On Load Management in Service Oriented Networks

Ehsan Elahi*, Jason Barron*, Micheal Crotty*, Miguel Ponce de Leon*, Rashid Mijumbi*, Steven Davy*, Dimitri Staessens† and Sander Vrijders†

*Telecommunications Software & Systems Group
Waterford Institute of Technology, Waterford, Ireland
Email: eelahi, jbarron, mcrotty, miguelpdl, rmijumbi, sdavy@tssg.org
†iMines, Ghent University, Belgium
Email: dimitri.staessens, sander.vrijders@intec.ugent.be

Abstract—In traditional Service-Oriented Architecture (SOA), dedicated intermediate nodes called load balancers are usually deployed in data centers in order to balance the load among multiple instances of application services and to optimize the resource utilization. But on the other hand, the addition of these nodes increases the installation and operational cost of data centers. These load balancers distribute incoming flows to multiple outgoing ports usually by hashing them. The selection of outgoing ports is either on a round robin basis or based on some other heuristics e.g. queue length, feedback from neighbors etc. Such load balancing approaches do not consider getting live feedback from the service end and therefore are not able to dynamically change the amount of allocated resources. Moreover, in a Microservices-Architecture (MsA), the load on microservices is usually not considered by the front-end application while sending them the jobs. Although the lifetime of flows toward microservices is short, however, considering their actual load while allocating them a job may result in more optimal resource utilization.

In this paper, a distributed load management scheme is proposed for service oriented networks based on the current Internet architecture. In this scheme, lightweight interconnected management agents are used to decide the availability for a particular service instance and help in optimal distribution of the flows. The proposed scheme can also be applied in other emerging internetworking architectures such as RINA.

I. INTRODUCTION

In a typical Service-Oriented Architecture (SOA) [6], multiple instances of a server application are deployed on multiple nodes for scalability and availability. This also minimizes the impact of a node failure, thus making the system more reliable. Dedicated intermediate nodes called load balancers are usually deployed in front of these multiple instances of the application server nodes to distribute the load and to optimize the resource utilization. The role of load balancers is to make sure that each node gets more or less equal amount of traffic at any given time. Typically, in SOA, a single load balancer is deployed for an entire service ecosystem. However, deploying a single load balancer for all the services within a data center leads to a single point of failure. It can also become a potential bottleneck as all the traffic has to pass through it.

The next logical solution is to deploy a separate load balancer for each service in the data center. This way, the incoming traffic is classified on the basis of the required service and then sent to the respective load balancer which then schedules the flows according to a predefined load balancing scheme. While deploying a separate load balancers per service could solve the single point of failure and bottleneck issue for the entire service ecosystem, there may still be a single point of failure and bottleneck for multiple instances of a single service. In addition, deployment of multiple load balancers per service in order to avoid the single point of failure, adds more complexity to the system. This also makes the process of adding/registering/removing one or more instances of a particular service more complex and error prone. Furthermore, adding a new service requires the addition of a new load balancer which, over time, leads to dozens of load balancers in the data center thus making it difficult to orchestrate and maintain.

Moreover in a microservices environment, dozens of small components are working parallel in concert with each other and might have created dozens of internal network connections among each other. In some cases the algorithmic complexity of the load balancing scheme becomes more than that of an individual component in the MsA. Adding additional instances of one or more component and adding one or more load balancer per component could exponentially increase the number of simultaneous connections within a microservices environment as shown in Figure 1. This can be modeled as \( \frac{n(n-1)}{2} m^2 \). Where \( n \) is the number of services in the system and \( m \) is the number of instances per load balancer. As a result, the resource consumption by the load balancers in the microservice environment could significantly increase with the increase in microservice instances thus making scalability an open challenge.

In this paper, a distributed load management scheme which is equally applicable to both SOA and MsA is proposed. In our proposal, lightweight smart management agents (MA) are collocated with each service to monitor the current load and share the load with each other and a service level manager. These management agents are able to start and shutdown a service according to the current service demands. Each MA maintains a list of service agents/components (SA) so that
Whenever a new service request comes in, it can be forwarded to the most suitable SA. The proposed scheme is not only applicable in the current Internet architecture but can also be used in other emerging internetworking architectures such as RINA [1]. RINAs communication model is based upon distributed Inter-Process Communication (IPC). RINA sees networking as an extension to local IPC and argues that networking is IPC and only IPC [2]. As explained in section II, RINA offers a structured approach to the service oriented architecture and it’s IPC model is not unlike the microservices architecture. Therefore in this paper, a RINA based data center network is used as a use case for experiments and analysis of the proposed scheme. Simulations are conducted in RINASim [3] which is an omnet++ based open source simulator for the RINA architecture.

The rest of the paper is organized as follows: Section II describes some of the background about the load balancing and load distribution. Section III presents some of the latest research contributions on load balancing in SOA as well as MsA, Section IV presents the proposed distributed load management scheme, while Section V discusses the simulation setup and analysis of results. Section VI gives concluding remarks and future directions.

II. BACKGROUND

A. Load Balancing vs Load Distribution

Optimal resource utilization and load management among hundreds of thousands of servers and their mirrors in one or more data centers located at different geographical locations is imperative to minimize the energy utilization, response time and operational costs.

It is worth noting that load balancing is architecturally different from load distribution. In load balancing, the intermediate node or the load balancer has to keep track of the status of all flows which are being forwarded from this node. The intermediate node performing the load balancing could be a single point of failure and a potential bottleneck.

Load distribution is the redirection of flows towards different paths and instances of the same server application. This has the advantage of there being no need for dedicated hardware nor a need to maintain the states of the flows at intermediate nodes. In general, load distribution is achieved by either the distribution of flows across multiple paths towards the same destination and/or distribution of flows towards multiple instances of the same application running on multiple server machines. The load can be distributed on the basis of geographical location of the (client and server) applications or many other parameters such as; the current load on a server, or data and flow requirements of the client application.

In spite of architectural differences, the goal of both, load balancing and load distribution is to ensure service availability and to minimize the response time. Examples of load distribution include MPTCP [12], ECMP [13] [14], load distribution using DNS [15], and the use of different server mirrors e.g. Ubuntu mirrors. Deployment of an intermediate node at the entry point (POP) of a data center that can distribute traffic evenly (e.g. load balancing using reverse proxy [16]) is a typical example of load balancing.

B. SOA and MsA

In software engineering the service-orientation principle has been advocated for many years, where the logic required to solve a large problem can be better constructed, carried out, and managed, if it is decomposed into a collection of smaller and related pieces, each of which addresses a concern or a specific part of the problem. Service-Oriented Architecture (SOA) encourages individual units of logic to exist autonomously yet not isolated from each other. Within SOA, these units are known as services [6].

On the other hand, the MsA style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms. These services are built around business capabilities and are independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies [8].

Conventional monolithic web applications usually consist of a core, which represents the business logic and all functionalities. Often, this core is logically divided into modules or other structures, but it constitutes one distinct application and is deployed as one. This core has several adapters, enabling the communication to the outside, like databases or user interfaces. Such monoliths are widely spread, since they can be easily developed, tested and deployed. With the help of a load balancer they can even scale. However, in recent years some critical downsides have become clear:

- With more and more features, a monolithic application becomes huge, often including many million lines of code. Hence, as the complexity increases the implementation of new features becomes more difficult.
- Continuous integration and deployment become impractical, since build and start-up time increases drastically with the size of the application.
- Scalability becomes difficult when different modules of the monolith have different hardware requirements.
- Any failure within any module of the application may lead to a failure of the entire application.
- Migration to new technologies and frameworks is extremely difficult [9].

In order to overcome the obstacles of monolithic applications the Microservices architectural style has evolved. Certain
principles, initially defined by Martin Fowler in [8] may be followed when applying that style.

C. Recursive Internetworking Architecture (RINA)

RINA is aimed at providing configurable, secure, resilient, predictable and flexible network services. RINA’s communication model is based upon distributed Inter-Process Communication (IPC). Two applications on the same host can communicate with each other through a local IPC facility provided by the operating system. The same IPC facility can also be extended to allow two applications to communicate on different hosts in the same or different networks. This extended IPC facility is termed as Distributed IPC Facility (DIF) [1]. Unlike the five-layer model of the Internet, in RINA, a DIF can be seen as a single type of layer which can be recursively repeated as many times as required, providing the same functions/mechanisms but tuned under different policies to operate over different ranges of the performance space (e.g. capacity, delay, loss).

As can be seen in Figure 2, the IPC processes at the same level across the network form a DIF. Each DIF has it’s own set of properties and access control mechanisms. The IPC processes and application processes at the upper layers registers with the lower level IPC processes in order to communicate with other IPC processes and applications. There are consistent APIs to interface with the IPC processes. In this way each layer can be repeated recursively as many times as required.

RINA offers structured mechanisms for QoS, mobility management, resiliency and security. Each flow within the network has an associated set of QoS properties called QoS cubes. If the available DIF is unable to support the QoS requirements of a certain flow, a new DIF layer can be created, end-to-end, to meet the flow requirements. Thus, RINA has been envisioned as the future of service and policy oriented networks. RINA allows monolith applications to be broken down into independent micro services where these microservices can communicate with each other on dedicated DIFs. The simple and elegant architecture minimizes the number of network protocols, thus reducing the complexity and facilitating the network management. Therefore, RINA is used as the use case scenario for the implementation and experimentation of the load distribution scheme being discussed in this paper.

III. RELATED WORK

The dissection of load balancing and distribution frameworks exposes three major components:

A. Admission/Scheduling/Forwarding Algorithms

In order to determine the server machine to which to forward a request, the load balancers use a variety of scheduling and forwarding algorithms. A survey and comparative analysis of such algorithms has been presented in [18] and [19]. The selection of an algorithm is mainly dependent on it’s algorithmic complexity, and scalability. However, service provider’s policy also plays an important role in the selection of an algorithm suitable for it’s own business. The comparison and discussion of load balancing algorithms is out of scope of this paper. Load balancing and distribution in MsA is discussed recently and presented in [20], [21] and [31]. However, due to certain constraints such as complexity, asynchronicity, resiliency, and elasticity as discussed in [10] and [11]; it is not that simple to get benefits from MsA by mere deploying an off-the-shelf load distribution technique for all kinds of microservices. For each set of microservice, a fully customized technique and algorithm would be required. On the other hand, RINA is inherently resilient to failures and therefore less complex in nature. This means it is easy to customize the service deployment in RINA with a higher level of controllability.

B. Load balancer deployment based on either software or hardware

Load balancing can be centrally controlled or distributed. Typically a dedicated node called load balancer is deployed to distribute the incoming traffic among multiple available server machines however distributed load balancing techniques can also be seen widely in the literature. Table I presents a comparison of different load balancing techniques with the proposed RINA based load balancing on the basis of given self-explained parameters.

C. Provision for relocation/migration of VMs

One of the main objectives of service providers is to reduce energy consumption and to optimize resource utilization. Migrating a VM from one location to another or turning a VM off can save significant amount of energy. Boutaba et al. [17] describe how VM migration could work and the challenges associated with it. The question of when to migrate or turn off a VM is still unanswered. We advocate that by coupling this problem with the load balancing mechanism can answer this question. To the best of our knowledge, there is no load balancing or distributing technique that suggests when to migrate a VM or turn it off or on.
Distributed Load Management in RINA

RINA provides “policy” hooks (which are standardized) and has implicit support for multi-homing, security and QoS. Ideally, load balancing in RINA based data centers should not need to have special hardware or dedicated nodes, and as such with an efficient load distribution mechanism can abandon the need for load balancing, thus minimizing the installation and operational cost of the data centers.

The load management in RINA based data centers is more customizable and simple. Both categories of load management i.e. balancing and distribution can be achieved in RINA. The diagram in Figure 3 outlines how load balancing could work in RINA.

There is no single best solution for all types of workloads and flow characteristics and there are multiple ways of achieving it. Each comes with a set of pros and cons and suitable for some special use case scenarios. In this research, the most generic mechanism for load distribution is presented and will eventually result in balancing the load across the multiple paths as well as among the multiple instances of the server application process.

As shown in Figure 3, consider a data center with n instances of the same server Application Process (AP). In RINA, the set of these application instances is called a Distributed Application Facility (DAF), and each member of the DAF set is known as Distributed Application Process (DAP). In this paper, we refer to the set of server application instances as Server-DAF, and also define a distributed management system called as Application Management DAF Management System (AM-DMS) with multiple management agents which are collocated with each DAP. The server instances are managed by this AM-DMS. Load information would be exchanged among some or all members of the DAF and the agents in AM-DMS. The Server-DAF will be on a private/restricted DIF called Tenant-DIF. Each DAP will also be on a Public-DIF through which client APs can get connected with server APs. The DAPs share their load statistics with each other as well as with the agents of AM-DMS and unanimously create a list of instance names, belonging to the same AP, which are available to accept new flow requests. Each DAP shares this list with the IPC process in the Public-DIF. Whenever a client AP sends a request for DIF name resolution to its DIF Allocator (DA), it responds with the Public-DIF name where one of the server AP instance is available. This server instance also knows the load status of other server instances within the DAF and thus has the forwarding table based on the latest information. After joining or extending the Public-DIF, the client AP sends a flow allocation request to the IPC process within that DIF. Now the IPC process forwards the request towards the selected server AP instance from the list. At this point there are important options that need to be discussed:

A. Pseudo Random Load Distribution

The AM-DMS runs a DAF wide load sharing policy to make a list of server AP instances which are available to accept new flows at that point of time, irrespective of their capabilities. Whenever a new flow request comes in, the IPC process in the Public-DIF selects one instance name and forwards that request to that instance. This is a case of unicast. In the case of broadcast, the IPC selects all the members from the list to forward the flow allocation request. In the case of multicast, the IPC selects the multicast members from the list to forward the flow allocation request.

After looking at the QoS requirements in the allocate request of the client AP, the IPC processes shortlists those server AP instances which can meet those requirements and then selects the instance name randomly, on round-robin basis, geographically nearest or the one at top of the list.

The limitation of this approach is that the IPC processes in the Public-DIF do not know about the current state of the server instance. Each server application instance has a limited service rate. The IPC processes within the Public-DIF are not aware of this limit. It may happen that IPC processes forward flow allocation requests to a specific server AP instance at a rate larger than its service rate, causing potential overrun. This

### Table I: Load Balancing and Distribution Mechanisms

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Distributed</th>
<th>Scalable</th>
<th>Location</th>
<th>Consider</th>
<th>Server</th>
<th>Hashes</th>
<th>Proactively</th>
<th>Robust to</th>
<th>Decides on</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECMP [14]</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>DNS Based RR [15]</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>LB-Named [22]</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Peer-Flow VLB [23]</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>HEDERA [24]</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Google’s B4 [25]</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CONGA [26]</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>VD-LB [27]</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DRILL [28]</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Presto [29]</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Syn-Race [30]</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>RINA Based LB</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>
may be due to a burst of requests coming in at a certain point of time. So until the Server-DAF updates the list of available server instances, one or more instances could become flooded.

This limitation is mitigated by making application instances capable of redirecting the flow requests to the more suitable instance. In this way, whenever a certain AP instance, which is not currently accepting new requests, receives a flow request it will redirect that request to another instance which it assumes might be available. This process continues until a suitable AP instance can be found to accept the flow request.

### B. Geographical Load Distribution (Minimum Hop Selection)

The IPC processes within the public DIF not only know about the instances of the AP which are currently accepting new flows but also know about the number of physical hops these instances are located on. In this way, the flow requests coming from the client APs could be forwarded to the server AP instance which is located at the minimum number of hops distance. The possibility that the IPC processes in the public-DIF do not know about the very current state of the server instances cannot be ruled out in this option. Therefore, the server instance has to be able to forward/redirect the flow requests to other server AP instances.

The next section presents the simulation setup and discusses the results comparing both the load distribution options mentioned above.

### V. SIMULATION AND RESULTS

Both the proposed load distribution approaches were implemented in the RINASim [3] which is an open source OMNET++ based implementation of RINA. The load distribution implementation is available in the TSSG-LB branch [5] of the RINASim [4] implementation. The main load distribution algorithm can be seen in stats.cc and stats.h files. In this experiment we created a fat tree like data center topology with 4 PODs and 8 server machines per POD thus 32 total server machines as shown in Figure 4. There are 30 clients having random connection subscription between 1Mbps to 2Gbps. A continuous stream type of traffic is generated between client and server APs. Each client generates a stream of PDUs for 5 seconds at random intervals between 1-100 msec. Each experiment is repeated 10 times and the average of each traced value is used for result and analysis purposes.

In the first experiment, as can be seen in the Figure 5, each server machine is set to process 22,000 PDUs per second. Beyond this limit the next PDUs will be forwarded to the next AP instance. The result shows that 6 out of 32 servers run at their full limits while 26 servers remained idle causing under utilization of resources. On the other hand when we configure our load balancing algorithm in such a way that when one server is running at 1/4th of its capacity or beyond that, then the next incoming load is diverted to the next available server. The result in Figure 5 shows an even distribution of load across all the available resources.

In the next experiment, the two load distribution approaches (Random Load Distribution and Min-hop Selection) discussed in Section IV are compared to each other. Figure 6 shows that flows experienced more average delay in the random selection method than the min-hop selection method. This is very obvious that in random selection, the load balancing algorithm forwards the flow request to any available server instance irrespective of how far it is located from the client. Each client has its own connection speed, generates PDUs at random intervals therefore each client would experience different delays. However it is apparent that min-hop selection method offers less average delays than the random selection method. The result shows that 6 out of 32 servers run at their full limits while 26 servers remained idle causing under utilization of resources. On the other hand when we configure our load balancing algorithm in such a way that when one server is running at 1/4th of its capacity or beyond that, then the next incoming load is diverted to the next available server. The result in Figure 5 shows an even distribution of load across all the available resources.

In the next experiment, the two load distribution approaches (Random Load Distribution and Min-hop Selection) discussed in Section IV are compared to each other. Figure 6 shows that flows experienced more average delay in the random selection method than the min-hop selection method. This is very obvious that in random selection, the load balancing algorithm forwards the flow request to any available server instance irrespective of how far it is located from the client. Each client has its own connection speed, generates PDUs at random intervals therefore each client would experience different delays. However it is apparent that min-hop selection method offers less average delays than the random selection method.
method.

Figure 7 shows the average throughput gained by each client while using both methods respectively. In random selection method, the average throughput for each flow merely exceeds 200Mbps while with the use of min-hop selection method, the same flow with same properties, can achieve near 2Gbps of throughput.

With random selection method, the selection algorithm picks one server instance randomly from a list of available server instances without bothering of how far the selected instance is located from the client instance. As a result, this method produces more delays, more packet drop ratio and less throughput as shown in Figure 8.

VI. CONCLUSION AND FUTURE WORK

From users to large scale ISPs, data center (DC) and cloud providers want a more adaptable, configurable, flexible, and resilient network on which to build different services. RINA is the answer to these questions.

Load distribution scheme presented for RINA can distribute the workload of server instances. It is also capable of evenly balancing the network load. The key point lies in the management agents collocated with server instances which decides the availability of server instances. Poorly selected server instances could cause adverse effects on the network load and congestion could occur in the network. Therefore, the selection of server instances should be as smart as possible. There are countless selection methods available in order to select one server instance out of a list. In this paper, two methods, random selection and min-hop selection are presented. It is shown that min-hop selection method outperform the random selection one.

The future plan is to compare some other methods e.g. QoS requirements based server instance selection, server machine capabilities based selection etc. A modified load distribution algorithm will also be used in future experiments in which the scheme will make one server instance as root and forward the flow request to the root. The root will forward/redirect to its leaves hierarchically until the flow request found a more suitable instance.

REFERENCES

[13] IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridge Local Area Networks Amendment: Equal Cost Multiple Paths (ECMP), 802.1Qbp