Data center resource management for in-network processing

Marco Micera
Politecnico di Torino

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

Data centers distributed systems can nowadays make use of in-network computation to improve several factors: DAIET [12] inventors claim to achieve a 86.9%-89.3% traffic reduction by performing data aggregation entirely in the network data plane. Other solutions like NETCHAIN [7] and INCBRICKS [9] 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, CLOUD-MIRROR [8] 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 is (in part o completely) capable of basic In-Network Processing (INP) operations. The objective of this thesis is to model and evaluate an Application Programming Interface (API) through which applications can ask for resources in a data center exploiting INP capabilities while providing guarantees (e.g., bandwidth).

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

1.2 Problem statement

Using INP to keep scaling data centers' performance seems a promising idea: Daiet [12] inventors claim to achieve a 86.9%-89.3% traffic reduction, hence reducing servers' workload; NetChain [7] can process queries entirely in the network data plane, 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) (e.g., Apache YARN [15], Google Omega [13]) are not completely network-unaware: for instance, some of them are capable of satisfying affinity rules. CloudMirror [8] 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.

1.2.1 Modeling INP resources

This Master's thesis goal 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 capable of asking for INP resources through an API. To do that, INP resources must be modeled not only to support currently existing INP solutions such as [12] [7] [9] [6], 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 [8] 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/or INP services as vertexes or links. Tenants 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 currently existing technologies exploited in data centers nowadays. This chapter starts with a general description of how resource are managed in a data center § 2.1 (e.g., Virtual Machines) and ends up with a brief introduction to 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 being 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 containers. Flexibility and portability are both automatically achieved thanks to this resource virtualization, resulting in some software that can be deployed dynamically, run by multiple platforms and even live migrated; availability is usually simply achieved by not co-locating VMs in a single server or rack.

Furthermore, a Resource Manager's aim is to manage all resources in a cluster and to schedule applications (sometimes referred as *jobs*) assigning the corresponding VMs/containers to the best subset of servers.

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

2.1.1 Scheduler types

2.1.2 Other design choices

2.2 In-network processing

2.2.1 A definition

Within this project, INP refers to the technique that exploits network switches to modify and/or store data packets, without involving any kind of higher-layer devices. Therefore, approaches that make use of middle-boxes do not fall within our definition of INP. This is why INP is different from *active networking* and Network Function Virtualization (NFV).

2.2.2 Examples

2.3 Network Function Virtualization

2.3.1 Examples

Chapter 3

Analysis

Chapter 2 introduced how resources can be managed in a data center and different network techniques such as INP and NFV. 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

3.1.1 Guarantee provisioning: CloudMirror [8]

CloudMirror [8] allows client applications to specify bandwidth and high availability guarantees.

3.1.1.1 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) models 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 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 [8] inventors to come up with a new model.

3.1.1.2 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.

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 vertex a to vertex b is labeled with an ordered pair of numbers $< S, R >$, indicating respectively the guaranteed bandwidth with which VMs in a can send traffic to VMs in b and the guaranteed bandwidth with which VMs in b can receive traffic from VMs in a:

3.1.1.3 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*** ...

3.2 Existing in-network processing 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 [12]

Daiet [12] is a system that performs in-network data aggregation for partition/aggregate data center applications (big data analysis such as MapReduce [5], 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 [11]), hence justifying the usefulness of this system. The system has been designed for P4 [2] and programmable ASICs, and it can be used on any other Software Defined Networking (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.

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. 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. 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 [2], 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.

3.2.1.5 Conclusions

Besides all the drawbacks brought by P4 [2] and 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 [7]

NetChain [7] 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 [7] an acceptable solution only when a small amount of critical data must be stored in the network data plane, e.g., coordination services.

NetChain [7] 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. 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 [7] agent). 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.

Consistency. A variant of Chain Replication [14] 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 [14] protocol is that objects are stored on programmable switches instead of servers. Switches are logically connected together in order to form an oriented chain: 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 [2] 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 [2] drawbacks were already discussed in section 3.2.1.3.

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 [7]: 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.

3.2.2.4 Conclusions

NetChain [7] inventors state that the on-chip memory of programmable switches in enough for coordination services. Assuming a 10 MB partition allocated on each NetChain [7] 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 [3] 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 [9]

IncBricks [9] 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 [9] 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. 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.

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. IncBox units must dedicate some local storage to realize the caching system.

3.2.3.4 Conclusions

3.2.4 Aggregation protocol: SHArP [6]

SHArP [6] 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 [6] is targeted to support the two most used APIs in the HPC area today: MPI [16] and OpenSHMEM [4]. The kind of operations that can be offloaded to the network are: (i) MPI [16] 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`. All ANs must form an aggregation tree. Multiple trees are allowed in a system. Similarly to other in-network aggregation solutions like Daiet [12], 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 [16] 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 [16]), either the cluster resource manager (like Slurm [17] or IBM® Spectrum LSF) or an MPI [16] launcher like `mpirun` will contact the AM, which will dedicate SHArP [6] resources to the job and return back the list of these allocated resources. At this point, a SHArP [6] group has been created. Each SHArP [6] daemon `sharpd` running on every group member will establish 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 [6] has been implemented using InfiniBand [10] as communication standard with Mellanox's SwitchIB-2 devices, which provide support for data reduce operations and for barriers. Nodes in the SHArP [6] tree are InfiniBand [10] end nodes, and links are implemented using InfiniBand's [10] *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 [16]). 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.

3.2.4.4 Conclusions

For MPI [16] applications SHArP [6] brings a significant advantage in terms of latency: tests show that the latency improvement factor (latency experienced without SHArP [6] divided by the one experienced with SHArP [6]) is proportional to the message size. Even in the worst case, with a message size of just 8 MB, an MPI [16] execution with SHArP [6] is twice as fast as the one without using it.

Chapter 4

Requirements

4.1 In-network key-value store

4.2 In-network data aggregation

Chapter 5

System

Chapter 6

Evaluation

Chapter 7

Conclusions

Acronyms

AM Aggregation Manager. 17, 18

AN Aggregation Node. 17, 18

API Application Programming Interface. 4, 5, 17

HPC High Performance Computing. 17

INP In-Network Processing. 4, 5, 7, 8, 10

NFV Network Function Virtualization. 7, 8

RM Resource Manager. 5, 6

SDN Software Defined Networking. 10, 15

TAG Tenant Application Graph. 5, 9

VM Virtual Machine. 6, 8, 9, 16

VOC Virtual Oversubscribed Cluster. 8

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.