Data center resource management for in-network processing

Marco Micera

Politecnico di Torino, TU Darmstadt

Contents

Chapter 1

Introduction

This Master's thesis has been written at the Technische Universität Darmstadt, Germany, during a six-month Erasmus+ exchange and under the supervision of Prof. Patrick Eugster [†], M.Sc. Marcel Blöcher [†] and Prof. Fulvio Risso [‡].

1.1 Abstract

Nowadays there exist several In-Network Processing (INP) solutions that allow tenants to improve their application performance in terms of different metrics: DAIET [20] inventors claim to achieve an 86.9%-89.3% traffic reduction by performing data aggregation entirely in the network data plane. Other solutions like NetChain [13] and IncBricks [15] let programmable switches store data and process queries in order to cut end-to-end latency. It is now even possible to provide guarantees to applications with specific requirements: for instance, CloudMirror [14] enables applications to reserve a minimum bandwidth.

For the time being, it seems that there is still no valid resource allocation algorithm that takes into account the presence of a network having a data plane that supports (partially or completely) INP. This thesis has mainly two goals: (i) model and evaluate an Application Programming Interface (API) through which applications can ask for INP resources and (ii) argue the importance of a scheduler which is able to reject INP requests and propose their server-only equivalent when needed (e.g., high switch utilization).

1.2 Problem statement

Using INP to scale up data centers and improve their overall performance seems a promising idea: Daiet [20] inventors claim to achieve an 86.9%-89.3% traffic reduction, hence reducing servers' workload; NetChain [13] can process queries entirely in the network data plane, thus eliminating the query processing at servers and cutting the end-to-end latency to as little as half of an RTT.
Current data center Resource Managers (RMs) (such as Apache YARN [23], Google Omega [21]) are not completely network-unaware: for instance, some of them are capable of satisfying affinity rules. CloudMirror [14] even provides bandwidth guarantees to tenant applications. Still, current RMs do not consider INP resources. As a consequence, tenant applications cannot request INP services while asking for server resources.

[†] Distributed Systems Programming Group, Technische Universität Darmstadt, Germany
[‡] Computer Networks Group, Politecnico di Torino, Italy

1.2.1 Modeling INP resources

The goal of this Master's thesis consists in investigating how to model INP resources and how to integrate them in RMs.

In order to offer INP services to a tenant application, the latter should be able to ask for INP resources through an API. To do that, INP resources must be modeled not only to support currently existing INP solutions such as [20] [13] [15] [10], but also to support future ones. It might be convenient to derive a single model to describe both server and INP resources.

Classic tenant application requests can often be modeled as a key-value data structure. CloudMirror [14] requires a Tenant Application Graph (TAG) as an input, which is a directed graph where each vertex represents an application component and links' weights represent the minimum requested bandwidth. One possible model could be based on a TAG, describing network resources and INP services as vertexes or links. Tenant applications could either use the same model used within the data center or a simplified one, adding another level of abstraction.

Finally, a network-aware placement algorithm in the Resource Manager should then be able to allocate the requested resources accordingly.

Chapter 2

Background

The aim of this chapter is to introduce the fundamental technologies used in data centers, without which In-Network Processing (INP) would not even be possible. This chapter starts with a general description of how resources are managed in a data center § 2.1 (e.g., Virtual Machines) and ends with a brief introduction of network techniques and concepts which are strictly related to Resource Managers.

2.1 Resource management in data centers

In a data center, resources of any kind are virtualized in order to achieve higher flexibility, portability, and availability. Usually, both compute and storage resources are virtualized by means of Virtual Machines (VMs) and/or containers. Flexibility and portability are both automatically achieved thanks to this resource virtualization, resulting in software that can be deployed dynamically, run by multiple platforms and even live migrated; availability is usually simply achieved by not co-locating VMs and/or containers within a single power domain.

Nowadays, multiple Resource Managers (RMs) are using different approaches to solve different design issues. This section examines these existing RMs while trying to categorize them based on how they face different scheduling problems.

2.1.1 Glossary

This glossary is intended to list the set of definitions for all the terms used in this thesis. This is needed to avoid ambiguity, since this terminology has never been standardized.

2.1.1.1 Physical architecture

The elementary computational unit in a data center is called *node*. As mentioned in § 2.1, nodes are being virtualized by means of VMs and containers. Nodes run on *servers*, which are grouped into *racks*. Servers within a rack are usually connected through a so-called *Top of Rack (ToR)* switch. Server racks are then grouped into *cells* (or *pods*), that some times may be special-purpose. Usually, a cell is grouped with few small ones used for testing to form up a *cluster*. Most of the RMs manage resources in one cluster. One or more clusters form a *data center*, which together form a *site*.

2.1.1.2 Resources

In this thesis, resources are divided by (i) their level of abstraction and (ii) their type. The glossary reported at the end of this thesis follows a bottom-up approach, starting from those

terms belonging to the lowest level of abstraction.

2.1.2 Scheduling architectures

Monolithic. Probably the most simple scheduler architecture out there: a monolithic scheduler consists of a single instance scheduler applying the same scheduling algorithm for every incoming job (so there is no concurrency between resource requesters). A centralized scheduling logic can support more refined job placement. On the other hand, the absence of parallelism causes a higher latency with respect to other architectures. Although the single instance could distinguish among different job types hence treating those differently, its maintenance is not trivial, due to its single instance (and codebase) nature.

Two-level. This architecture requires computer clusters to be dynamically partitioned in sub-clusters, each having a dedicated scheduler. A centralized resource allocator determines which and how many resources should be made available to each scheduler: this is done by sending *offers* to schedulers (*pessimistic concurrency*). Obviously, conflicts can be avoided by not offering the same resource to multiple schedulers at the time. The same entity is in charge of dynamically divide clusters into sub-clusters: this is done to avoid resource fragmentation.

Shared-state. Schedulers are not mapped to sub-clusters like in the previous case. Multiple schedulers have access to the entire cluster and there is no centralized resource allocator assigning resources to schedulers. In this architecture, schedulers will try to acquire resources (*optimistic concurrency*), having not only the possibility of choosing between all the resources in the cluster, but also to ask for those which have been already acquired by another scheduler. In order to achieve this, a centralized data structure called *cell state* maintains the allocation information of all resources in the cluster, providing in fact a *shared-state* of it. Schedulers will try to acquire resources by atomically modifying this cell state, that will actually be modified only if the request does not cause any conflict. Each scheduler makes its own resource-allocation decision on a private copy of the cell state, which is updated every time the scheduler tries to acquire some resource, no matter what the outcome of the attempt is.

2.1.3 Taxonomy

This section tries to underline the different scheduling design issues that RMs must face by building a simple short taxonomy, following the guidelines provided by Google in their Omega [21] paper.

Scheduling work partitioning. The workload can be distributed across schedulers in three different ways: (i) workload-type unaware load balancing, (ii) workload partitioning to specialized clusters and, (iii) a combination of the two.

Interference. Schedulers can concurrently ask for the same resources. In the pessimistic approach, different resources are offered to different schedulers, making it impossible for them to compete for the same resource: this of course represents a lack of parallelism, since resource offers are made by a logically centralized entity. Instead, the optimistic approach lets every scheduler claim the desired resources and only conflicting requests are denied, which of course introduces an overhead.

Choice of resources. Schedulers can pick amongst (i) all cluster resources or (ii) a subset of those. When resources are divided into disjoint sets there will be no concurrency by definition. Making all resources available to all scheduler will make it easier for schedulers to place jobs with particularly stringent needs, and it is also useful when the scheduling decision must be taken based on overall state (e.g., the amount of free resources in the cluster).

Preemption. Schedulers can be either allowed to preempt other schedulers' job assignments or not. Allowing preemption brings a greater flexibility at the cost of interrupting an already-running job. Since all state-of-the-art RMs support preemption, this design choice will be taken for granted and hence omitted when comparing different RMs in § 3.1.7.

Allocation granularity. Considering that jobs contain multiple tasks that can be scheduled on different resources, schedulers can either (i) incrementally schedule tasks as soon as new resources become free or (ii) schedule a job only when all tasks can be scheduled on the spot. Not all job types can exploit the incremental resource acquisition. This technique can also bring the system to a deadlock in case there is no back-off mechanism that releases resources once a job cannot acquire all resources in a reasonable amount of time.

2.2 Software Defined Networking (SDN)

Perhaps the most fundamental technology without which this thesis project would not be possible. Software Defined Networking (SDN) is a paradigm whose main pillar is the separation of the control and forwarding (data) plane. The brain of a Software Defined Network resides in its logically centralized SDN controller, that ... via a SDN Southbound Interface (SBI).

2.3 Network Function Virtualization (NFV)

These virtualization techniques previously mentioned in § 2.1 (e.g., VMs and containers) can also be used to virtualize network functions (e.g., firewalls, network monitors, etc.) in order to obtain all the advantages that come along with virtualization in general, such as better resources utilization, fault-tolerance, higher flexibility, and so on.

When combining Service Function Chaining (SFC) with Network Function Virtualization (NFV), network functions ...

2.4 In-Network Processing (INP)

Within this project, data center INP refers to the technique of delegating some parts of computation to programmable switches, hence reducing servers' workload.

Common operations performed by these devices are data aggregation and key-value storage management.

Chapter 3

Analysis

Chapter 2 introduced how resources can be managed in a data center (§ 2.1) and different network techniques such as In-Network Processing (INP) (§ 2.4) and Network Function Virtualization (NFV) (§ 2.3). This chapter's aim is to dig into the details of these systems and to extract common patterns between similar INP solutions in order to be able to derive a model capable of fully describing INP resources.

3.1 Resource Management Frameworks (RMFs) analysis

After having described the basics behind resource management in § 2.1, it is NOW time to dig into the details of existing RMs and categorize them following the taxonomy introduced in § 2.1.3.

3.1.1 Borg [24]

Borg [24] is the first container-management system developed by Google. Jobs are divided in two groups depending on their workload type (long-running and batch) and they were initially scheduled by a logically centralized controller called *Borgmaster*. After Google developed the shared-state Omega [21] scheduler (analyzed in § 3.1.2), Borg [24] adopted the shared-state architecture as well, hence being no more a monolithic solution.

3.1.1.1 Scheduling

Each cell has its own logically centralized controller called *Borgmaster*, which is essentially split in two parts: (i) the scheduling part, consisting in one or more schedulers differentiated by the workload type they handle and (ii) a management unit in charge of handling client Remote Procedure Calls (RPCs) and communicating with all other Borg [24] agents. Schedulers mainly work on tasks rather than jobs. Tasks are scanned in a round-robin fashion and their priority is also taken into account. Schedulers must first find all available machines for the task to be scheduled (also considering those currently acquired by a task with lower priority) and then find the best machine amongst them all. This second part is done by taking into consideration not only user-specified preferences, but also data center global goals like minimizing of preempted tasks and allocating tasks on machines which already have the needed packages installed in order to reduce the installation time, which usually takes about 80% of the total start-up latency.

3.1.1.2 Details

Jobs. Jobs in Borg [24] are split into multiple tasks which run within a single cell: tasks belonging to one job cannot be spread amongst multiple cells in the same cluster. In fact, Borg [24] only operates on cells. Each job has some properties such as a name, an owner, and most importantly, constraints about *machines* (e.g., processor architecture, etc.) which will determine where its tasks will be scheduled. Tasks instead have *resource* requirements (e.g., CPU cores, RAM, etc.) expressed in terms of *quotas* (an array of resource quantities). Task properties can be modified at run-time by the job owner: this is done by pushing a new job configuration file to Borg [24] and ordering the scheduler to update the involved tasks via a non-atomic transaction. Most of the workload in Borg [24] run in containers rather than VMs to avoid the virtualization overhead.

Fairness. Jobs have a priority expressed as an integer. Workload types are even more differentiated with the definition of non-overlapping priority bands, one for a different kind of workload. Borg [24] schedulers are preemptive, and in order to contain the negative effects of preemption cascades the system does not allow internal preemption for certain priority bands.

Resource reclamation. Tasks do not fully use their resources for their entire lifespan. This is why the *Borgmaster* estimates every few seconds what is the actual amount of resources that each task needs and reclaims the unused resources to make them available for other tasks. This is done by periodically contacting Borg [24] agents running on each cluster machine, requesting for fine-grained resource consumption information. The initial estimated amount of resources actually needed by a task corresponds with its maximum limit, and it then slows according to the actual consumption. If the actual resource usage exceeds the estimation, than the latter is rapidly increased. This technique justifies why Borg [24] inventors have noticed that dedicating clusters for different workload types is inconvenient. They showed how segregating long-running and batch jobs in different specialized clusters requires 20% to 30% more machines than having clusters who run both type of jobs using resource reclamation.

3.1.2 Omega [21]

Omega [21] is a parallel, lock-free and optimistic cluster scheduler by Google. As said in [4], it was born after Borg [24] with the aim of improving its software engineering. There is no central resource allocator: all of the resource-allocation decisions take place in the schedulers. Multiple schedulers were first introduced in Omega [21] and then in Borg [24], making the latter scheduler no more monolithic.

3.1.2.1 Scheduling

This solution makes use of a data structure called *cell state* containing information about all the resource allocation in the cluster. Each cell has a shared copy of this data structure, and each scheduler is given a private, local, frequently-updated copy of cell state that it uses for making scheduling decisions. According to the optimistic concurrency technique, once a scheduler makes a placement decision, it updates the shared copy of cell state with a transaction. Whether or not the transaction succeeds, a scheduler re-syncs its local copy of cell state afterwards and, if necessary, re-runs its scheduling algorithm and tries again. Omega [21] supports specialized schedulers: authors have showed the advantages of a MapReduce [7] specialized scheduler in [21].

3.1.2.2 Conclusions

Since schedulers do not have access to all cluster resources, Mesos [11] cannot support preemption across different sub-clusters and it cannot apply policies that make use of the complete cluster state.

3.1.3 Apache™ Hadoop© YARN [23]

Apache™ Hadoop© YARN [23] (for the sake of brevity: Apache™ YARN [23]) is the Resource Manager of Apache™ Hadoop©, a framework for distributed processing across clusters.

Apache™ Hadoop© was initially an open source implementation of MapReduce [7], but then the programming model has been separated from the resource management function, resulting in an application-independent RM known as Apache™ YARN [23].

3.1.3.1 Entities

For each tenant application there is an *Application Master* whose task is to (i) manage the application life cycle and (ii) negotiate the resources that the application needs with the central RM, making Apache™ YARN [23] a monolithic scheduler with no interference between tenant applications. Each node then has a *Node Manager* thanks to which the RM can allocate tasks on it. The *Node Manager* must also periodically monitor resource availability and report failures.

3.1.3.2 Scheduling

Application Masters issue resource request to the RM, containing containers properties and locality preferences. Upon receiving a resource request, the centralized scheduler generates containers using available resources periodically advertised by the nodes themselves. The outcome of this procedure is reported to the Application Master corresponding to the tenant application who initiated the request. Application Masters are also informed upon inserting new nodes into the system.

3.1.3.3 Details

Preemption. The RM can also ask to Application Masters to revoke some resources in case of a shortage. The application will then have a few choices: for instance it can yield containers that are less important or checkpoint its current status. If an application does not collaborate with the RM upon receiving a *revoking request*, the RM will forcibly terminate those targeted containers.

Failures. The RM represents a single point of failure for the system and its restart causes the termination of all containers in the cluster, including all their Application Masters. Node failures are detected by the RM using timeouts (nodes have to periodically contact the RM). The RM will then inform all Application Masters who are responsible for responsible for the application life cycle.

3.1.3.4 Conclusions

Undoubtedly, Apache™ YARN [23] dedicates less attention to scalability due to its de facto monolithic scheduler: there are multiple Application Masters who just take care of the application life cycle and do not perform scheduling, which is done instead by a single RM.

However, Apache™ YARN [23] authors state that the centralized RM can assure fairness, capacity and locality thanks to the central and global view that it has on the system. They justify this by pointing out that Apache™ Hadoop© is an open platform which lets different independent sources share the same cluster, unlike other "*closed-world*" schedulers like Google Omega [21].

3.1.4 Mesos [11]

Mesos [11] is a two-level cluster scheduler based on *resource offers*. It has multiple schedulers since Mesos [11] has been conceived to share clusters between different cluster computing frameworks since the beginning of its development. By contrast, Apache™ YARN [23] was initially embedded in the first version of MapReduce [7] and subsequently became independent out of the necessity to scale Apache™ Hadoop©.

3.1.4.1 Entities

This scheduler has a logically centralized resource *allocator* in charge of offering resources to different schedulers. It is called *Mesos master* and it is replicated for fault tolerance. A scheduler with its *executor* (worker) node are together called *framework*. Nodes running on cluster nodes are called *Mesos slaves*.

3.1.4.2 Scheduling

Initially, every cluster node reports to the master node its own available resources. Based on this data, the master node can then offer resources to application frameworks based on a particular policy. The master node does not offer the same subset of resources to each scheduler. Obviously resource conflicts can be avoided by not offering the same resource to multiple schedulers at the time. Upon receiving resources offers, application frameworks can either reject the offer (in case it does not satisfy all framework's constraints) or tell the master which tasks need to be run on the dedicated resources. Mesos [11] already knows that certain types of frameworks always reject certain resource offers characterized by some factors, so frameworks can specify *filters* in order for the master to automatically avoid proposing certain kind of resources.

The resource allocation logic can be customized, and Mesos [11] includes an allocation module based on priority and one based one fairness. Tasks can be preempted, however frameworks can be offered *guaranteed* resources on which tasks cannot be preempted.

3.1.5 Guarantee provisioning: CloudMirror [14]

CloudMirror [14] allows client applications to specify bandwidth and high availability guarantees.

3.1.5.1 Motivation

Prior models are not suitable to represent interactive non-batch applications with very stringent bandwidth requirements. Both the hose and the Virtual Oversubscribed Cluster (VOC) model are inefficient as they over-allocate bandwidth. The main reason of why this happens is that both models *aggregate* bandwidth requirements between different application components into a single hose: as a consequence, the VM scheduler does not get to know the actual bandwidth needed between application components. At the opposite extreme there is the pipe model which, besides not exploiting statistical multiplexing, is not scalable since it requires a list of all bandwidth guarantees between pairs of VMs. This led CloudMirror [14] inventors to come up with a new model.

3.1.5.2 Tenant Application Graph

The TAG is a directed graph where each vertex represents an application component and links' weights represent the minimum requested bandwidth. Each vertex can have an optional *size*, denoting the number of VMs belonging to the component.

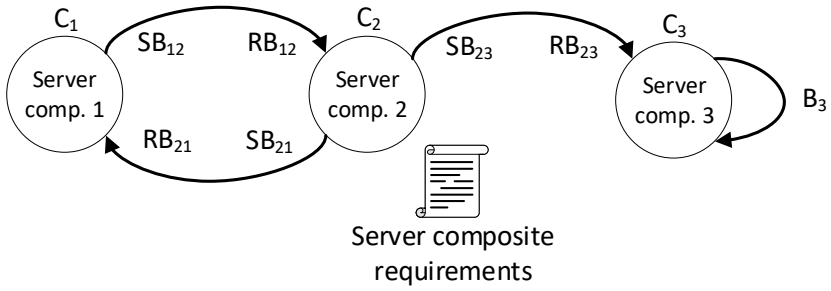

Figure 3.1: A TAG example

There are two types of edges: (i) self-loop edges, that are equivalent a hose model and (ii) standard vertex-to-vertex edges. A standard edge from $Tier1$ to $Tier2$ is labeled with an ordered pair of numbers $< SB_1 2, RB_1 2 >$, indicating respectively the guaranteed bandwidth with which VMs in $Tier1$ can send traffic to VMs in $Tier2$ and the guaranteed bandwidth with which VMs in $Tier2$ can receive traffic from VMs in $Tier1$.

3.1.5.3 Model advantages

The edge label format $< S, R >$ allows the model to exploit statistical multiplexing, since S can represent the peak of the sum of VM-to-VM demands instead of the (typically larger) sum of peak demands needed by the pipe model. ***TO BE CONTINUED*** ...

3.1.6 Firmament [9]

Firmament [9] is a centralized cluster scheduler capable of supporting multi-dimensional resource requirements (§ 3.1.6.2). The scheduler finds an embedding solution by solving a min-cost max-flow optimization problem over a graph called *flow network* (§ 3.1.6.1).

3.1.6.1 Flow network

The flow network is a directed weighted graph that contains both physical and logical resources. Similarly to any other graph-based scheduler (e.g., [17]), a physical resource is connected to a logical resource anytime the former can be allocated on the latter. In order to reduce the overall amount of arcs, the flow network makes use of *equivalence class aggregates* which group nodes with similar characteristics (e.g., tasks within a job, physical machines with the same micro-architectural topology, etc.). An aggregate receives arcs from all entities belonging to it. Physical aggregates can also receive arcs from those logical nodes that can be allocated on any of the physical entity represented by the aggregate.

The physical network is faithfully recreated in the graph thanks to aggregates: for instance, different machines belonging to the same rack are connected to the same *rack aggregate*, all racks are connected to the cluster aggregate and so on. The description does not either stop at the machine level, since also sockets and cores are represented in the graph. Again, inclusion relationships are expressed in terms of arcs. Special *unscheduled aggregates* (one for each job) receive an arc from every task.

The embedding solution is found by running an instance of a min-cost max-flow optimization algorithm over the network flow. If a task cannot be allocated on any of the physical machines, its flow will be directed to its corresponding unscheduled aggregate. Arcs' costs are determined by a given scheduling policy (or "cost model"): at the time of writing, Firmament [9] supports 9 different scheduling policies. Intuitively, arcs connecting tasks to unscheduled aggregates will have an higher cost with respect to the ones ending in a physical entity.

3.1.6.2 Multi-dimensional resource fitting

Firmament's [9] creator Malte Schwarzkopf introduced the coordinated co-location (CoCo) cost model in his PhD dissertation - Schwarzkopf, M. (2018). *Operating system support for warehouse-scale computing* (Doctoral thesis). https://doi.org/10.17863/CAM.26443.

This scheduling policy has an interesting feature: *admission control*. Basically a task can be allocated on a physical machine only when it satisfies all task's resource requirements (strict resource fit). If a task cannot be allocated on any of the physical machines, it cannot be even connected to the cluster aggregate. Given a logical resource in the network flow, CoCo is able to efficiently determine all subtrees in the physical topology in which it can be allocated. This is done by storing in each physical aggregate the minimum and maximum amount of available resources across its children.

3.1.7 RMFs comparison

The table below contains a quick comparison amongst RMs previously analyzed. In order to categorize those, the taxonomy introduced in § 2.1.3 has been used.

Resource Managers: quick summary

Borg [24]	Scheduling architecture	Shared-state
	Scheduling work partitioning	Specialized clusters
	Interference	Optimistic approach
	Choice of resources	All cluster resources
	Allocation granularity	???
Omega [21]	Scheduling architecture	Shared-state
	Scheduling work partitioning	Specialized clusters
	Interference	Optimistic approach
	Choice of resources	All cluster resources
	Allocation granularity	Per-scheduler policy
Apache™ YARN [23]	Scheduling architecture	Monolithic
	Scheduling work partitioning	—
	Interference	No interference
	Choice of resources	All cluster resources
	Allocation granularity	???
Mesos [11]	Scheduling architecture	Two-level
	Scheduling work partitioning	???
	Interference	Pessimistic approach
	Choice of resources	Subset of resources
	Allocation granularity	All-or-nothing

Table 3.1: Resource Managers comparison table using the taxonomy introduced in § 2.1.3

3.2 INP solutions

State-of-the-art INP solutions will be discussed in this section with the aim of deriving a model capable of fully describing INP resources. To that end, it is necessary to dig into the details and to recognize common patterns between them.

3.2.1 In-network aggregation: Daiet [20]

Daiet [20] is a system that performs in-network data aggregation for partition/aggregate data center applications (big data analysis such as MapReduce [7], machine learning, graph processing and stream processing). Instead of letting worker servers entirely perform computation on the data and then communicate with each other to update shared state or finalize the computation, the system let network devices perform data aggregation in order to achieve traffic reduction, thus reducing the processing load at the destination.

The inventors have proven that in-network data aggregation can reduce the network traffic significantly for machine learning algorithms (e.g., TensorFlow [1]) and for graph analytics algorithms (e.g., GPS [19]), hence justifying the usefulness of this system. The system has been designed for P4 [3] and programmable ASICs, and it can be used on any other SDN platform.

3.2.1.1 Details

Controller. When executing a MapReduce program, the job allocator informs the network controller of the job allocation to the workers. Then, the network controller pushes a set of rules to network devices in order to (i) establish one aggregation tree for each reducer and (ii) perform per-tree aggregation. An aggregation tree is a spanning tree from all the mappers to the reducer.

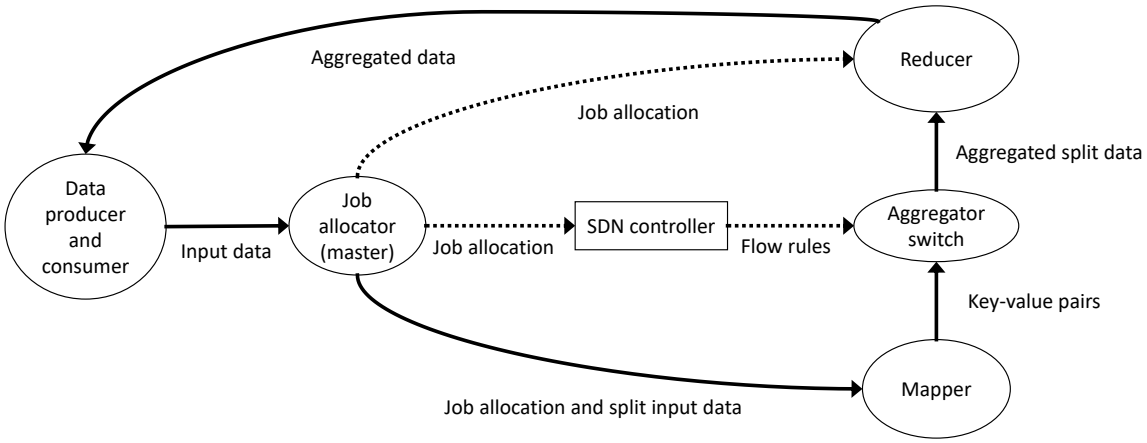

Figure 3.2: Daiet's [20] basic topology and entities

Packets. Since every reducer has its own aggregation tree associated with it, network devices should know how to correctly forward traffic according to the corresponding tree: to achieve this, a special *tree ID* (that could coincide with the *reducer ID*) packet field allows network devices to distinguish different packets belonging to different aggregation trees.

Obviously, they must also know the output port towards the next network device in the tree and the aggregation function to be performed on the data.

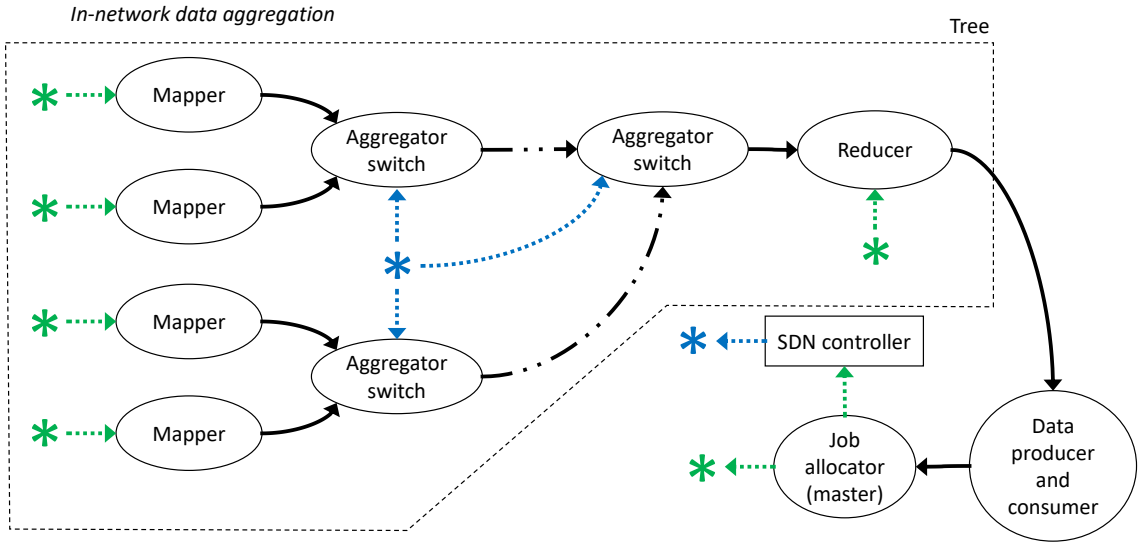

Figure 3.3: Daiet's [20] extended topology

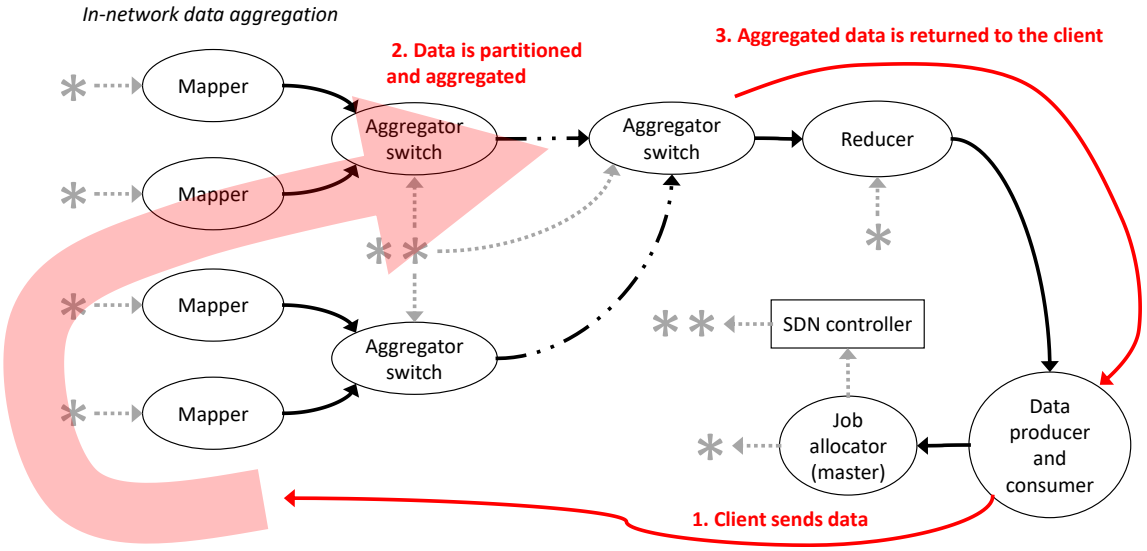

Figure 3.4: Daiet's [20] logical communication pattern

Packets are sent via UDP (therefore communication is not reliable) with a small preamble that specifies (i) the number of key-value pairs contained in the packet and (ii) the *tree ID* whose packet belongs to. They payload is not serialized to achieve a faster computation by network devices.

3.2.1.2 Algorithm

To store the key-value map, network devices use two hash tables for each tree: one for the keys and one for the values. Upon a collision, the algorithm checks whether the key is matching or just the hash is. In the former case, data aggregation is performed. In the latter case, the conflicting pair will end up in a *spillover bucket* that will be flushed to the next node as soon as it becomes full: this is done since this data is more likely to be aggregated by the next network device if it has spare memory.

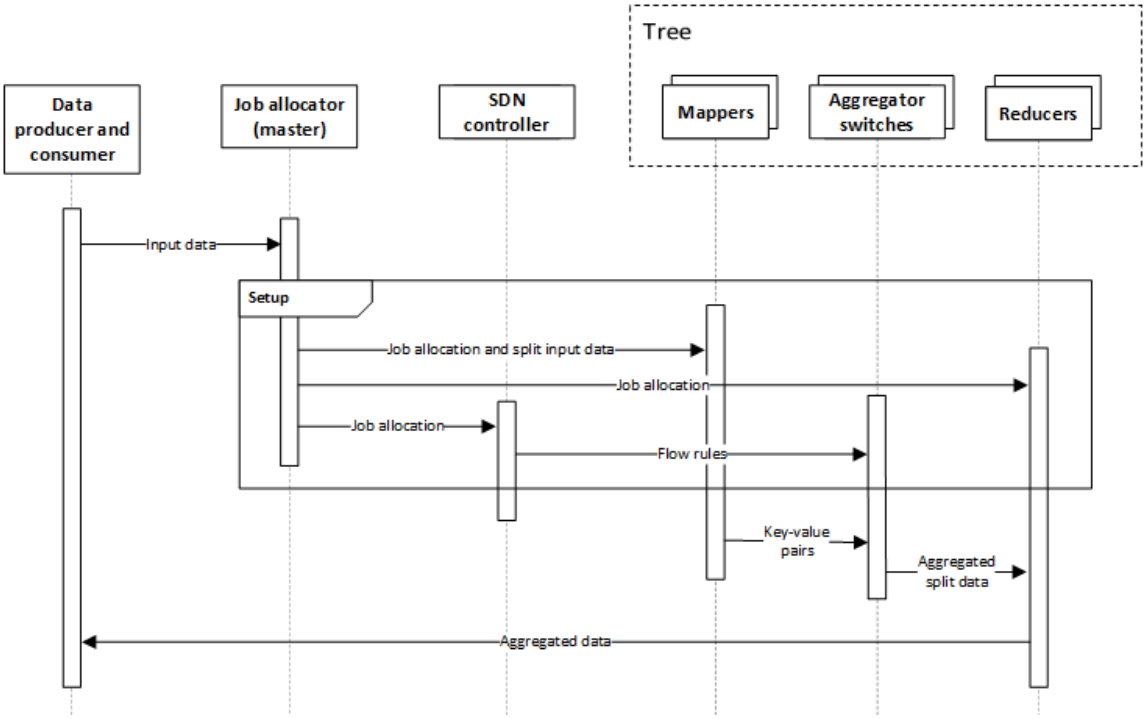

Figure 3.5: Messages exchanged during a Daiet [20] instance execution

A network device will also flush its data as soon as all its children (according to the aggregation tree) have sent their data: this is made possible by forcing network devices to send a special *END* packet after transmitting all key-value pairs to their successor.

Used indexes are stored in a *index stack* to avoid scanning the whole hash table when flushing.

3.2.1.3 Implementation

The network data plane has been programmed using P4 [3], which brings two main drawbacks: (i) a match-action table cannot be applied more than once for the same packet, forcing the programmer to perform loop unrolling in case of multiple headers in the same packet that need to be modified by the same rule table and (ii) keys must have a fixed size, causing a big waste of memory in applications where keys have variable-lengths (e.g., strings). The second drawback will also cause arrays to have fewer but bigger cells, thus increasing the probability of collisions. If these collisions involve pairs having different keys (section 3.2.1.2), data will not be aggregated, causing traffic to increase.

3.2.1.4 Minimum system requirements

For each aggregation tree (i.e., for each reducer), network devices must form a tree whose root is connected to the reducer and whose leaves are connected to mappers. Each mapper has to be connected to exactly one network device of the lowest level. Network devices must (i) store two arrays (one for the keys and one for the values) and (ii) be able to hash keys. The solution requires that the system has a centralized SDN controller connected to all switches. The SDN controller must push flow rules to all switches belonging to at least one tree.

3.2.1.5 Conclusions

Besides all the drawbacks brought by P4 [3] listed in section 3.2.1.3, inventors claim to achieve a 86.9%-89.3% traffic reduction, causing the execution time at the reducer to drop by 83.6% on average.

3.2.2 Coordination services: NetChain [13]

NetChain [13] is an in-network solution for coordination services, such as distributed locking, barriers, etc. All these services are realized on top of a strongly-consistent and fault-tolerant key-value store, which is entirely implemented in the network data plane. The network device in charge of storing the distributed store is a programmable switch: this brings an obvious limitation in terms of storage size, that makes NetChain [13] an acceptable solution only when a small amount of critical data must be stored in the network data plane, e.g., coordination services.

NetChain [13] can process queries entirely in the network data plane, causing the end-to-end latency to drop from multiple RTTs to as little as half of one RTT since servers are not involved in query processing anymore.

3.2.2.1 Details

Packets. Custom UDP packets are used for queries, containing fields like *operation*, *key* and *value*. Read and write queries only involve the network data plane, while insert and delete queries involve the network controller to set up entries in switch tables and to perform garbage collection, respectively.

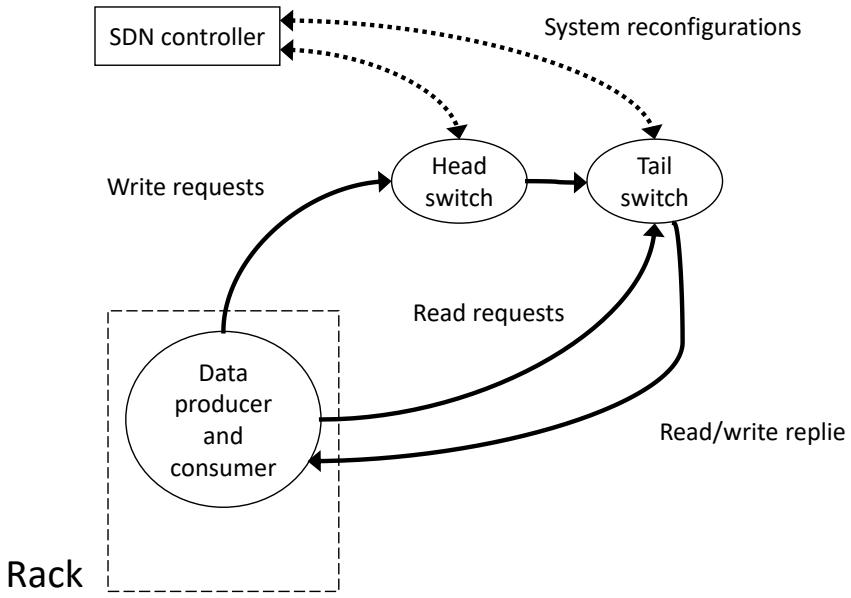

Figure 3.6: NetChain's [13] basic topology and entities

This is acceptable since coordination services usually perform read and write queries on already-existing objects, e.g., locks. Each switch has its own IP address, and packet headers contain the list of addresses of switches to be traversed, allowing those to properly forward packets to the their successors (from head to tail for write queries and the opposite for read queries). This list of IP addresses is inserted by the client (a NetChain [13] agent).

Consistency. A variant of Chain Replication [22] is used in the data plane to handle read and write queries and to ensure strong consistency, while switches reconfiguration is handled by the network control plane. The main difference with the standard Chain Replication [22] protocol is that objects are stored on programmable switches instead of servers. Switches are logically connected together in order to form an oriented chain.

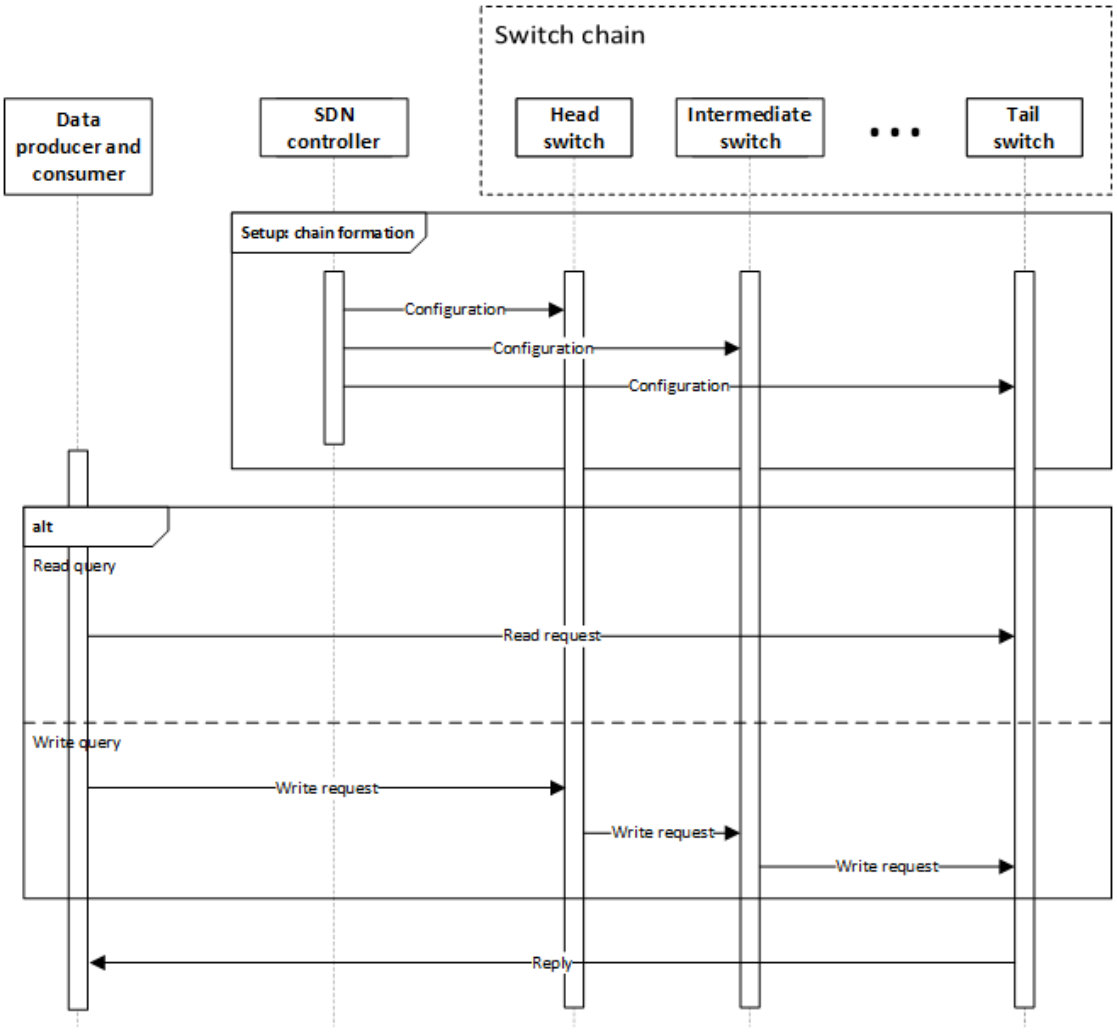

Figure 3.7: Messages exchanged during a NetChain [13] instance execution

Read queries are processed by the *tail* switch while write queries are sent to the *head* switch, which will forward the updated state to the rest of the chain.

The key-value store is partitioned amongst *virtual nodes* using consistent hashing, mapping keys to a hash ring. Each ring segment is stored by $f + 1$ virtual nodes allocated on different physical switches, hence tolerating faults involving up to f switches.

3.2.2.2 Implementation

The network data plane has been programmed in P4 [3] while the controller has been coded in Python, it runs on a server and communicates with switches through the standard Python RPC library. Switches agents are Python processes who run in the switch OS. Some P4 [3] drawbacks were already discussed in section 3.2.1.3.

The out-of-order UDP delivery problem is resolved by adding sequence numbers to write queries, hence serializing those operations, while the loss of packets is coped by *client-side retries* based on timeouts.

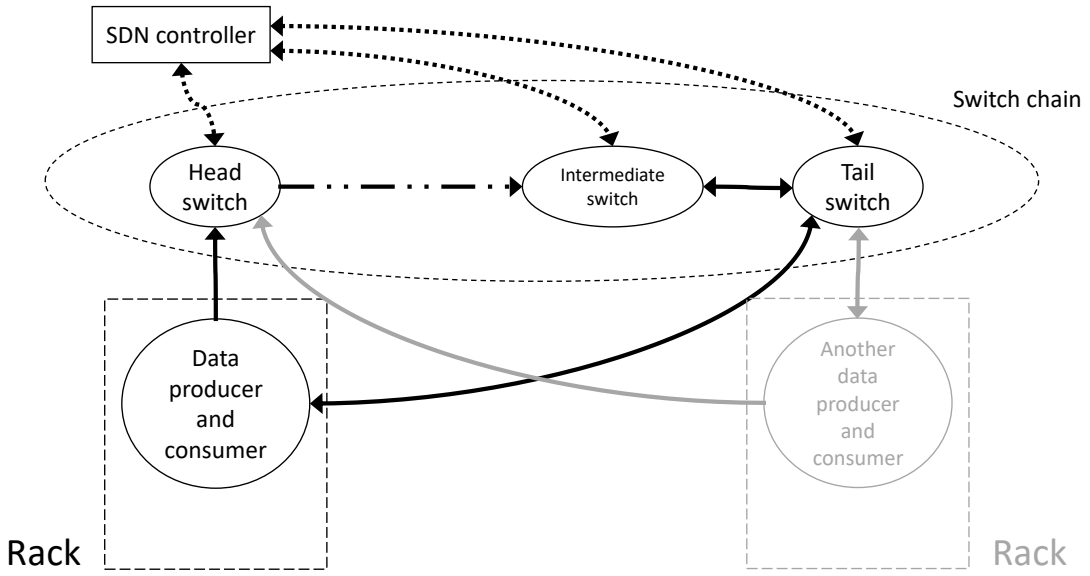

Figure 3.8: NetChain's [13] extended topology

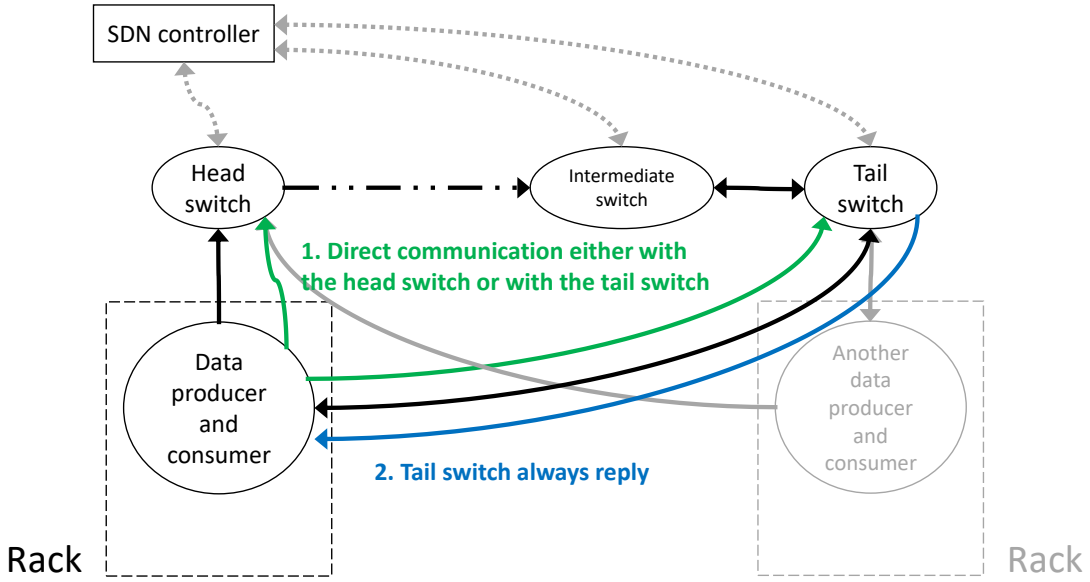

Figure 3.9: NetChain's [13] logical communication pattern

3.2.2.3 Minimum system requirements

Network devices must form a chain of length $f + 1$ in order to tolerate f failures. The client must include the list of IP addresses of all the $f + 1$ switches to be traversed in each query packet header (from head to tail for write queries and the opposite for read queries; storing the entire backward list for read queries is only necessary in case of tail failures).

Network devices must dedicate some local storage to NetChain [13]: more specifically, they need to store a (i) register array to store values and a (ii) match-action table to store the keys' location in the register array and the corresponding action to be performed. The solution requires that the system has a centralized SDN controller connected to all switches. The SDN controller must handle switches reconfigurations.

3.2.2.4 Conclusions

NetChain [13] inventors state that the on-chip memory of programmable switches in enough for coordination services. Assuming a 10 MB partition allocated on each NetChain [13] switch, a data center with 100 switches can provide a $(10\,\text{MB} \cdot 100)/3 = 333\,\text{MB}$ storage with a replication factor of three: that would be enough for the average number of files (22k, from 0 to 1 byte) managed by a typical Chubby [5] lock service instance, as cited by Google in their corresponding paper. Likewise, inventors claim that switches total memory is enough for a distributed locking system: assuming 30 B locks, the previously-mentioned example would be capable of storing $333\,\text{MB}/30\,\text{B} = 10M$ concurrent locks.

3.2.3 In-network caching fabric: IncBricks [15]

IncBricks [15] is a hardware-software co-designed system for in-network caching: it makes use of network accelerators attached to programmable switches whenever complicated operations should be performed on payloads. Supporting multiple gigabytes of memory, network accelerators overcome the limited storage problem typical of programmable switches, which usually have a memory of tens of megabytes.

3.2.3.1 Details

Hardware. IncBricks [15] is composed by two components: (i) IncBox, an hardware unit consisting of a network accelerator and a programmable switch, and (ii) IncCache, a software system for coherent key-value storage. Packets arriving to an IncBox device are first managed by the switch, which forwards the packet to the network accelerator only if it is labeled as an in-network cache one.

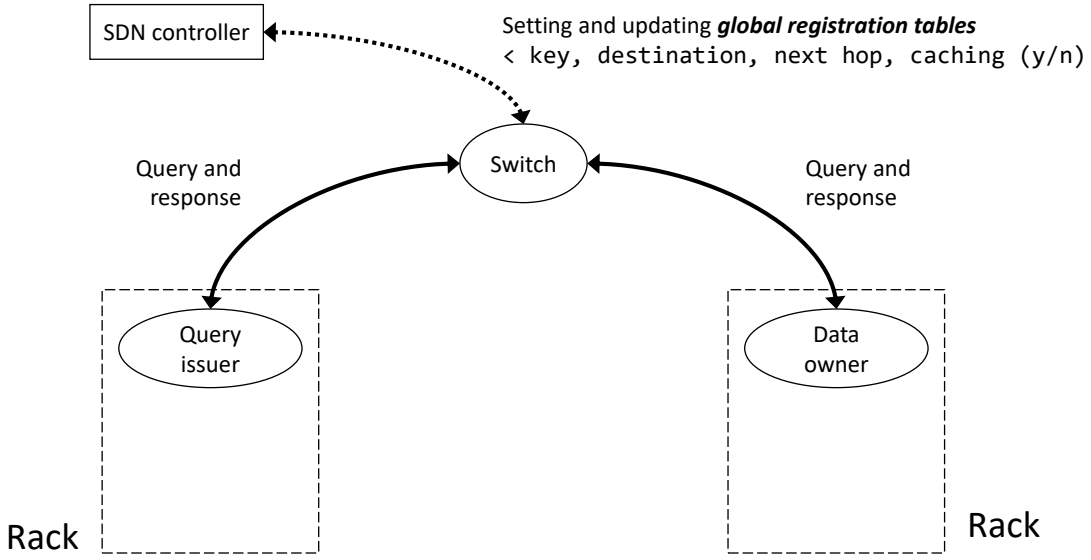

Figure 3.10: IncBricks' [15] basic topology and entities

If there is a match, the programmable switch will check whether the packet has been already cached by the network accelerator or not, and will forward the packet to the right network accelerator attached to it in the former case.

Logic. The system has been designed having a multi-rooted tree topology in mind. For each key the centralized SDN controller comes up with a set of *designated* IncBox units allowed to cache that key. Any other IncBox unit placed between these designated units won't cache data with that specific key.

Then, for a given key and a given destination node, the SDN controller establishes a unique path of designated IncBox units. Every IncBox unit in the system will get to know (i) the set of immediate designated successors (according to the tree topology) for every key it is responsible of and (ii) the unique successor used for a given destination and a given key. This data is stored in the so-called *global registration table*. Storing the former information can be useful in case of failures since it is possible to build alternative paths immediately, making the whole system more reliable. As soon as a failure is detected, the SDN controller updates all the involved tables.

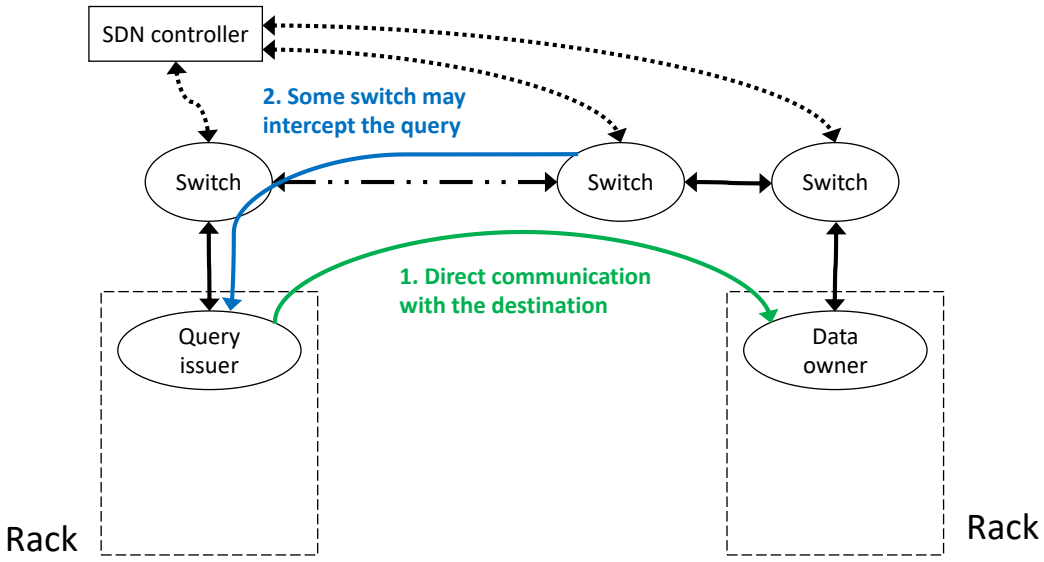

Figure 3.11: IncBricks' [15] logical communication pattern

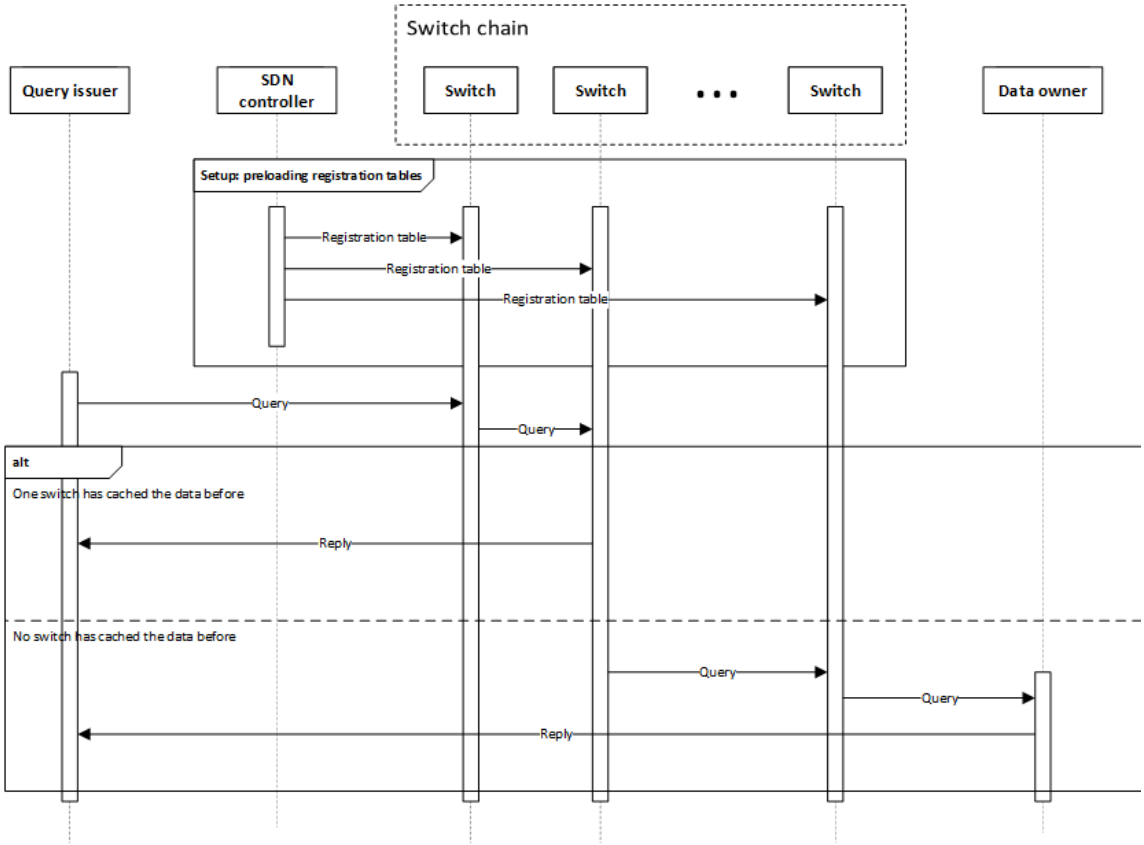

Figure 3.12: Messages exchanged during an IncBricks [15] instance execution

3.2.3.2 Implementation

The storage has been implemented using a bucket spilling hash table plus a hash index table ***TO BE CONTINUED*** ...

3.2.3.3 Minimum system requirements

Communicating nodes (VMs) represent the leaves of the tree. Each path must include exactly one root switch.

All things considered, it seems reasonable to state that the actual required topology is a chain starting from a leaf, passing through a root node and ending on another leaf.

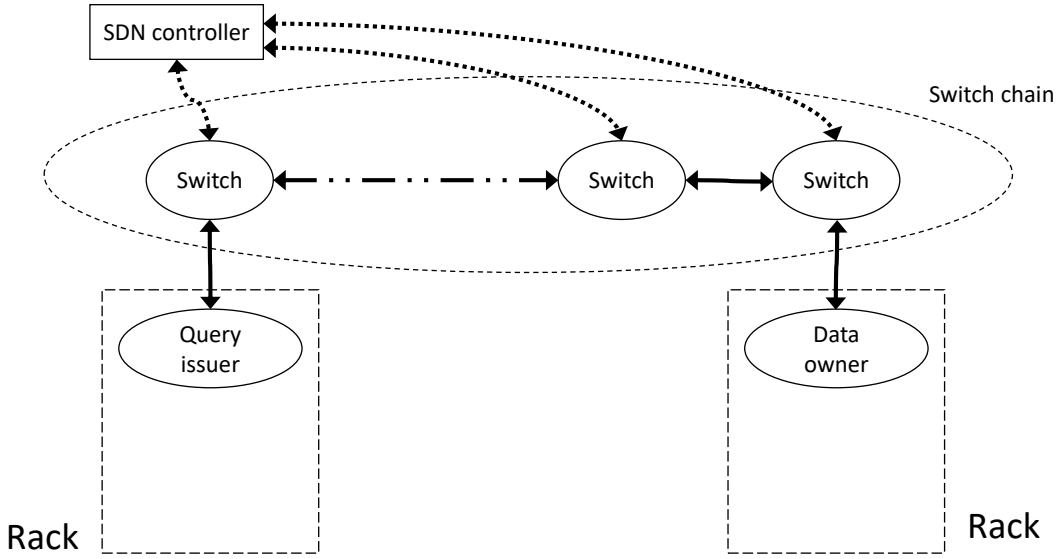

Figure 3.13: IncBricks' [15] extended topology

IncBox units must dedicate some local storage to realize the caching system. The solution requires that the system has a centralized SDN controller connected to all switches. The SDN controller must set configure network devices in order for them to forward IncBricks [15] packets accordingly.

3.2.3.4 Conclusions

3.2.4 Aggregation protocol: SHArP [10]

SHArP [10] stands for *Scalable Hierarchical Aggregation Protocol*, and it defines a protocol for reduction operations. This solutions aims to accelerate High Performance Computing (HPC) applications by offloading some operations to the network. SHArP [10] is targeted to support the two most used APIs in the HPC area today: MPI [25] and OpenSHMEM [6]. The kind of operations that can be offloaded to the network are: (i) MPI [25] barrier, reduce and allreduce, (ii) logic operands like sum, min, max, or, xor, and, (iii) integer and floating-point operations.

3.2.4.1 Details

Entities. Aggregation Nodes (ANs) are logical entities performing reduction operations. Such a node can either execute on a network device or on a server, and it is implemented as a daemon, namely `sharpd`.

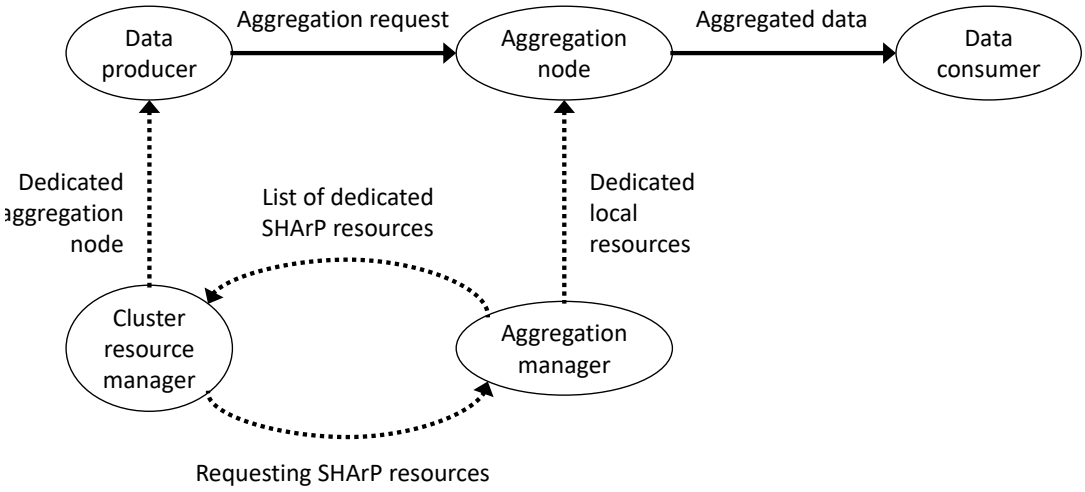

Figure 3.14: SHArP's [10] basic topology and entities

All ANs must form an aggregation tree. Multiple trees are allowed in a system. Similarly to other in-network aggregation solutions like Daiet [20], data is aggregated along the aggregation tree by ANs, until it reaches a root AN that is in charge of distributing the result.

The protocol also introduces the concept of *group*, consisting in a subset of physical hosts that are connected to the tree leaves: for instance, in MPI [25] a group coincides with a *communicator*. One aggregation tree supports multiple groups.

Resources are managed by a special management entity called Aggregation Manager (AM). Faults and errors are always notified to this node, that will also take care of freeing all resources belonging to the tree in which the error occurred. The detection of faults and errors cannot be done using timeouts since HPC APIs do not bound the duration of aggregation operations: this is why faults must be necessarily detected by monitoring.

Data flow. Eventually, an host program will need to execute a job on multiple nodes. When the job is launched and all host processes have been created (e.g., a communicator in MPI [25]), either the cluster resource manager (like Slurm [27] or IBM® Spectrum LSF) or an MPI [25] launcher like `mpirun` will contact the AM, which will dedicate SHArP [10] resources to the job and return back the list of these allocated resources. At this point, a SHArP [10] group has been created. Each SHArP [10] daemon `sharpd` running on every group member will establish

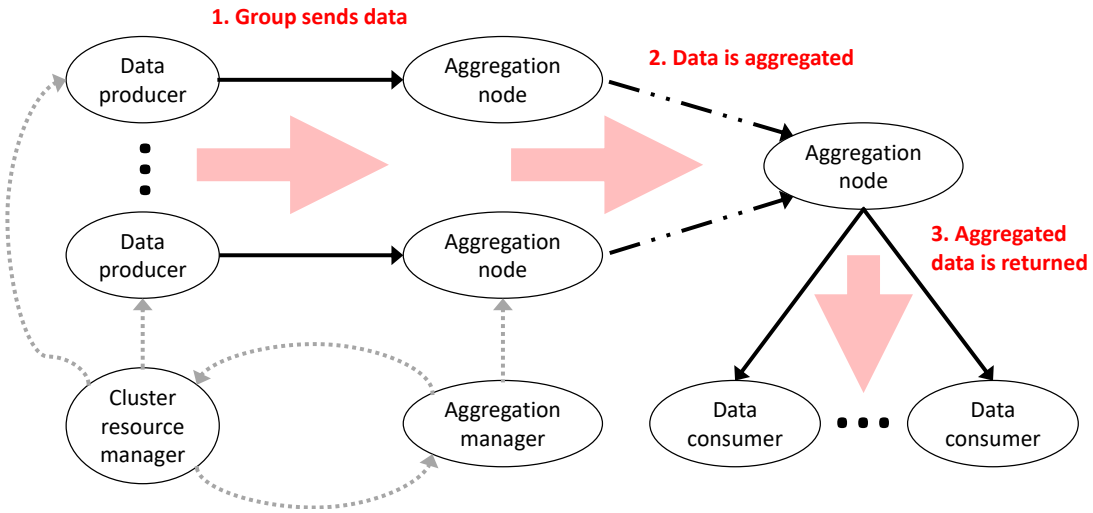

Figure 3.15: SHArP's [10] logical communication pattern

a reliable connection to the dedicated leaf switch. The AM informs the ANs about the switch resources allocated to the application, not allowing it to exceed the allocation.

Once connections have been established, all group members send an aggregation request message to their parent AN. Each AN waits for all its children requests before sending the aggregated data piggybacked on another aggregation request to the parent node. ANs temporarily maintain a data structure to track an aggregation operation's progress.

As soon as the tree root node receives data from all its children, it performs the final aggregation and it sends the result to a destination, that could be (i) one or more process belonging to one or more groups or (ii) an external process not belonging to any reduction group. In the former case, the aggregation tree is used to redistribute the result.

3.2.4.2 Implementation

SHArP [10] has been implemented using InfiniBand [16] as communication standard with Mellanox's SwitchIB-2^{TM} devices, which provide support for data reduce operations and for barriers. Nodes in the SHArP [10] tree are InfiniBand [16] end nodes, and links are implemented using InfiniBand's [16] *Reliable Connection*s. When distributing a result using a multicast address, though, an unreliable delivery mechanism is used. One AN can participate to at most 64 different aggregation trees.

3.2.4.3 Minimum system requirements

ANs (usually run by network devices) must form a tree whose leaves are connected to data producers (a *communicator* in MPI [25]). Each data producer is connected to only one tree leaf. The root AN, instead, is connected to the data consumer, which receives the final aggregated result. ANs must dedicate part of their local memory to the system.

The special management unit (AM) must act as a SHArP [10] RM, dedicating SHArP [10] resources to those entities who request for them.

3.2.4.4 Conclusions

For MPI [25] applications SHArP [10] brings a significant advantage in terms of latency: tests show that the latency improvement factor (latency experienced without SHArP [10] divided by

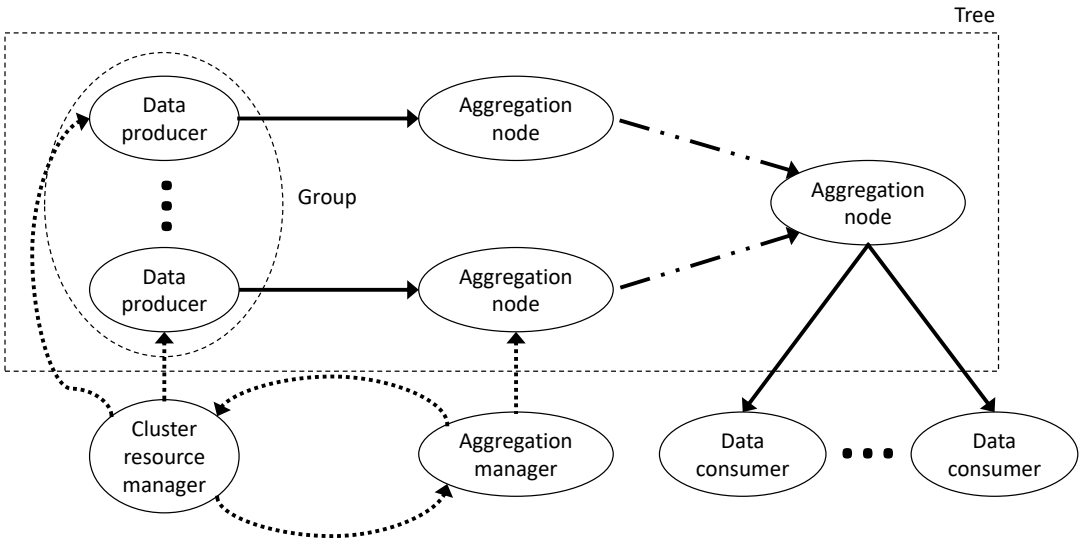

Figure 3.16: SHArP's [10] extended topology

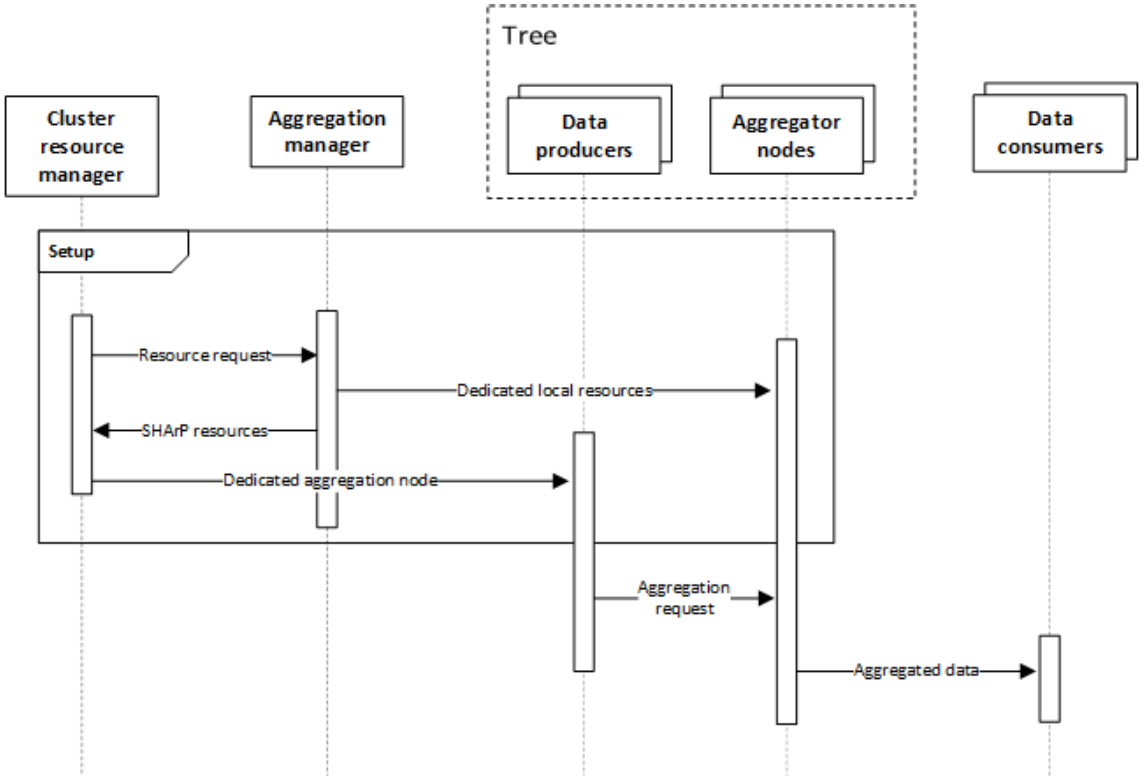

Figure 3.17: Messages exchanged during a SHArP [10] instance execution

the one experienced with SHArP [10]) is proportional to the message size. Even in the worst case, with a message size of just 8 MB, an MPI [25] execution with SHArP [10] is twice as fast as the one without using it.

3.3 Integrating INP resources in Resource Managers (RMs)

There is currently no RM nowadays that (i) handles all logical server, switch and edge resources, (ii) supports high-level requests (e.g., with the use of composites) and (iii) is capable of considering multiple resource dimensions. However, there are different schedulers that fulfill, even partially, a subset of those requirements. This paragraph tries to cover all the aspects of INP-integration from a theoretical point of view.

3.3.1 Interaction with different architectures

Different RM scheduling architectures were introduced in § 2.1.2. This section tries to analyze how can different scheduling architectures manage INP resources, highlighting all the drawbacks brought by each one of them. This brief analysis does not make any assumption regarding the INP resource model since its format is out of the scope of this specific section.

Monolithic. In this architecture, resource requests cannot be concurrent by definition since there is just one logically centralized scheduler. However, this architecture does not scale due to its intrinsic nature. Such a scheduling architecture could be used later on to build up a simple dummy INP-aware scheduler that might act as the standard baseline used to compare the actual scheduler developed in chapter 5, but it will be not considered during the design phase of the latter since it is not scalable.

Two-level. In the two-level architecture a centralized node master (or *allocator*) proposes resources to the schedulers. Let us suppose that the allocator not only proposes server resources, but also INP ones: during the period of time which lasts from the instant in which resources are offered until the scheduler accepts or denies the offer, all resources must be locked. Locking INP resources though means locking either network devices' resources (to perform computation) or links' bandwidth (for guarantees), which can heavily affect the performance of all nodes using these shared network resources.

One could think of not locking INP resources during the delay introduced by the scheduler when accepting or denying the request, but the allocator would need to repeat the whole resource offering process if a scheduler accepts some INP resources that have been acquired by some other scheduler during this period of time.

Shared-state. A shared-state scheduler could include the INP resources state in the *cell state* data structure. Supposing that tenant applications can request for both server and INP resources, schedulers will try to acquire those by modifying the cell state through an atomic commit. Schedulers that will have to satisfy requests containing supposedly-longer resource request lists are more likely to cause an higher amount of conflicts when attempting to write the share cell state data structure. This could heavily affect the whole RM performance.

3.3.2 RMs' network awareness levels

Existing RMs have a different awareness level of switch and link resources. This is one possible division that groups RMs in three groups.

VMs proximity-aware. Most of the RMs out there can spread VMs across different failure domains like machines, racks and power domains. This group is not worth discussing since this kind of RMs do not consider any kind of resources rather than server ones. Some examples of VMs proximity-aware RMs are Omega [21], Apache^{TM} YARN [23] and Mesos [11].

Bandwidth-aware. Some RMs like CloudMirror [14], Oktopus [2], Kraken [8] and Proteus [26] allow tenants to specify bandwidth demands. These RMs let tenants express their requests by using "virtual network" models like Virtual Clusters (VCs), Time-Interleaved Virtual Clusters (TIVCs), Virtual Oversubscribed Clusters (VOCs) and Tenant Application Graphs (TAGs).

Oktopus [2] and Kraken [8] assume that every VM can be placed on every physical server, completely ignoring server-local resource requirements. This is not acceptable in a real-world scenario in which different logical server resources have different resource requirements. Also, it is worth to notice that Kraken [8] allows tenants to *upgrade* their bandwidth requirements, placing again those VMs that have been placed in parts of the data center in which the new bandwidth requirements cannot be satisfied anymore.

Still, none of these RMs is able to manage any kind of switch resources.

Network resources-aware. The most interesting group consists in those RMs that are actually aware of network resources. To the best of my knowledge, there is only one embedding solution that belongs to this group, namely the one introduced in "On tackling virtual data center embedding problem" [17] by Rabbani, Md Golam, et al. presented in the *IM 2013: IFIP/IEEE International Symposium on Integrated Network Management* conference. This solution allows tenants to explicitly specify (i) logical server resources, (ii) logical switch resources and (iii) bandwidth demands as logical edge resources.

Tenants use a graph to express their resource requests. The graph is expressed as a key-value map that includes (i) a set of VM resources, (ii) a set of logical switches resources and (iii) a set of logical links connecting the above entities (and their minimum required bandwidth).

Its placement algorithm is interesting for the scope of this thesis since it is the only one that takes into account all three types of resources mentioned before. The placement phase is divided in three parts: (i) the VMs placement, (ii) the logical switches placement and (iii) the logical links placement. The problem of placing VMs is reduced to a min-cost flow one like showed in Figure 3.18.

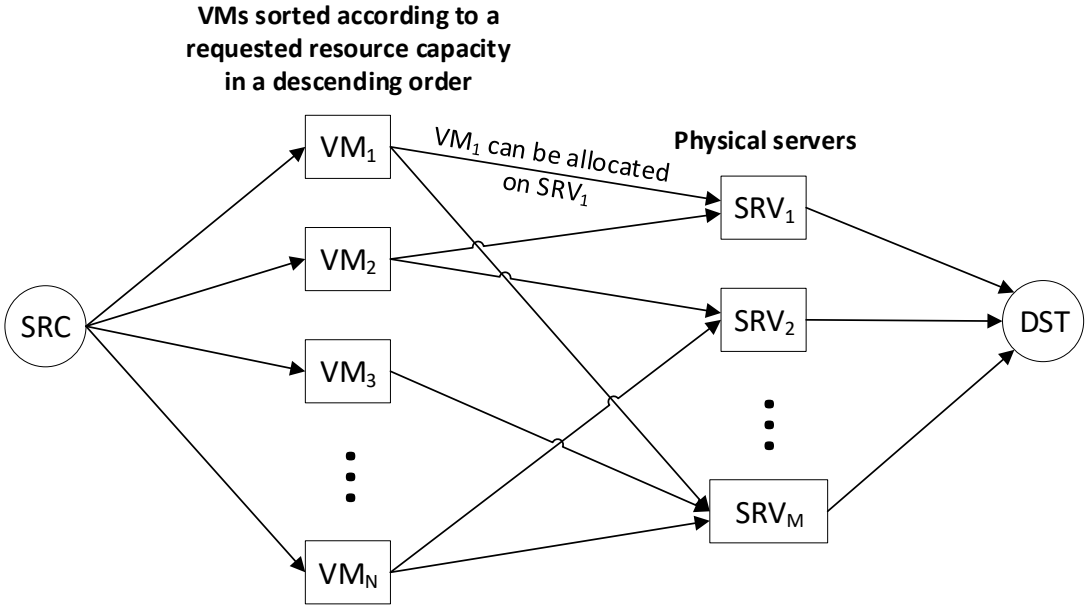

Figure 3.18: VM placement in [17]

The graph shown in Figure 3.18 is built in the following way: VMs are sorted according to a requested resource capacity in a descending order and placed in the left side of the image. Physical server are instead placed in the right side. VMs are connected to physical servers only if they can be allocated on them. In order to reduce this placement problem to a min-cost flow one, a dummy source and destination node are added like shown in Figure 3.18 so that an instance of a min-cost flow problem solver can be run. The outcome of this phase provides the allocation of VMs on physical servers.

It is important to notice how VMs are sorted based on just one resource dimension. Even though the model used in [17] supports an infinite number of resource dimensions, the placement algorithm only supports one dimension and extending it to support multiple dimensions is not trivial.

Same thing is done for the placement of logical switches, as shown in Figure 3.19.

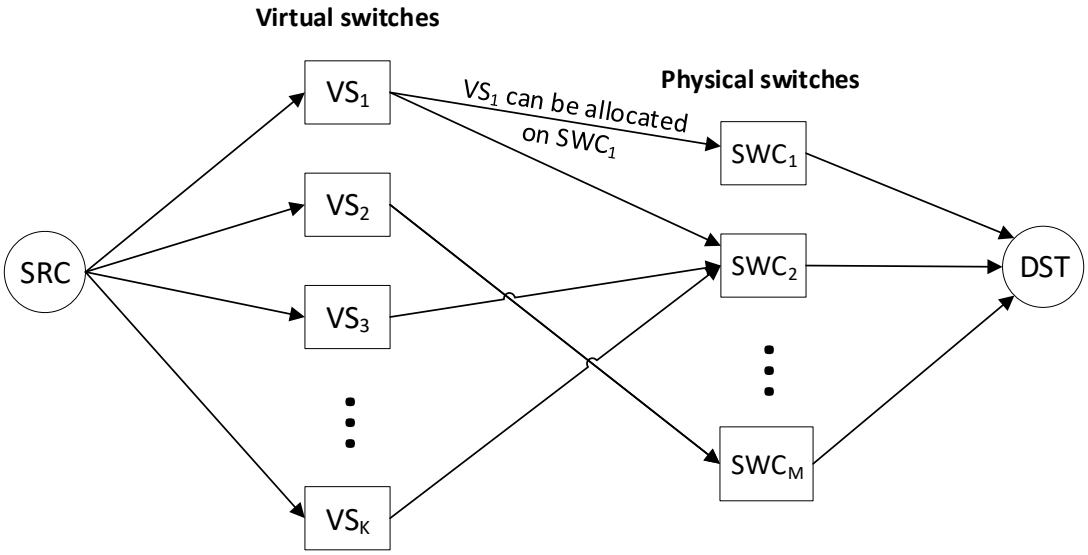

Figure 3.19: Logical switches placement in [17]

The mapping of logical switches to physical ones is done independently of the outcome of the previous phase (i.e., VM placement). Figure 3.20 shows how this could lead to a bandwidth waste in case a logical switch connecting two VMs is mapped to a physical switch that is far away from the physical servers on which the VMs have been previously placed.

The third and last step simply consists in mapping logical links between two logical entities (VMs or switches) to the shortest physical path connecting the physical devices on which the logical entities have been allocated.

The placement algorithm is not fault tolerant since it tries to map as many VMs as possible to the same physical server in order to minimize server resource fragmentation.

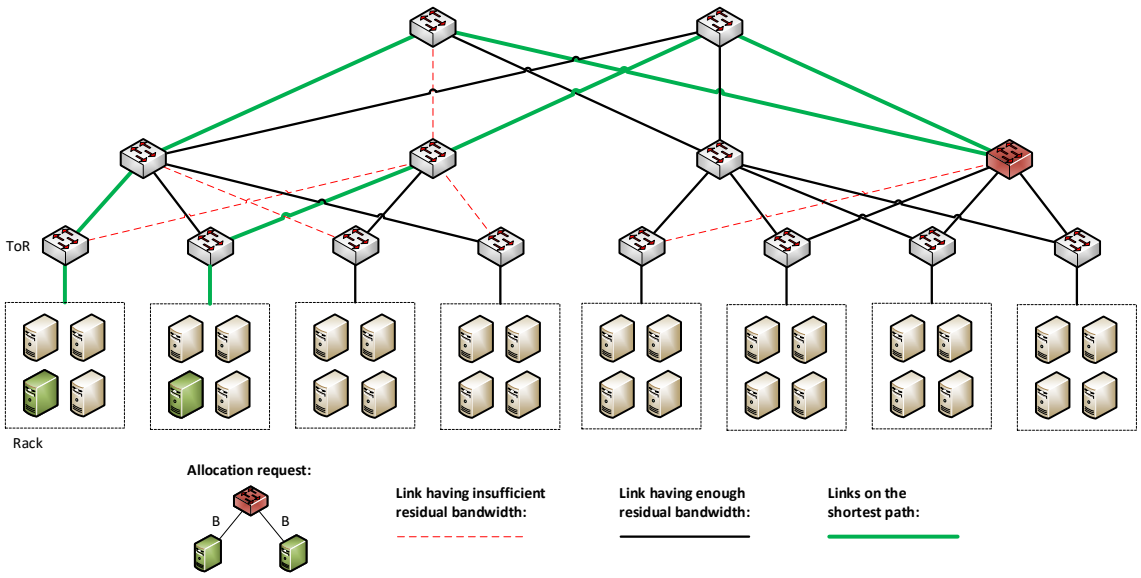

Figure 3.20: An example of an inefficient logical switch placement in [17]

3.4 Resource models

Several network abstractions have been proposed to provide simple APIs to tenant applications, requiring them to only specify high-level resource requests without knowing the actual data center infrastructure.

The section contains a brief analysis of resource models used by tenants and a short discussion about their ability to describe INP solutions.

3.4.1 Virtual Cluster (VC)

Proposed in the Oktopus [2] paper, a Virtual Cluster (VC) is a logical one-level tree in which N VMs are connected to a single *virtual switch* by a bidirectional link of bandwidth B.

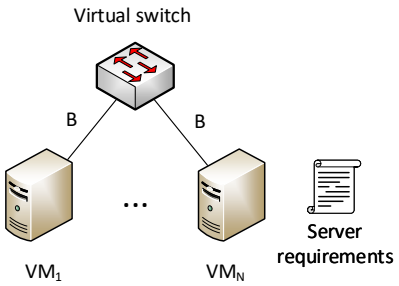

Figure 3.21: A graphical representation of the VC model

The virtual switch has a bandwidth of $N \cdot B$, hence the VC has no oversubscription, unlike the Virtual Oversubscribed Cluster (VOC) model described in § 3.4.2. Authors say that the absence of oversubscription makes it suitable for data-intensive applications.

3.4.1.1 Usage

Kraken [8]. The VC is the only resource model used by Kraken [8]. The system allows tenants to update their minimum guarantees in terms of bandwidth and the amount server resources.

However, tenants can only express the number of needed computing resources (called *computing units*) and not their internal requirements (like CPU cores, memory, etc.). The assumption that all resources are the same is not acceptable in an INP scenario.

Oktopus [2]. Tenant applications can choose one of the three following options to express their request: they can either (i) stick with the classic resource request by listing individual server resource demands without expressing bandwidth guarantees, accepting to simply get some share of the network resources, (ii) choose a VC, most likely used to request resources for data-intensive applications which do not tolerate oversubscribed networks or (iii) specify a VOC for those applications having more intra-component communication than an inter-component one.

Yet, when the Oktopus [2] system receives a request expressed with a VC, it assumes that VMs can be allocated on any server with enough *free VM slots*, exactly like Kraken [8] with its *computing units*.

3.4.2 Virtual Oversubscribed Cluster (VOC)

Also proposed by Oktopus [2] authors, the Virtual Oversubscribed Cluster (VOC) model consists in N VMs arranged in groups of size S. Similarly to the VC model, VMs belonging to the same group are connected to a single virtual switch by a bidirectional link of bandwidth B. Therefore, every virtual switch connecting groups of S VMs has total bandwidth of $N \cdot B$. All groups are then connected together by a unique *root virtual switch*, making this model's topology a two-level logical tree.

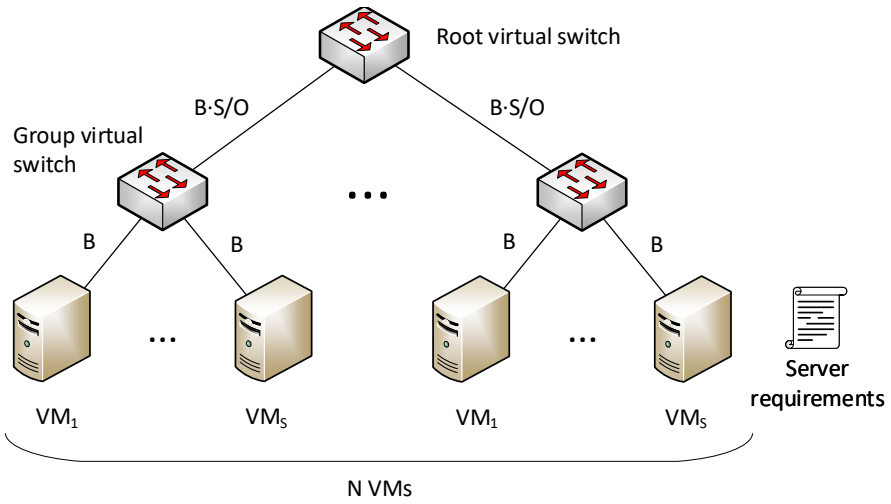

Figure 3.22: A graphical representation of the VOC model

Links connecting all virtual switches to the root one have an oversubscription factor of O, meaning that each of those links has a bandwidth of $S \cdot B/O$. The VOC model aims to relax the dense connectivity requirement between all VMs, having oversubscription just for inter-group communication.

3.4.2.1 Usage

Oktopus [2]. Tenants who use a VOC model for their requests are aware of the bandwidth oversubscription for inter-group communications as they explicitly have to specify the over-

subscription factor O in the reuqest. Oktopus [2] though still does not make any distinction between server resources, making it an unpractical solution for handling INP resources.

3.4.3 Tenant Application Graph (TAG)

As described in § 3.1.5.2, the Tenant Application Graph (TAG) model is a directed graph in which vertexes represent server components and links' weights represent the requested sending and receiving bandwidth, respectively, SB and RB.

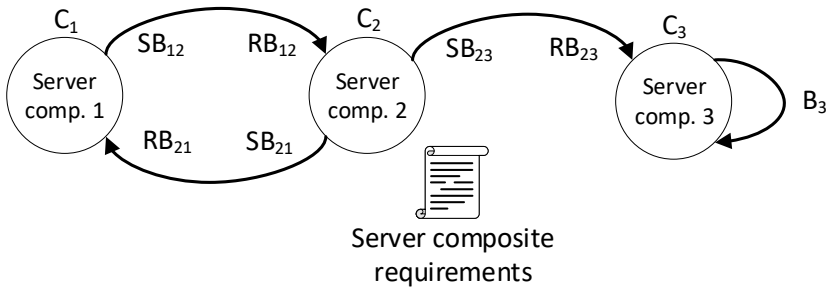

Figure 3.23: A TAG example

The model has been introduced and used by CloudMirror [14] since the previously-mentioned models are inefficient as they over-allocate bandwidth (§ 3.1.5.1). This model is decoupled from any network topology, allowing RMs to be more flexible during resource allocation.

3.4.3.1 Usage

CloudMirror [14]. The CloudMirror [14] placement algorithm just considers *VM slots*, assuming that all server resources have the same requirements. This is not true for INP applications since different INP resources require different network devices specifications. The placement algorithm tries to find lowest sub-tree in the physical topology that can host the number of requested VMs taking into account the bandwidth requested between those: this means that the scheduler is aware of the residual bandwidth on links, but it considers all network devices to be the same.

3.4.4 Fine-grained resource requests

Most of state-of-the-art RMs [11, 24, 21, 18, 23] deal with resources in the simplest yet most descriptive way: a list of server-only resources demands (e.g., CPU cores, memory, etc.).

3.4.4.1 Usage

Mesos [11]. First, the Mesos [11] logically centralized resource allocator offers different fine-grained server resources to different schedulers. Schedulers will then reply to the allocator with the information about the tasks to be run on servers. The allocator is completely network-unaware and hence it cannot neither offer any other kind of resource types nor accept tasks to be run on network devices.

Oktopus [2]. In Oktopus [2], tenant applications can use (i) the VC model to express virtual networks characterized by a dense connectivity, (ii) the VOC model for oversubscribed virtual networks and (iii) a fine-grained resource list to request server resources only. The system makes

a distinction between requests expressed by a *virtual network* (the first two options) and those that are not (fine-grained list). The latter kind of requests has the lowest priority, meaning that the bandwidth not used by virtual networks is equally distributed amongst applications that did not request for any bandwidth guarantee. In conclusion, when sending a resource request to Oktopus [2] expressed by means of a resource demand list, the RM will not take care of bandwidth guarantees since there is no way for tenant applications to specify these demands.

Borg [24]. Borg [24] does not support any network abstraction and it only allows tenants to request resources by listing them explicitly. A scheduler first step's consists in finding the subset of machines on which the task could run, and while doing this, it does not consider any kind of aspect regarding the network. During the second step, namely scoring, which consists in finding the best subset of machines on which the task can be allocated, the only network aspect taken into consideration is the failure domain: tasks are allocated across failure domains (e.g., machines, racks, power domains) for fault-tolerance.

Apache™ YARN [23]. Besides containers properties, tenant applications issuing requests can include locality preferences (e.g., node-level, rack-level and global locality preferences). Exactly like Borg [24], this is the only network aspect taken into consideration during a job placement.

3.4.5 High-level goals

Rather than a fully descriptive resource model, high-level goals specification can be seen as an add-on to the other previously-mentioned models that is worth mentioning for the purpose of this thesis. Taking Bazaar [12] as an example, one possible high-level goal is the job completion time. Goals like these eventually need to be translated into classic resource requirements to be correctly interpreted by a RM.

Chapter 4

Requirements

This chapters lists a set of requirements that the system described in chapter 5 must satisfy.

4.1 Resource model

Here is a list of functional and non-functional requirements for the resource model.

The model exposed to tenants (tenant-side model) and the one internally used by the RM (RM-side model) could be different.

4.1.1 Functional requirements

FR1 The tenant-side model must be able to describe server and INP composites.

FR2 The tenant-side model must be able to describe all kinds of INP composites.

FR3 The RM-side model must be able to describe all kinds of INP composites.

FR4 The tenant-side model must allow the tenant to specify different bandwidth demands for different composites and INP composites.

FR5 The resource model must be able to describe any kind of network topology.

FR6 The tenant-side model (and the corresponding APIs) must not change upon the release of new INP solution or version.

FR7 The translation from composite requirements to logical resource requirements must be done by the RM and not by the tenant application.

4.1.2 Non-functional requirements

NFR1

4.2 Scheduler

Chapter 5

Design

5.1 INP aspects of interest to RMs

- (Aggregation) Tree
  - **Leaf aggregator switches fan**: since aggregator switches in Daiet [20] need to wait data from all its children, the higher the fan is, the more time a switch will need to wait, the slower the system will perform
  - **Tree height** and the **number of switches**: the higher, the more messages, the smaller messages are; the shorter, the less messages, the bigger messages are
  - **Data producers distribution**: data producers sending data with the same key should be connected to "close" leaves in order for them to perform data aggregation lower in the tree
  - **Tree balance**: an extremely unbalanced tree could cause some parts of it to wait for data to be aggregated in the deepest parts
- Chain
  - **Chain length**: the longer, the more reliable the system is, the higher the latency for write queries in NetChain [13]

5.2 System design

This section contains a brief description of how the whole system works: from the tenant resource request to the actual resource placement in the physical topology.

5.2.1 Overview

As required by **FR1** and **FR2**, tenant applications must be able to express their requests in terms of logical resources and composites. The latter are just a simplification for tenant applications, and they need to be translated into a set of logical resources so that the placement algorithm could allocate those. This translation can be done by the RM by means of a *template database*, that maps pre-determined composites into their equivalent made out of just logical resources, as shown in Figure 5.1.

Logical resources represent the input of the placement algorithm. Those will be assigned to physical resources, as shown in Figure 5.2.

The whole system design is depicted in Figure 5.3.

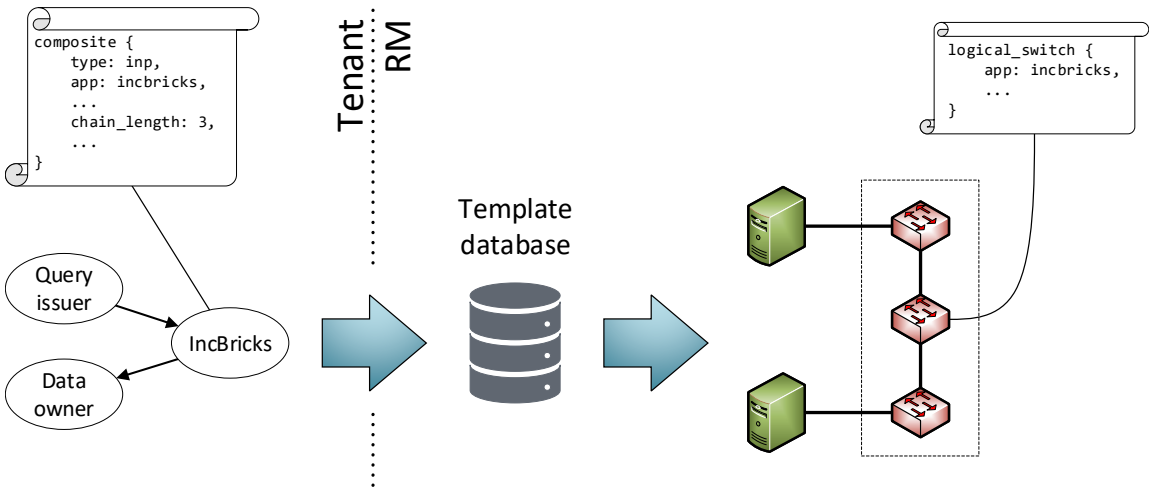

Figure 5.1: Translating composites to logical resources

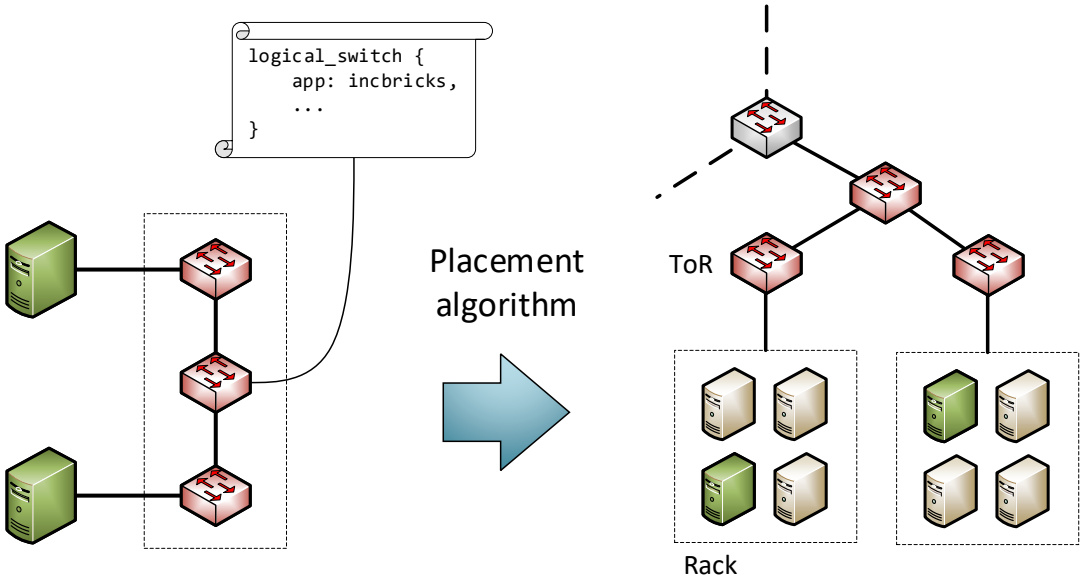

Figure 5.2: Placement of logical resources to physical resources

5.2.2 Composites translation methods

Composites can have multiple properties specifying some application's requirements or constrains.

A first approach called *passive mapping* would require the tenant application to explicitly express internal composites' properties that **directly** affect its equivalent expressed in terms of just logical resources. Figure 5.4 shows an example of a tenant application explicitly specifying the chain length of its IncBricks [15] composite and the bandwidth demands B_1 and B_2 towards and outwards it, respectively. This of course increases the expressiveness of the tenant application with the cost of making the interface more complex to use.

With the opposite approach (*active mapping*), tenant applications do not have to specify internal composites' properties, but instead more abstract performance goals. These high-level composites' goals will be then translated by the RM. An example of this translation is showed

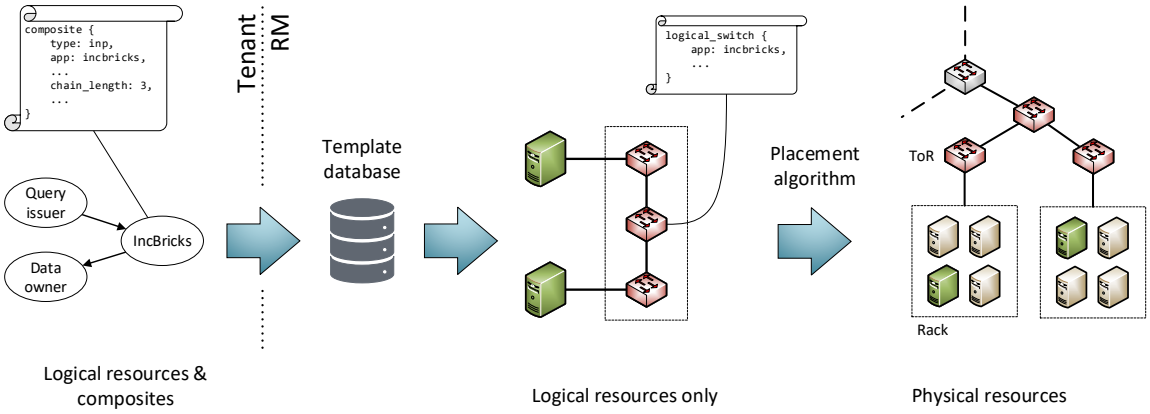

Figure 5.3: From the tenant-side model to physical resources

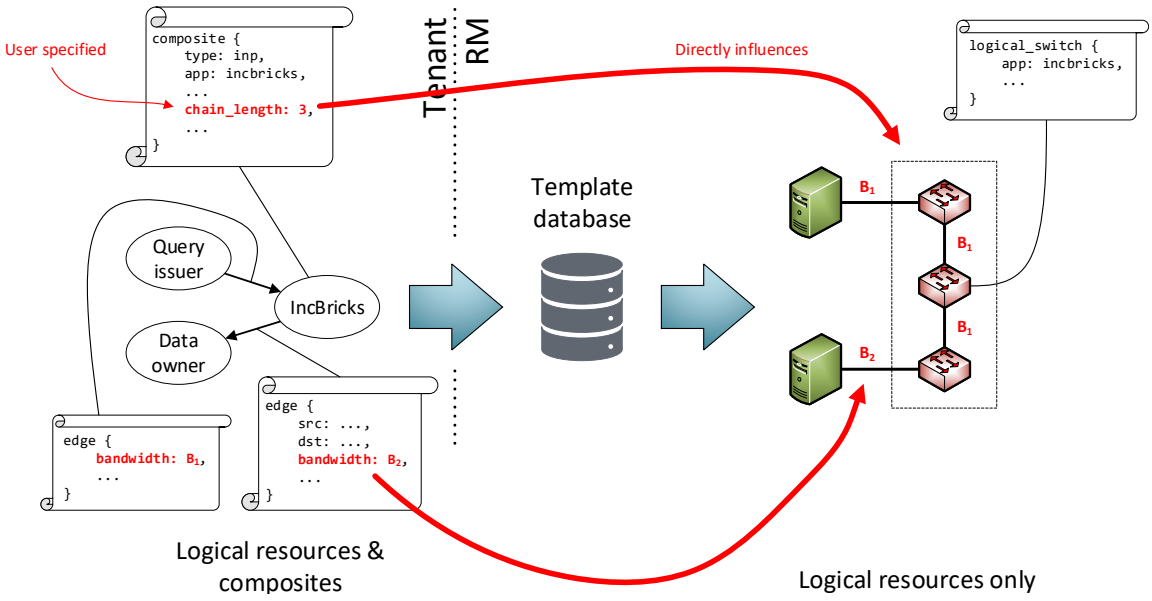

Figure 5.4: Passive template mapping

in Figure 5.4, where the requested tuple rate is transformed accordingly into topology and bandwidth constraints. This approach simplifies the interface exposed to tenant applications by not letting them taking care of internal composites properties that might be unknown to developers.

5.3 Resource model design

In order for tenant applications to request INP resources, there must be one or multiple models capable of describing them. This chapter will first discuss the integration of INP resources in currently used models described in § 3.4 and will then introduce a proposed model which must be capable of describing existing INP solutions and possibly even future ones.

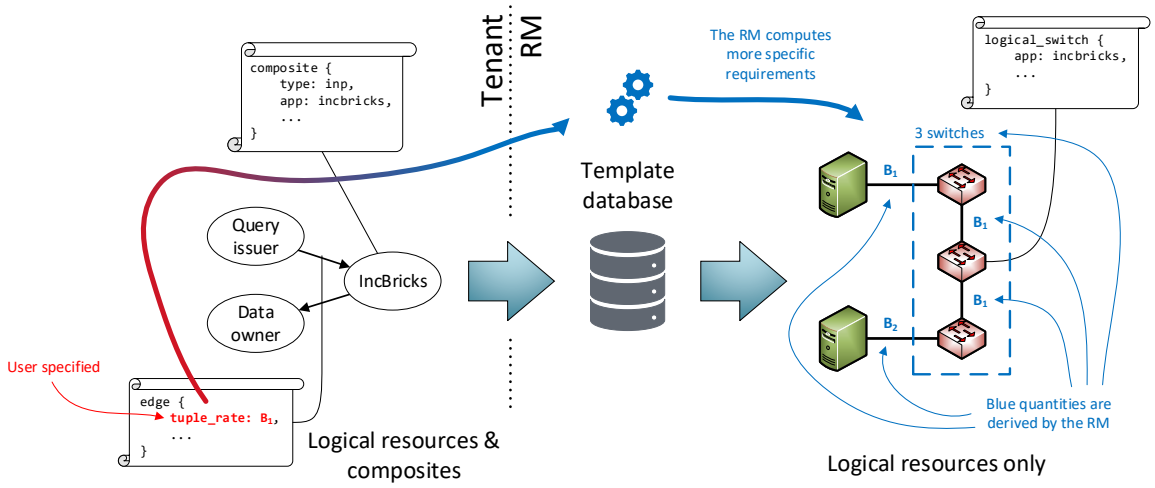

Figure 5.5: Active template mapping

5.3.1 Integrating INP resources in existing resource models

Different resource models (§ 3.4) are currently used by RMs to offer and manage data center resources. Most of the times they allow the description of logical server resources only, but some Resource Management Frameworks (RMFs) like [8, 14, 2] also provide bandwidth provisioning by allowing tenants to request for logical edge resources. A lot of them are network-aware in the sense that they are aware of failure domains. [17] allows tenants to request for logical switch resources but it does not support the requests of logical edge resources and composites.

For each resource model previously described in § 3.4 this section will discuss whether it can be used for this system, using the requirements list reported in § 4.1.

5.3.1.1 Virtual Cluster (VC)

Quick recap. The VC model consists in N VMs connected to a single *virtual switch* by a link of bandwidth B. Figure 5.6 shows an example.

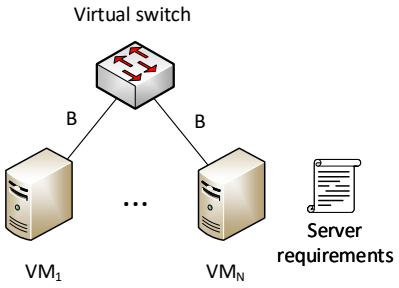

Figure 5.6: A graphical representation of the VC model

Requirements check. Expressing just one bandwidth value B for all connections is obviously a big limitation which would cause a waste of bandwidth for those applications having nodes that require different amounts of bandwidth. This does not satisfy ✗ **FR4**. The Time-Interleaved Virtual Cluster (TIVC) model introduced in Proteus [26] overcomes this limitation since bandwidth constraints are expressed by time-varying functions instead of a constant fixed

value. Both models though (i) rely on a single virtual switch that is not suitable for tenant applications which require a more complex logical switch topology such as a chain or a tree (✗ **FR5**) and (ii) do not support logical switch resource requests (✗ **FR2**).

5.3.1.2 Virtual Oversubscribed Cluster (VOC)

Quick recap. The VOC model is the oversubscribed extension of the VC one. Figure 5.7 shows an example.

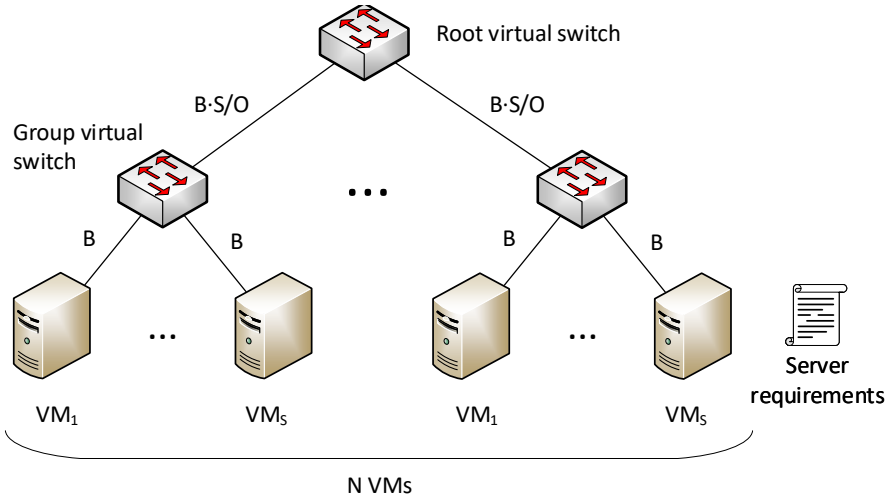

Figure 5.7: A graphical representation of the VOC model

Requirements check. The VOC model has the advantage of not requiring a virtual switch of bandwidth $N \cdot B$, that could be unpractical when N is particularly large. When a tenant application issues a resource request using a VOC, it just requests for the $< N, B, S, O >$ tuple, being respectively (i) the number of VMs, (ii) the bandwidth needed for intra-group communication, (iii) the group size and (iv) the oversubscription factor. Still, certain kinds of applications might need different values of B for different groups (✗ **FR4**). Furthermore, the two-layer logical tree topology might not suit perfectly for applications that require a chain having more than 2 network devices or a tree topology with more than 2 layers (✗ **FR5**). The VOC model still does not support logical switch resource requests (✗ **FR2**).

A possible variant. The two limitations previously mentioned can obviously be overcome by allowing to specify (i) logical switch resources (satisfying ✓ **FR2**) (ii) different bandwidth demands for different groups (satisfying ✓ **FR4**) and (iii) an arbitrary tree height. The corresponding variant is shown in Figure 5.8.

However though, tenant applications still (i) cannot express any kind of switch topology (e.g., a switch loop for an in-network Distributed hash table (DHT) chord), violating ✗ **FR5**, and (ii) convert INP high-level requirements (e.g., in-network total cache size, lock requests per second, ...) into switch resource requirements (✗ **FR7**).

5.3.1.3 Tenant Application Graph (TAG)

Quick recap. A TAG is a graph where vertexes represent server composites, connected by logical edge resources. Figure 5.9 shows an example.

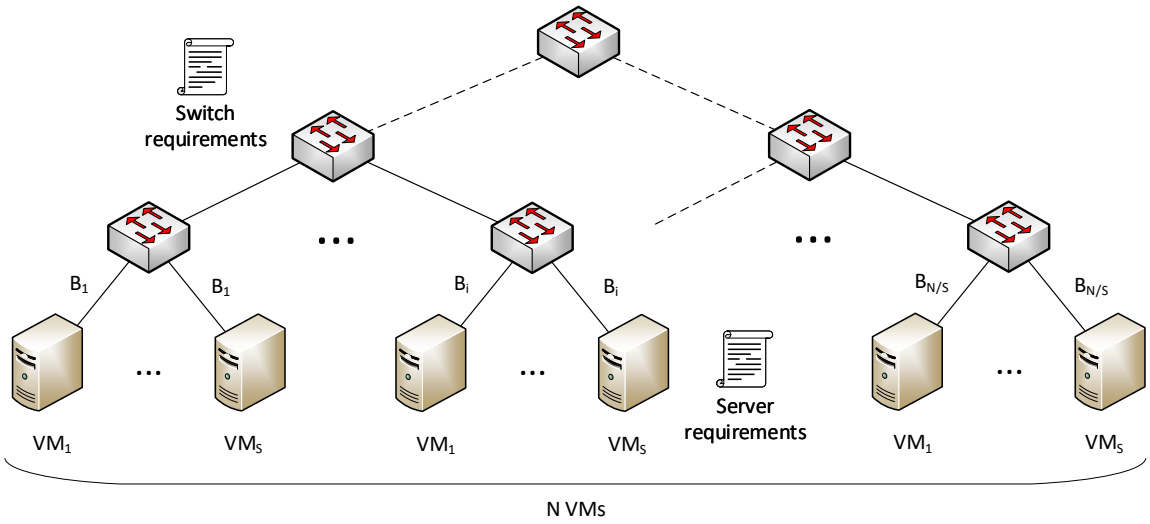

Figure 5.8: A possible VOC variant with an arbitrary tree height

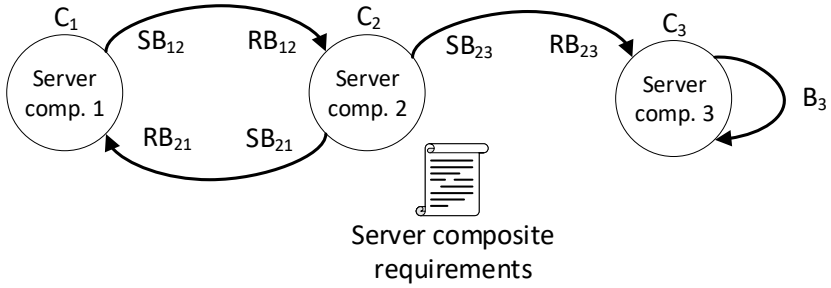

Figure 5.9: A TAG example

Requirements check. The model does not allow tenant applications to request for INP composites (✗ **FR1**) and for logical switch resources (✗ **FR2**). Since this model satisfies all other requirements, it will be used as a starting point for the resource model proposed in § 5.3.2.

5.3.2 Model proposal

The proposed model is based on TAG and it allows tenant applications to specify (i) all composites (✓ **FR1**) and logical resources (✓ **FR2**) (ii) different bandwidth demands (✓ **FR4**) for different entities (iii) any kind of network topology (✓ **FR5**). An example of proposed model is depicted in Figure 5.10.

It is worth noticing that with the introduction of composites in the model, the translation to logical resources must be done by the RM and not by the tenant application (✓ **FR7**).

5.4 Generic groups

5.4.1 In-network caching

Example:

- IncBricks [15]

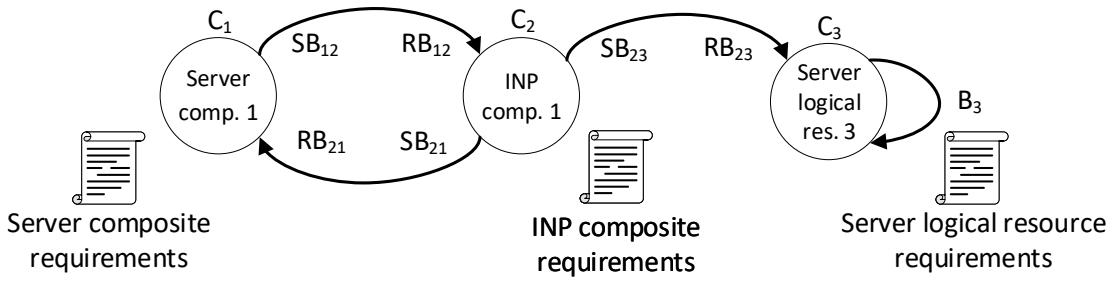

Figure 5.10: The proposed model based on TAG

Network devices must:

- form a chain that connects the two communicating nodes
- dedicate part of their local memory to store a key-value cache

The query issuer:

- executes queries
- is a logical server resource
- must be connected to the head switch

The data owner:

- owns data
- is a server logical resource
- must be connected to the tail switch
- is not the query issuer
- can reply to queries in case no switch has cached the data

The SDN controller must:

- be connected to all the network devices
- configure network devices in order for them to forward IncBricks [15] packets along the switch chain

5.4.2 In-network coordination

Example:

- NetChain [13]

Network devices must:

- form a chain
- dedicate part of their local memory to store a distributed map
- execute coordination primitives

Query issuers:

- execute queries
- are logical server resources
- must directly contact either the head switch or the tail one
- must include the list of IP addresses of all switches belonging to the chain in the packet

The SDN controller:

- must be connected to all the network devices
- must form the chain and handle switches reconfigurations

5.4.3 In-network data aggregation

Examples:

- Daiet [20]
- SHArP [10]

Network devices must:

- form a tree whose root is connected to data consumers and whose leaves are connected to data producers
  - Daiet [20]: just one data consumer
  - SHArP [10]: one or more data consumers
- dedicate part of their local memory to store a key-value map
- be able to perform basic operations on data, such as writing and hashing
- wait for all its children to send aggregated data

Data producers must:

- be connected to exactly one tree leaf

Data consumer(s) must:

- be connected to the tree root

A special unit:

- Daiet [20]: must push flow rules to all switches belonging to at least one tree. It is the SDN controller.
- SHArP [10]: must act as a RM, dedicating SHArP [10] resources to those entities who request for them. It must not necessarily be the SDN controller.

Chapter 6

Evaluation

Chapter 7

Conclusions

Resources glossary

Composite template describing a high-level logical component. It can be made out of other composites and/or logical resources.. 37–41, 43

INP composite composite describing a high-level INP application, e.g., *IncBricks* [15], *NetChain* [13], etc. 37, 43

Server composite composite describing a high-level server component, e.g., *web server*, *databases*, etc. 42

Logical resource logical representation of a physical resource. 14, 37–39, 43

Logical edge resource properties of virtual connections between two logical resources, e.g., bandwidth, latency, etc. 41, 42

Logical server resource virtualized server physical resource, often implemented by means of a VM, container or an entire physical server. 41

Logical switch resource logical representation of a physical switch resource not mapped to any physical switch device. 41–43

Physical resource physical hardware component of limited availability within a physical machine. 14, 38–40

Physical server resource resource of a physical server machine.

Physical switch resources resource of a physical switch, network accelerator, middle-box and of every kind of network device originally intended to forward packets.

Resource model model capable of describing composites and logical resources. The model exposed to tenants and the one internally used by the RM could be different. 37, 41, 43

RM-side model resource model internally used by the placement algorithm in order to allocate logical resources. 37

Tenant-side model resource model exposed to tenants by the system API. 37, 40

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253, New York, NY, USA, 2011. ACM.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[5] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[6] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[8] C. Fuerst, S. Schmid, L. Suresh, and P. Costa. Kraken: Online and elastic resource reservations for multi-tenant datacenters. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.

[9] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Firmament: Fast, centralized cluster scheduling at scale*. Usenix, 2016.

[10] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, et al. Scalable hierarchical aggregation protocol (sharp): a hardware architecture for efficient data reduction. In *Proceedings of the First Workshop on Optimization of Communication in HPC*, pages 1–10. IEEE Press, 2016.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[12] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 10:1–10:14, New York, NY, USA, 2012. ACM.

[13] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, 2018. USENIX Association.

[14] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 467–478, New York, NY, USA, 2014. ACM.

[15] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 795–809, New York, NY, USA, 2017. ACM.

[16] G. F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[17] M. G. Rabbani, R. Esteves, M. Podlesny, G. Simon, L. Z. Granville, and R. Boutaba. On tackling virtual data center embedding problem. In *IM 2013: IFIP/IEEE International Symposium on Integrated Network Management*, 2013.

[18] D. K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly Media, Inc., 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[19] S. Salihoglu and J. Widom. Gps: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[20] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA, 2017. ACM.

[21] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[22] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004.

[23] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[24] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[25] D. W. Walker and J. J. Dongarra. Mpi: A standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[26] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The only constant is change: incorporating time-varying network reservations in data centers. *ACM SIGCOMM Computer Communication Review*, 42(4):199–210, 2012.

[27] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.