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 [8] 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 [4] and INCBRICKS [6] 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 [5] 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 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 [8] inventors claim to achieve a 86.9%-89.3% traffic reduction, hence reducing servers' workload; NetChain [4] 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 [11], Google Omega [9]) are not completely network-unaware: for instance, some of them are capable of satisfying affinity rules. CloudMirror [5] even provides bandwidth guarantees to tenant applicaitons. 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 [8] [4] [6] [3], 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 [5] 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

2.1 In-network processing: a definition

Within this project, in-network processing (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).

Chapter 3

Analysis

3.1 Daiet [8]

Daiet [8] is a system that performs in-network data aggregation for partition/aggregate data center applications (big data analysis such as MapReduce [2], 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 [7]), hence justifying the usefulness of this system. The system has been designed for P4 and programmabile ASICs, but it can be used on any other SDN platform.

3.1.1 Details

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.

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.

3.1.2 Algorithm

To store the key-value map, network devices use two hash tables (with single-element buckets) 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 if the next network device 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 letting network devices send a special *END* packet after transmitting all key-value pairs to the next node.

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

3.1.3 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.

3.1.4 Conclusions

P4 brings the drawback of not having variable-lenght data structures, forcing programmers to waste a lot of memory in case of variable-length keys such as strings. 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 CloudMirror [5]

CloudMirror [5] allows client applications to specify bandwidth and high availability guarantees.

3.2.1 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 [5] inventors to come up with a new model.

3.2.2 Tenant Application Graph

The *Tenant Application Graph* (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:

3.2.3 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*** ...

3.3 NetChain [4]

NetChain [4] is an in-network key-value storage solution.
The network device in charge of storing the distributed storage/cache is a programmable switch: this brings an obvious limitation in terms of storage size, which makes NetChain [4] an acceptable solution only when a small amount of critical data must be stored in the network data plane (e.g., locks). NetChain [4] also processes queries entirely in the network data plane.

3.3.1 Details

A variant of Chain Replication [10] is used in the data plane to handle read and write queries and to ensure strong consistency, while reconfiguration operations are handled by the network control plane.

The main difference with the standard Chain Replication [10] 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 at the end of the chain (the *tail*) while write queries are sent to the *head* of the chain and the state is forwarded along the chain.

The key-value store is partitioned among *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. ***TO BE CONTINUED*** ...

3.3.2 Minimum system requirements

Network devices must form a chain of length f in order to tolerate f failures. A pair of communicating VMs must be connected to at least one network devices belonging to the chain.

Network devices must dedicate some local storage to NetChain [4]: 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.

3.4 IncBricks [6]

IncBricks [6] is a solution 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.

3.4.1 Details

IncBricks [6] 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. The storage has been implemented using a bucket spilling hash table plus a hash index table ***TO BE CONTINUED*** ...

3.4.2 Minimum system requirements

Chapter 4

Requirements

4.1 In-network data aggregation

Examples:

- Daiet [8]
- SHArP [3]

Network devices must:

- form a tree whose root is connected to the sink VM and whose leaves are connected to data producers
- 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

The sink must:

- be connected to the tree root

Data producers must:

- be connected to exactly one tree leaf

4.2 In-network data storage

Examples:

- NetChain [4]
- IncBricks [6]

Network devices must:

- form a chain connecting the two communicating nodes
- store data along the path

Chapter 5

System

Chapter 6

Evaluation

Chapter 7

Conclusions

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.