International Journal of Applied Engineering Research
ISSN 0973-4562 Volume 10, Number 11 (2015) pp. 29539-29554
© Research India Publications
http://www.ripublication.com

Adapting Map-Reduce Programming Model With Container Based Virtualization For Self-Organizing Networks

Premnath KN[1], Dr. Srinivasan R[2] and Dr. Elijah Blessing Rajsingh[3]
School of Computer Science and Engineering, Karunya University, Coimbatore, Tamil Nadu, India
[1]premnathkn@gmail.com [2]srini0402@gmail.com
and[3] elijahblessing@karunya.edu

Abstract

This paper presents and puts forward an execution technique that could potentially address the need of Centralized Self-Organizing Networks (SON) use cases, considering the high data load and quick processing need for network wide data. Key challenges faced in Centralized SON use cases are to do with processing Key Performance Indicators (KPIs) of the network quickly and also to cater the need for evolving network topology. KPIs are generally derived from network events, performance counters that are periodically collected from the multi-technology, multi-vendor and multi-layer Heterogeneous Network. The needs of the SON use cases are addressed by applying well-known Map-Reduce [1] programming model with newly emerging container based virtualization [2, 3] techniques. To demonstrate the validity of proposed execution technique, performances of generic algorithms used by SON use-cases are evaluated. Evaluation results illustrate that these execution techniques can achieve significantly higher performance with commodity hardware.

Introduction

About centralized SON and need of computing.

Centralized SON solution [4, 5] is becoming a de-facto method when network wide data to be considered for SON use cases [6, 7]. Network wide KPIs are generally collected by respective Element Management (EM) or Operations Support Systems (OSS) software. OSS Software is generally provided by respective equipment vendors. Periodicity, data format and KPIs also differ for each technology, though they intend to provide the same high level meaning like network quality, capacity utilization, energy consumption etc., With the introduction of Software Defined

Networks (SDN) [8] the network topology and flow control of the traffic is becoming more dynamic than the traditional statically configured networks. Centralized SON solution typically follows the cycle of "Collecting data" (both network Configuration and KPIs), "Processing data" (based on measurements and performance counters, KPIs) and performs network changes based on the algorithms of SON use cases [9]. Additionally SON use cases are addressed as part of "Self Configuration", "Self Optimization" and "Self-Healing" functionalities [6] which indicates the applicability of SON use cases across all the phases of Network evolution (Network Planning, Design, maintenance and Optimization). All SON use cases [7, 9] can be realized as Centralized SON solution in Network Management System (NMS) level. Further in the paper SON functionalities refers to "Self-Configuration", "Self-Optimization" and "Self-Healing". SON use cases refer to the following mind map derived from 3GPP and NGMN standards [5, 7].

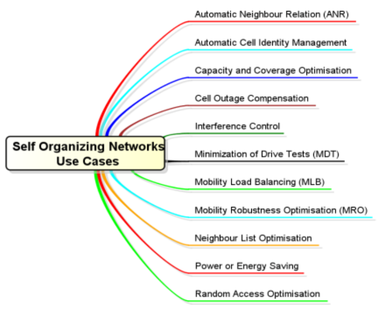
Self Organizing Networks Use Cases
Automatic Neighbour Relation (ANR)
Automatic Cell Identity Management
Capacity and Coverage Optimisation
Cell Outage Compensation
Interference Control
Minimization of Drive Tests (MDT)
Mobility Load Balancing (MLB)
Mobility Robustness Optimisation (MRO)
Neighbour List Optimisation
Power or Energy Saving
Random Access Optimisation

Figure 1: SON Use-cases [5, 7]

The key objective of Centralized SON during "Self Optimization" phase is to continuously tune the network that runs in Optimized level according to the changing network traffic. Some of the prominent SON use cases that require such immediate adaptations are Coverage and Capacity Optimization (CCO), Load Balancing (LB) and Dynamic Automatic Neighbour Relation (ANR). These use cases involves processing large network events, KPIs for every cell and its interference level with neighbouring cells. Current systems are not able to handle such high load quickly. Sometimes the optimization takes more than 8 hours in worst case for technologies like GSM frequency optimization cases. This leads to optimization of network with past condition rather than current behaviour. The expected changes for CCO, LB and ANR are less than 15 minutes. Or sometimes close to real time.

To meet such objectives adapting efficient algorithms [9] is required.

This paper proposes Map-Reduce programming model with container based virtualization for Self-Organizing Networks.

Further paper is organized as follows. Section 2 contains a detailed description of execution technique of Map-Reduce programming model with container based virtualization and additionally with the task distributing principles. Section 3 Execution technique proposal for SON use cases. Section 4 Applying proposed execution techniques on SON ANR Use case Section 5Evaluation and

Analysis of proposed techniques on SON ANR uses case, and followed by a conclusion in Section 6.

Proposed Execution Technique and Principles

The key technical or non-functional requirements for addressing the needs of Centralized SON use cases are:

1. Processing large data sets
2. Parallel execution of algorithm for every intended network element across available resources. (Example: LTE cells, WCDMA cells etc.,)
3. Resilient to failures (both hardware and software crashes)
4. Efficient execution time
5. Shared nothing architecture [10]
6. Able to run according to the availability of resources (Example: Memory, CPU)
7. No single point of failure

From the emerging architectural recommendation in analytics area, Map-Reduce [1] programming model as proposed by google engineers looks very promising in the addressing the above need from I to IV.

Recent advancement in cloud computing [11, 2 and3] looks very promising for the above need from V to VII.

About Map-Reduce Programming Model

Map-Reduce is a programming model, with an associated implementation for processing and generating large data sets [1]. As specified in the Map-Reduce paper [1] many real world tasks are expressible in this model. Programs are specifically written in a functional style. Hence they are automatically parallelized and executed on a large cluster of commodity hardware.

The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the Map-Reduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The Map-Reduce library groups together all intermediate values associated with the same intermediate key "*K"* and passes them to the Reduce function. The Reduce function, also written by the user, accepts an intermediate key "*K*" and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.[1]

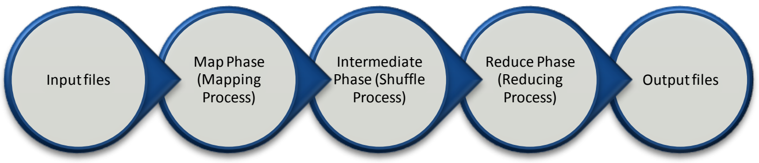
Input files
Map Phase (Mapping Process)
Intermediate Phase (Shuffle Process)
Reduce Phase (Reducing Process)
Output files

Figure 2: Overall process of Map-Reduce

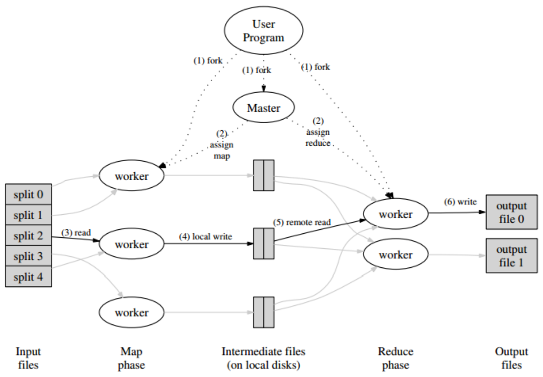

Figure 3: Execution Overview [1]

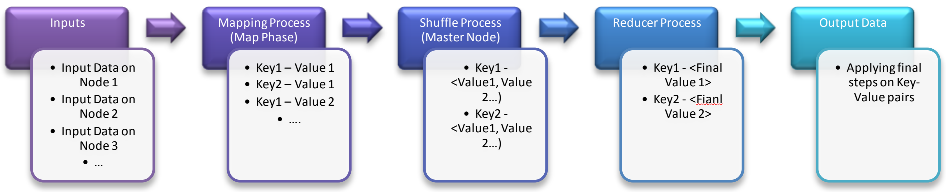

Figure 4: Execution Overview

Due to the above distinct functional phases as in Figure [4], it is easy to run several tasks in parallel.

1. Input files. These are normal static files stored across clusters (with redundancy/backup), normally referred as HDFS (Hadoop Distributed File System) or GFS (Google File System) in case of google. Instead of Input file in our adaptation for SON we will send chunks of network elements that are intended to be addressed by a SON function.
2. Map Phase/Mapping process. In this phase network element performs a specific task. For example, identifying neighboring cells based on distance, hand over statistics etc… For the key ("*K*") "Cell1" the intended values are neighbouring "Cell2" with distance value and so on. Mapping the

neighbouring cell valueis written in Mapper function as part of SON algorithm implementation.

3. Intermediate Phase (Shuffle Process). In this process all the values of a specific key "*K*" are grouped together in individual nodes. For a specific key "*K*" Cell1 all its neighbours are shuffled and grouped together as "Value1, Value2, Value3…"
4. Reducer Phase. In this process the final value forevery key is derived. Example: Consolidating the required neighbours values for "*K*" "Cell1" based on distance criteria. In Shuffle Process "*K*" "Cell1" could have 10 values. But in the reducer phase among the 10 neighbours required neighbours are identified based on criteria (ex: distance).
5. Output Phase. In this process the final steps for the SON function are applied based on the intended algorithm for the SON use case.

About Container Based Virtualization

Cloud computing is an emerging domain with following key elements:

1. Software defined Environments (Infrastructure as a Service (IaaS))
2. Software defined Platforms (Platform as a Service (PaaS))
3. Software defined Networking and Storage (IaaS)
4. Orchestration of Cloud environment

Majority of Open Source Cloud Operating systems like Open Stack [11] are addressing the above key elements.

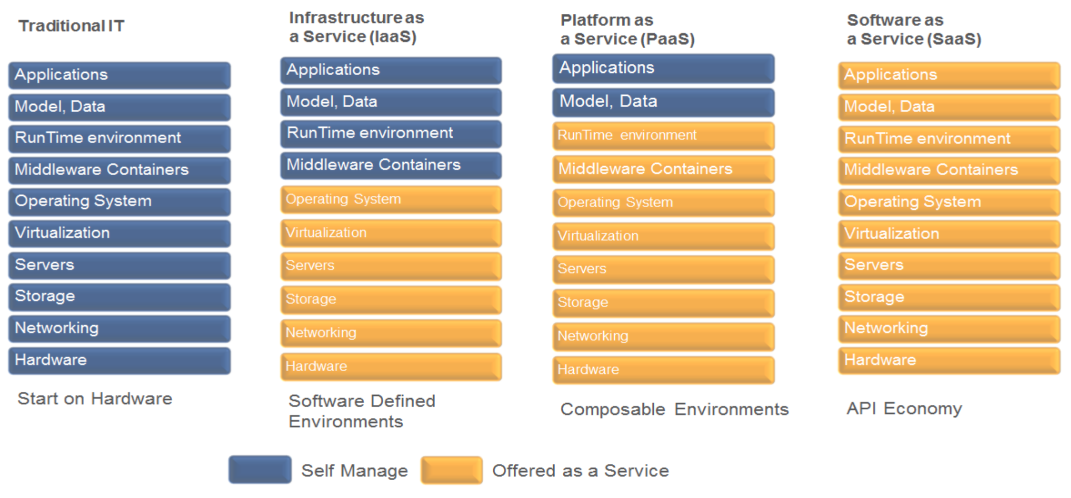

Figure 5: Transition of Software model from "Traditional IT" to "Software as a Service" cloud computing [13]

Hypervisors [12] are the key fundamental invention happened in the past two decades paying a path for virtualization and cloud computing (managing virtualized environment with storage and networking). Virtualization includes all the facets of the hardware like compute node, network and storage.

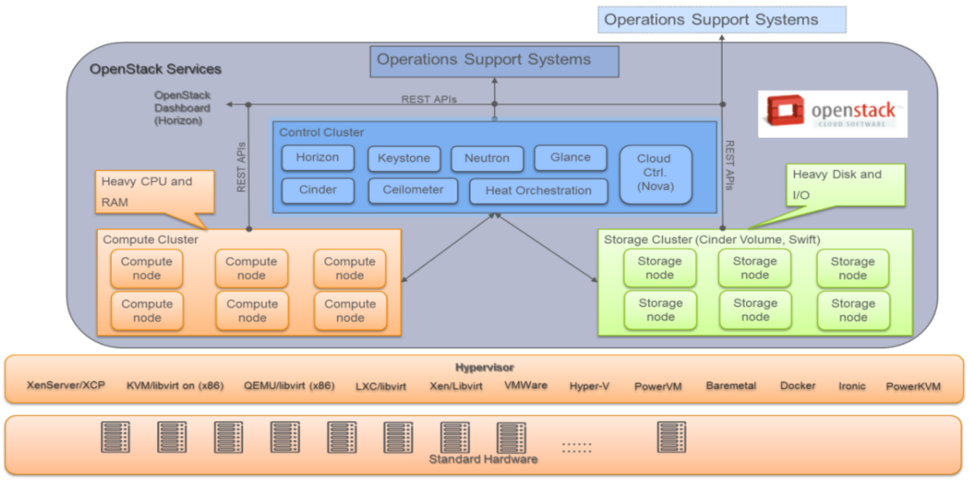

Figure 6: Open Stack cloud computing layers [11]

Open Stack simply follows the following design principles:

- Scalability & Elasticity
- Everything should be Asynchronous
- All required components should be horizontally scalable
- Always use shared nothing architecture (SN) or sharding
- Distribute everything (move logic to where state naturally exists)
- Accept eventual consistency and use it where it is appropriate
- Requires Test with submitted code

Along with the virtualization evolution, container based virtualization also started emerging. The first successful container from Linux foundation referred as “Linux Container” (LXC& LXD) [2] moved the virtualization to next step. LXC was the first successful container based virtualization that met the key manifesto addressing the isolation of only dependent software for execution. But was complex enough to orchestrate multiple layers within the containers. The challenge related to building, shipping and running these containers was addressed by Docker [3]. Docker is an extension of LXC. Other emerging containers are Rocket[14] and LXD [2] (command line tool for LXC)are being explored by open source projects during later 2014. The challenge related to orchestrating these LXC containers was addressed by google kubernetes [15]open source project. It’s becoming clear the next wave of container based virtualization is fast evolving.

Hypervisors emulate hardware and they are referred as Virtual Machines (VM). While they emulate hardware they also have the complete operating system running as part of virtual machines.

Containers are based on shared operating systems and specifically rest on single Linux instance (for running the Docker or LXC instances on windows machine, Linux instance could be an emulated instance using QEMU or windows supported hypervisors). They are much thinner than fat virtual machines. Fundamentally this means majority of VM size is reduced and container has only a neat capsule with small required dependent software for the application that we run in the container.

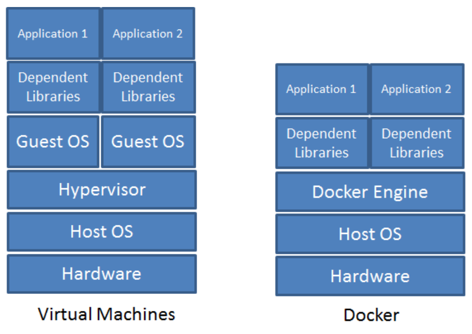

Figure 7: Difference between "Virtual Machine" and "Container based Virtualization" [16]

Guest Operating Systems are in GBs, though the Application could be few MBs. In case of LXC based Docker container would comprise Application and its dependencies. This key difference brings down the overall resource (memory, CPU) consumption of the hardware. Additionally Docker and LXD based utilities are available to retain the container process for longer time, spawn based on available resources or destroy once the task is done.

Adaptation of container brings in our key non-functional requirement "Able to run according to the availability of resources (example: Memory, CPU)".

Proposed Execution Technique For SON Use Cases

By combining the benefits of "Map-Reduce programming model" and "Container based virtualization" we are able to address all the identified non-functional requirements of SON functionalities and use cases as mentioned in Section 2.

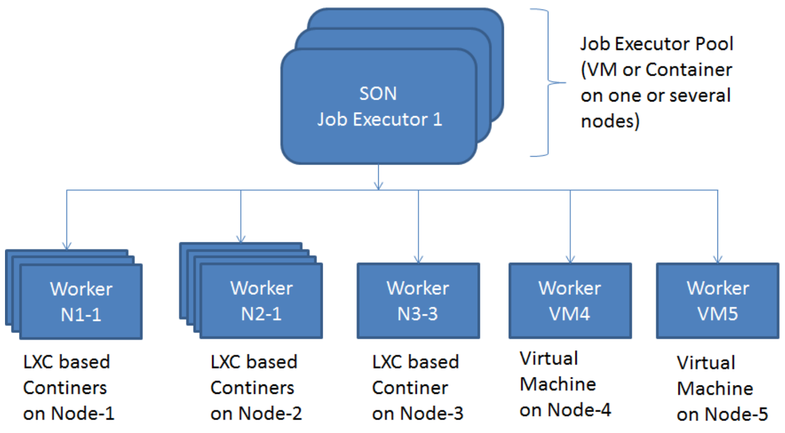
SON
Job Executor 1
Job Executor Pool (VM or Container on one or several nodes)
Worker N1-1
LXC based Continers on Node-1
Worker N2-1
LXC based Continers on Node-2
Worker N3-3
LXC based Continer on Node-3
Worker VM4
Virtual Machine on Node-4
Worker VM5
Virtual Machine on Node-5

Figure 8: Execution Technique For SON Use Cases

Execution technique proposed in Figure [8] has two key elements, “SON Job Executor” and container virtual instances (includes “LXC based Containers” or virtual machines).

Pool of SON Job Executor and Containers provide the platform for parallel task execution. Job executors are inspired from Map-Reduce[1] Master nodes and “Worker Nodes” Containers are inspired from Map-Reduce Worker nodes. Both Master nodes and Worker nodes are resilient to failures. The method proposed in Map-Reduce[1] for Master and Worker nodes are applicable for “SON Job Executor” and “Worker Nodes”.

SON Algorithms involves several granular steps in achieving a result for the SON use case. The key challenge is executing those steps in a stateless fashion where the model is complying with shared nothing architecture [10]. When we are able to design a SON algorithm in a stateless way, it becomes easy to run the algorithm in parallel on smaller worker threads. In our execution technique “Worker nodes” acts as smaller worker threads and “SON Job Executor” executes the algorithm either in parallel or fetches the required data from state full nodes. Some of the “Worker nodes” could be state full nodes for caching network centric configuration information (could be for unavoidable reasons until that part of the algorithms is able to evolve to stateless maturity).

Applying Proposed Execution Techniqueson SON Use Cases

Automatic Neighbour Relation (ANR) [5, 7 and 9]

Neighbour Relation plays a crucial role in mobility of user equipment between cellular cells. ANR addresses the key aspect of creating optimized neighbour relations for every cell in the cellular network. Neighbour relation could be Intra-Radio Access Technology (RAT) (eg: within WCDMA across frequencies) or Inter-Radio access technology (eg: between LTE and WCDMA). There are technology specific (LTE, WCDMA, GSM) restriction about how many neighbour relations each cell could potentially have for both Intra and Inter RAT relations. This leads to a situation where a generic ANR SON Algorithm could potentially create optimized number of neighbour relations based on key handover criteria’s and allowed number of relations. ANR is part of “Self Configuration” function.

Further in this section we discuss about applying Magnetic Field Model (MFM) [9, 17] based on ANR algorithm.

The Magnetic Field Model (MFM) [17] is inspired and derived from the basic behaviour and specification of magnets, in particular, based on their magnitude (magnetic field strength) and direction (magnetic poles). An electromagnet is a type of magnet in which the magnetic field is generated by the flow of Electric Current (EC). The strength of the magnetic field (B) is directly proportional to the strength of the EC (I). Thus, by increasing the magnitude of the electric current, them agnitude of the magnetic field increases and thereby influences the magnetic field for larger distances. In the MFM, each cell is considered as an electromagnet.

Detailed steps of the algorithms are as follows:

Step 1) Identify the cells for which ANR to be applied (could be subset of the cells within the network or the whole network) based on any specific or generic criteria.

Step 2) For each cell perform the steps as mentioned below:

a) Create cell-pair with other cells in the network.

b) Find MFM, Repulsion factor "α Tesla"and Attraction factor "βTesla" between the cell pairs based on distance and frequency attributes (frequency attributes are for Inter and Intra Neighbour relations).

Consideration of cell pair distance could bebased on basic user parameter (example: < 2 Km) (or)based on empirical path loss models[18] widely used in wireless network:

Calculate the distance between call-pairs and apply the distance with appropriate empirical propagation model and calculate the propagation loss. If the calculated propagation loss is within the acceptable range of propagation Repulsion factor α Tesla will be higher and Attraction factor β Tesla will lower proportionally. Value of α Tesla and β Tesla can be based on calculated propagation loss.

Okumura's Hata Urban Propagation Model [18]

This model is intended for large cells with base station being placed higher than the surrounding rooftops and meant for 150 to 1500 MHz.

$P_{L,\,urban}(d)$ dB = $69.55 + 2.16 \log_{10}(f_c)-13.82\log_{10}(h_t)-a(h_r)$
$+(44.9-6.55\log_{10}(h_t))\log_{10}(d)$

Here, f_c is the carrier frequency, h_t is the height of the transmitting (base station) antenna, h_r is the height of the receiving (mobile) antenna, and $a(h_r)$ is a correction factor for the mobile antenna height based on the size of the coverage area.

For ANR case assume $h_r$is neighbouring cell.

Cost 231 Extension to Hata Model [18]

The European Cooperative for Scientific and Technical (COST) research extended the Hata model to 2 GHz as follows:

$P_{L,\,urban}(d)$ dB = $69.55 + 2.16 \log_{10}(f_c)-13.82 \log_{10}(h_t) - a(h_r)$
$+(44.9 - 6.55 \log_{10}(h_t)) \log_{10}(d) + C_M$

C_M is 0 dB for medium sized cities and suburbs and is 3 dB for metropolitan areas. The remaining parameters are same as (i).

ERCEG Model [18]

Erceg model is based on 1.9GHz macro cells based on experimental data collected by AT&T. The terrains are classified in three categories. Category A is hilly terrain with moderate-to-heave tree density and has high path loss. Category B is hilly terrain with light tree density or flat terrain with moderate-to-heavy tree density and intermediate path loos. Category C is flat terrain with light tree density and has a low path loss.

The median path loss at distance $d > d_0$ is given by:

P_LdB = $20 \log_{10}(4\pi d_0/\lambda) + 10\gamma \log_{10}(d/d_0) + s$, for $d > d_0$

Here, λ is the wavelength in metres, γ is the path-loss exponent with:

$\gamma = a - bh_b + d/h_b$

h_b is the height of the base station in metres (between 10m and 80m), $d_0 = 100$ m, and a,b,c are constant dependent on the terrain category. These parameters are listed in the table below.

Table 1: Parameter Values Based on Terrain

Model Parameter	Terrain Type A	Terrain Type B	Terrain Type C
a	4.6	4	3.6
b	0.0075	0.0065	0.005
c	12.6	17.1	20

Model is valid for frequencies close to 2 GHz and for receive antenna heights close to 2 m.

For other frequencies and antenna heights (between 2 m and 10 m), the following correction terms are recommended

$PL_{modified} = PL + \Delta PL_f + \Delta PL_h$

Here, PL, is the path loss given earlier, ΔPL_f is the frequency term, and ΔPL_h is the receive antenna height correction terms given as follows:

$\Delta PL_f = 6 \log_{10}(f/2000)$

$\Delta PL_{h} = -10.8 \log_{10}(h/2)$ for Categories A and B

$\Delta PL_{h} = -20 \log_{10}(h/2)$ for Category C

For ANR distance "d" is the calculated distance between cell pairs.

a) Identify top neighbour relations based on α Tesla and β Tesla values by sorting them appropriately.

Applying the above steps on proposed execution technique for ANR

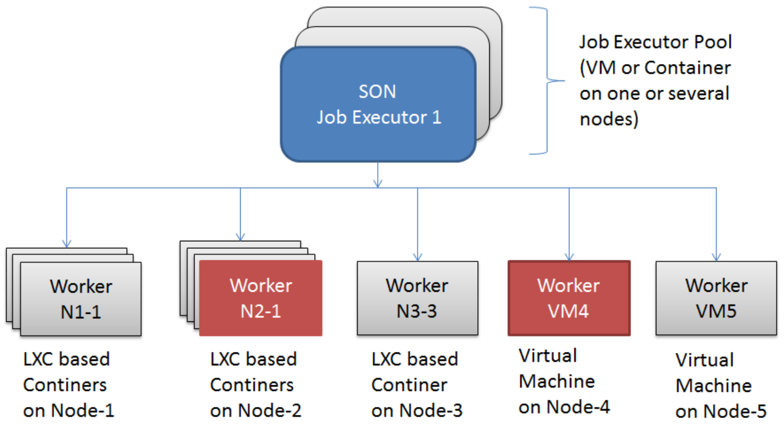

Figure 9:

1) Identifying the cells for which SON ANR use case to be applied. Since this is not more than a query or getting the list of available Cells, SON Job Executor could

delegate this task to one of the Worker Node and fetch the details or directly get the list of cells from one of the Virtual Machine Worker node that potentially has the information of the network topology always cached. In the above Figure 9 “SON Job Executor 1” use either “Worker N2-1” for fetching the details or uses “Worker VM4” for getting the list from the network topology cache.

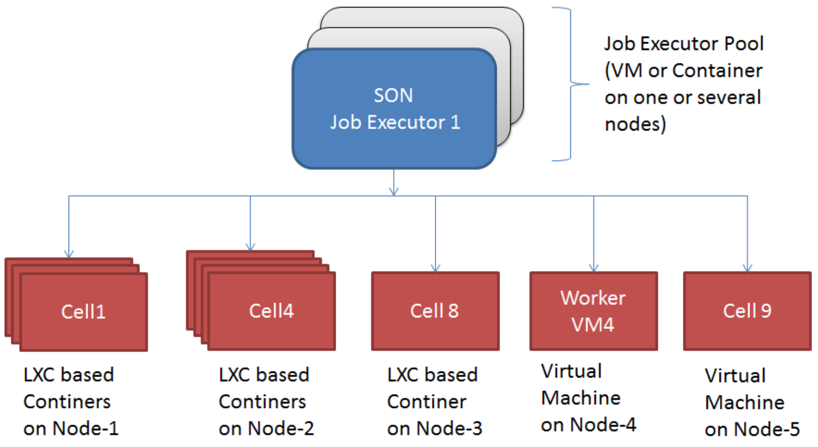

Figure 10:

2) SON Job Executor has information about available Worker Nodes. From the list of cells “SON Job Executor” delegates cell by cell to “Worker Nodes” for parallel execution of identifying neighbour relation. Once the worker node successfully completes all the sub-steps of step 2, provides the neighbour list back to “SON Job Executor”. As in Figure 10“SON Job Executor1” delegates cell by cell to all the containers and VMs on Node 1, Node 2, Node 3 and Node 5.

Note: Key advantage in such parallel execution is we are able to horizontally scale the configuration than traditional vertically scaled nodes. Also, additionally in case of failures the method is more resilient than vertically scaled node clusters. Also, the horizontally scaled nodes could be cheap commodity hardware rather than expensive server grade clusters.

3) SON Job Executor receives the “identified neighbour list” and delegates to another “Worker” for provisioning the same on the cell. This way the execution remains stateless and possibility of re-doing the sub step is highly possible for resilient cases.

4) If “Worker” does not respond or fails for “SON Job Executors” heart beat check, “SON Job Executor” re-delegates the same to another “Worker” container.

Neighbour List Optimization (NLO)

Optimization of neighbour list is part of “Self Optimization” function of SON. It almost follows the algorithm as specified in Section [4.1]. Only key difference is in sub-step of overall Step 2) where α Tesla and β Tesla values are derived based HO success ratio. At the end of overall step 2) neighbour lists are sorted based for α Tesla and β Tesla values for Inter and Intra NLO respectively.

Basic Uplink and Downlink Parameter Optimization

Physical Cell-ID (PCI) [17] and Physical Random access channel (PRACH) [17] are basic parameter configuration for LTE downlink and uplink parameters. MFM technique [17] can be applied with proposed execution technique for quick allocation.

Evaluations and Analysis

In this section we evaluate and analyse the effect of proposed execution technique pertaining to Execution time for different data sets and expressing them with Big-O notation. Also, evaluating and analysing the impact on Memory, Resilience behaviour of the proposed execution technique. Finally we are able to clearly conclude that there is definitely a trade-off between Execution time, memory and Resilience behaviour.

Execution Time Based on Rate of Growth

Asymptotic analysis of each step specified in the SON ANR algorithm used in Section [4.1] is as follows:

Table 2: Big-O Asymptotic Analysis of proposed SON ANR algorithm

Step	Total Time	Big O notation
Step 1) Identify cells for which neighbour relations to be defined.	A constant time “c” * “n” number of cells in the network. (cn)	O(n)
Step 2 a) Create cell-pair with other cells in the network.	A constant time “c” *number of cells in network “n” *number of other cells in the network “n-1” = “c” * “n” * (n-1) = $cn^2 - cn$	$O(n^2)$
Step 2 b) Find MFM α Tesla (Repulsion factor) and β Tesla (Attraction factor)	α Tesla = cn^2 β Tesla = cn^2	$O(2n^2)$
Step 2 c) Sort the identified neighbour relations based on α Tesla (Repulsion factor) and β Tesla (Attraction factor) values.	Bubble sort [22]: α Tesla = cn^2 β Tesla = cn^2	$O(2n^2)$
Step 2) Repeat Step 2 for each cell from the list of identified cells in Step 1.	Total time of Step 2 a) to Step 2 c) For sequential case: cells in network “n” *(2a + 2b + 2c) = “n” * (($cn^2 - cn$) + ($2cn^2$) + ($2cn^2$)) = $5cn^3 - cn^2$	$O(5n^3)$
Step 2) Repeat Step 2 for each cell from the list of identified cells in Step 1.	Total time of Step 2 a) to Step 2 c) For Parallel case (Proposed technique in this paper): (2a + 2b + 2c) = $5cn^2 - cn$	$O(5n^2)$

Let f(n) be the function which represents the given algorithm (SON ANR algorithm in sequential case).

$f(n)$ = Step 1 (n) + Step 2 ($5cn^3$- cn^2)

"a posteriori" for f(n) from the above table is:

$f(n) = 5cn^3 - cn^2+n$

Big-O Notation => $O(f(n)) = 5n^3$

Let p(n) be the function which represents the given algorithm (SON ANR algorithm in parallel case).

$p(n)$ = Step 1 (n) + Step 2 ($5cn^2$ - cn)

"a posteriori" for p(n) from the above table is:

$p(n) = 5cn^2 - cn+n$

Since the proposed architecture is primarily inspired from Map-Reduce programming model, let m(n) be the Map-Reduce asymptotic function. (Empty implementation in Map & Reduce interface for overhead calculation)

"a priori" of such MapReduce function is:

m(n) = n log n * total_nodes * (1/communication_time_between_nodes) where n is number of items. communication_time_between_nodes could be a basic ping as well.

Combining "a priori" m(n) with "a posteriori" p(n)which indicates proposed technique function:

pm(n) = n log n * total_nodes * (1/communication_time_between_nodes) + $5cn^2$- cn+n, where n is "number of cells" based on SON ANR algorithm, positive constant c such that

$0 \leq p(n) \leq c * O(pm(n))$

Big-O Notation => $O(pm(n)) = 5n^2$

Finally,Equation to compare isas follows:

$f(n) = 5cn^3 - cn^2+n$, where $0 \leq f(n) \leq c * O(f(n)) \rightarrow$ Equation 1

pm(n) = $5cn^2$ - cn+n +(nlog(n)*total_nodes*(1/communication_time_between_nodes)) $\rightarrow$ Equation 2

Equation 1 denotes the rate of growth when "n" (number of cell) increases.

Equation 2 denotes the rate of growth when "n" (number of cell) increases.

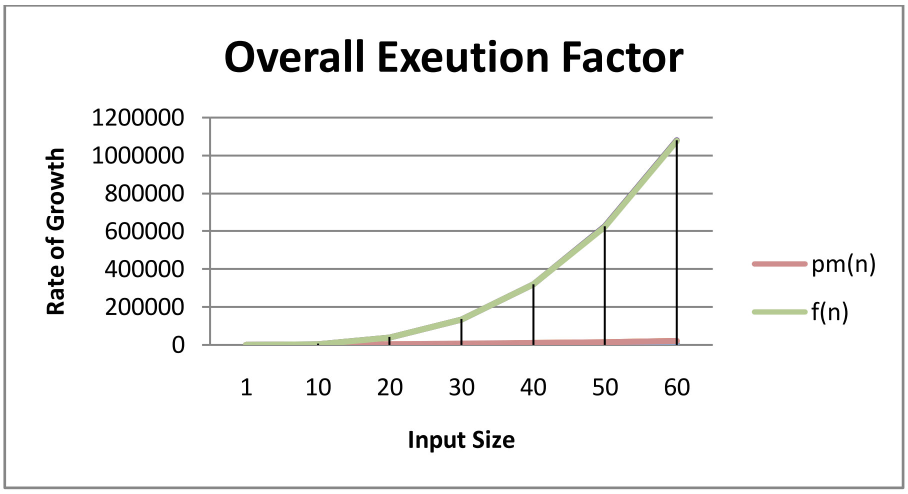

Figure 11: "Rate of Growth" of f(n) and pm(n) over the Input Size "n"

Table 3: Sample Data Representing "Rate of Growth"

n	n log n	log n	5n3	n2	5n2	Total Nodes	pm(n)	f(n)
1	0	0	5	1	5	15	5	5
10	10	1	5000	100	500	15	650	4910
20	26.0206	1.30103	40000	400	2000	15	2390	39620
30	44.31364	1.477121	135000	900	4500	15	5165	134130
40	64.0824	1.60206	320000	1600	8000	15	8961	318440
50	84.9485	1.69897	625000	2500	12500	15	13774	622550
60	106.6891	1.778151	1080000	3600	18000	15	19600	1076460

We could see when "n" (number of cells) increases for SON ANR pm(n) (proposed technique) performs significantly better than sequential execution f(n).

Resilience or Single Point of Failure
Map-Reduce programming model is very resilient compared to sequential execution model. In the proposed technique "SON Job Executor" is capable of re-spawning the worker containers in case of failures (due to communication or software un-responsiveness).

Memory
There is no significant difference in memory difference between sequential execution model and proposed technique. Proposed technique has memory overhead of container software that is significantly less when compared to vertically scaled systems (with dedicated operating systems). LXC based containers do consume memory overhead when number of LXC container increase. But, that overhead is significantly less when compared to the trade-off with "Execution time" and resilience nature of the proposed algorithm.

Conclusions
The proposed technique "Adopting Map-Reduce programming model with container based virtualization for Self-Organizing Networks" clearly shows significant advantage pertaining to "Execution Time" and "Resilience" (no single point of failure). Hence we could clearly see the key expectation of Centralized SON is met in the proposed technique. Further the paper could be extended for other Centralized SON use cases in detail.

Container based virtualization techniques helps in realizing the heterogeneous needs of the SON use cases with Map-Reduce programming model.

Appendix
The containers used for evaluating the effectiveness of the proposed technique are available at Docker repository[19] with container named "mfm" container. Use

"docker pull premnathkn/mfm" command to pull the repository into your local container.

The MFM codebase is available as open source in public repository [20, 21] for further reference.

Acknowledgements

The authors would like to thank Department of computer science, Karunya University for the facilities, guidance and support provided for performing research. Additionally authors would like to thank their work affiliations for encouraging employees to perform research that would benefit the "Information and Communication Technology" industry.

References

[1] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Google Inc., OSDI 2004, Paper available at http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[2] Linux Container, https://linuxcontainers.org/

[3] Docker, https://www.docker.com/

[4] Premnath, K N and Dr. Srinivas R, "Challenges in Self Organizing Networks for Wireless Telecommunications", ICRTIT 2011, IEEE Chennai, Paper available at http://ieeexplore.ieee.org/search/freesrchabstract.jsp?reload=true&navigation=no&arnumber=5972332

[5] 3GPP TR 36.902, V9.3.1, "Self-configuring and self-optimizing network (SON) use cases and solutions," December 2011.

[6] NGMN, "NGMN Releases Requirements on Self-Optimising Networks", January 2009. Available at http://www.ngmn.org/news/ngmnnews/newssingle2/article/ngmn-releases-requirements-on-self-optimising-networks-248.html

[7] NGMN, "NGMN Recommendation on SON and O&M Requirements", December 2008. Available at http://www.ngmn.org/uploads/media/NGMN_Recommendation_on_SON_and_O_M_Requirements.pdf

[8] SDN, "Software-Defined Networking(SDN) Definition", Available at https://www.opennetworking.org/sdn-resources/sdn-definition

[9] Premnath K N, Dr. Rajavelu Srinivasan and Dr. Elijah Blessing Rajsingh, "Magnetic Field Model (MFM) in Soft Computing and parallelization techniques for Self Organizing Networks (SON) in Telecommunications", October 2014, Available at http://www.igi-global.com/article/magnetic-field-model-mfm-in-soft-computing-and- parallelization-techniques-for-self-organizing-networks-son-in-telecommunications/118205

[10] Shared Nothing Architecture, "Definition at Sprinter", Available at http://link.springer.com/referenceworkentry/10.1007%2F978-0-387-39940-9_1512

[11] Open Stack, "Open source software for creating private and public clouds", Available athttp://www.openstack.org/

[12] Hypervisor, "A hypervisor or virtual machine monitor (VMM) is a piece of computer software, firmware or hardware that creates and runs virtual machines", Available at http://en.wikipedia.org/wiki/Hypervisor

[13] Cloud 101, "What IaaS, PaaS and SaaS companies do", Available at, http://venturebeat.com/2011/11/14/cloud-iaas-paas-saas/

[14] Rocket, "Core OS is building a container runtime, Rocket", Available at https://coreos.com/blog/rocket/

[15] Kubernetes, "Kubernetes is an open source system for managing containerized applications across multiple hosts", Available at https: //github.com/googlecloudplatform/kubernetes

[16] Virtualization Info, "Difference between VM and Container", Available at http://virtualization.info/en/

[17] Premnath K N, Pankajkumar Pradhan, Darshan R and Lars Christoph, "Self-configuration of basic LTE radio parameters using Magnetic Field Model", IWSON2 Paris, Available athttp://ieeexplore.ieee.org/ xpl/articleDetails.jsp?reload=true&navigation=no&arnumber=6328325

[18] Raj Jain, "Channle Models A Tutorial", Available at http://www.cse. wustl.edu/~jain/cse574-08/ftp/channel_model_tutorial.pdf

[19] MFM Container, "At Docker Hub", Available at https://hub.docker. com/u/premnathkn/mfm

[20] MFM Codebase, "At google code", Available at https://code.google. com/p/mfm-simulation/

[21] MFM Codebase, "At Git Hub", Available at https://github.com/ premnathkn/trunk.git

[22] Narasimha Karumanchi, "Data Structures and Algorithms Made Easy", ISBN 978-0-615-45981-3"