} the use of NaaS significantly reduces the experiment completion time. NaaS-aware applications are able to reduce the number of packets sent over the links and, hence, improve the overall network utilisation. Interestingly, even if the processing throughput of the NaaS box is limited to 1 Gbps, the completion time is reduced by 65% (respectively 96.7% for $R_{max} = 16$ Gbps). The reason is twofold. First, only a small fraction of the total traffic is NaaS-aware, and, hence, the bandwidth required to process NaaS-aware traffic is lower. Second, in many cases, the benefits deriving from reducing the traffic compensate for the slower processing rate.

In Figure 2(b), we plot the cumulative distribution of individual completion time of *all* flows, including non-NaaS ones. The graph shows that NaaS is not only bene-

ficial for NaaS-aware tenants but also for the rest. Indeed, by reducing the number of packets sent over the links, not only NaaS-aware applications' performance improves but also the overall data centre throughput increases because more bandwidth is available to non-NaaS users. For instance, with $R_{max} = 1$ Gbps, the *median* flow completion time is reduced by 63.18% (respectively 93.07% for $R_{max} = 16$ Gbps).

5 Conclusions

Although we are aware of the admittedly simple model used in the simulations, the results in the previous section are encouraging. Recent work [20, 21, 32] showed that achieving rates equal to or higher than 10 Gbps on a commodity PC is possible. This makes us believe that implementing NaaS on existing hardware is feasible.

However, before we can expect to see more widespread adoption of NaaS, a range of research challenges have to be overcome. These include scalability, performance isolation and programmability.

Also, cloud providers will require new pricing models for NaaS offerings. It is yet unclear if simple cost model based on processor cycles used and network bits transferred is appropriate to meter the execution of INPEs. However, we observe that, as shown in the previous section, NaaS does not need a widespread adoption to be cost-effective. Indeed, providing a small-fraction of applications with more fine-grained control over network resources allows a more efficient usage of the network, which results in improved performance for every tenant.

Acknowledgements. This work was partly supported by grant EP/F035217 ("DISSP: Dependable Internet-Scale Stream Processing") from the UK Engineering and Physical Sciences Research Council (EPSRC), and by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-06-3-0001.

References

[1] Big Data in Real Time at LinkedIn. `http://goo.gl/6OzCN`.

[2] Google Tree Distribution of Requests . `http://goo.gl/RpB45`.

[3] Memcached. `http://memcached.org`.

[4] Twitter Storm. `http://goo.gl/Y1AcL`.

[5] James Hamilton's Blog, 2011. `http://bit.ly/e3LVu8`.

[6] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM* (2008).

[7] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).

[8] ANAND, A., SEKAR, V., AND AKELLA, A. SmartRE: An Architecture for Coordinated Network-Wide Redundancy Elimination. In *SIGCOMM* (2009).

[9] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards Predictable Datacenter Networks. In *SIGCOMM* (2011).

[10] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *IMC* (2010).

[11] BORTHAKUR, D., GRAY, J., SARMA, J. S., MUTHUKKARUPPAN, K., SPIEGELBERG, N., KUANG, H., RANGANATHAN, K., MOLKOV, D., MENON, A., RASH, S., SCHMIDT, R., AND AIYER, A. Apache Hadoop Goes Realtime at Facebook. In *SIGMOD* (2011).

[12] CALVERT, K. L., GRIFFIOEN, J., AND WEN, S. Lightweight Network Support for Scalable End-to-End Services. In *SIGCOMM* (2002).

[13] CARZANIGA, A., AND WOLF, A. L. Forwarding in a Content-Based Network. In *SIGCOMM* (2003).

[14] CHOWDHURY, M., ZAHARIA, M., MA, J., JORDAN, M. I., AND STOICA, I. Managing Data Transfers in Computer Clusters with Orchestra. In *SIGCOMM* (2011).

[15] COSTA, P., DONNELLY, A., ROWSTRON, A., AND O'SHEA, G. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *NSDI* (2012).

[16] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: A Stream Database For Network Applications. In *SIGMOD* (2003).

[17] CURTIS, A. R., MOGUL, J. C., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., AND BANERJEE, S. DevoFlow: Scaling Flow Management for High-Performance Networks. In *SIGCOMM* (2011).

[18] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI* (2004).

[19] DOBRESCU, M., ARGYRAKI, K., AND RATNASAMY, S. Toward Predictable Performance in Software Packet-Processing Platforms. In *NSDI* (2012).

[20] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *SOSP* (2009).

[21] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: A GPU-Accelerated Software Router. In *SIGCOMM* (2010).

[22] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys* (2007).

[23] JACOBSON, V., SMETTERS, D. K., THORNTON, J. D., PLASS, M. F., BRIGGS, N. H., AND BRAYNARD, R. L. Networking Named Content. In *CoNEXT* (2009).

[24] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *TOCS 18*, 3 (2000).

[25] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing Declarative Overlays. In *SOSP* (2005).

[26] LU, G., GUO, C., LI, Y., ZHOU, Z., YUAN, T., WU, H., XIONG, Y., GAO, R., AND ZHANG, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *NSDI* (2011).

[27] MADHAVAPEDDY, A., HO, A., DEEGAN, T., SCOTT, D., AND SOHAN, R. Melange: Towards a "functional" Internet. In *EuroSys* (2007).

[28] MADHAVAPEDDY, A., AND SINGH, S. Reconfigurable data processing for clouds. In *FCCM* (2011).

[29] MIGLIAVACCA, M., PAPAGIANNIS, I., EYERS, D. M., SHAND, B., BACON, J., AND PIETZUCH, P. DEFCon: High-Performance Event Processing with Information Security. In *USENIX ATC* (2010).

[30] NAOUS, J., GIBB, G., BOLOUKI, S., AND MCKEOWN, N. NetFPGA: Reusable Router Architecture for Experimental Research. In *PRESTO* (2008).

[31] RAICIU, C., BARRE, S., PLUNTKE, C., GREENHALGH, A., WISCHIK, D., AND HANDLEY, M. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM* (2011).

[32] RIZZO, L., CARBONE, M., AND CATALLI, G. Transparent Acceleration of Software Packet Forwarding Using Netmap. In *INFOCOM* (2012).

[33] SHIEH, A., KANDULA, S., AND SIRER, E. G. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *HotNets* (2010).

[34] VIGFUSSON, Y., ABU-LIBDEH, H., BALAKRISHNAN, M., BIRMAN, K., BURGESS, R., LI, H., CHOCKLER, G., AND TOCK, Y. Dr. Multicast: Rx for Datacenter Communication Scalability. In *EuroSys* (2010).

[35] WETHERALL, D. Active Network Vision and Reality: Lessons from a Capsule-Based System. In *SOSP* (1999).

[36] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *SOSP* (2009).