ISBN 978–987–633–574–4

Service Proxy with Load Balancing and Autoscaling for a Distributed Virtualization System

Pablo Pessolani, Marcelo Taborda and Franco Perino

Department of Information Systems Engineering
Universidad Tecnológica Nacional – Facultad Regional Santa Fe
Santa Fe, Argentina
{ppessolani, mtaborda, fperino}@frsf.utn.edu.ar

Abstract. Cloud applications are usually composed by a set of components (microservices) that may be located in different virtual and/or physical computers. To achieve the desired level of performance, availability, scalability, and robustness in this kind of system is necessary to describe and maintain a complex set of infrastructure configurations.

Another approach would be to use a Distributed Virtualization System (DVS) that provides a transparent mechanism that each component could use to communicate with others, regardless of their location and thus, avoiding the potential problems and complexity added by their distributed execution. This communication mechanism already has useful features for developing distributed applications with replication support for high availability and performance requirements.

When a cluster of backend servers runs the same set of services for a lot of clients, it needs to present a single entry-point for them. In general, an application proxy is used to meet this requirement with auto-scaling and load balancing features added. Autoscaling is the mechanism that dynamically monitors the load of the cluster nodes and creates new server instances when the load is greater than the threshold of highest CPU usage or it removes server instances when the load is less than the threshold of lowest CPU usage. Load balancing is another related mechanism that distributes the load among server instances to avoid that some instances are saturated and others unloaded. Both mechanisms help to provide better performance and availability of critical services.

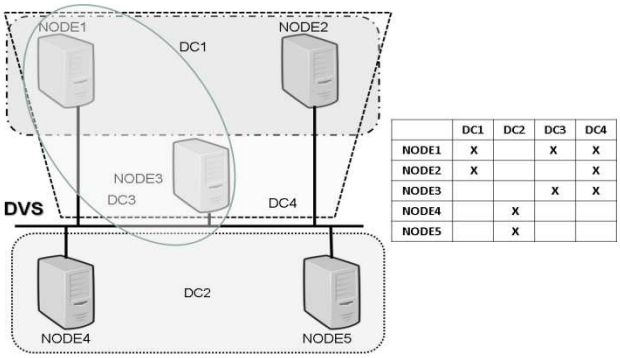
This article describes the design, implementation, and testing of a service proxy with auto-scaling and load balancing features in a DVS.

Keywords: Autoscaling, Load Balancing, Distributed Systems.

1 Introduction

Nowadays, applications developed for the cloud demand more and more resources, which cannot be provided by a single computer. To increase their computing and storage power, as well as to provide high availability and robustness they run in a

distributed environment. Using a distributed system, the computing and storage capabilities could be extended to several different physical machines (nodes). Although there are various distributed processing technologies, those that offer simpler ways of implementation, operation, and maintenance are highly valued. Also, technologies that provide a Single System Image (SSI) are really useful because they abstract the users and programmers from issues such as the location of processes, the use of internal IP addresses, TCP/UDP ports, etc., and more importantly, because they hide failures by using replication mechanisms. A Distributed Virtualization System (DVS) is an SSI technology that has all these features [1]. A DVS offers distributed virtual runtime environments in which multiple isolated applications can be executed. The resources available to the DVS are scattered in several nodes of a cluster, but it offers aggregation capabilities (allows multiple nodes of a cluster to be used by the same application), and partitioning (allows multiple components of different applications to be executed in the same node) simultaneously. Each distributed application runs within an isolated domain or execution context called a Distributed Container (DC). Fig. 1 shows an example of a topological diagram of a DVS cluster.



	DC1	DC2	DC3	DC4
NODE1	X		X	X
NODE2	X			X
NODE3			X	X
NODE4		X		
NODE5		X		

Fig. 1. Illustration of a DVS topology

A problem that must be considered when using a distributed application refers to the location of a certain service used by an external or internal client, or by another component of the distributed application itself. One way to solve this problem would be to use existing Internet protocols. With the DNS protocol, the IP address of the server can be located in the IP network, and with ARP the MAC address of the server can be located within a LAN. However, one issue that must be taken into account when working with a cluster is that the network and its nodes may fail, preventing continuity in the delivery of a given service.

When a cluster of backend servers runs the same set of applications for a lot of clients, it needs to present a single entry-point for their services. In general, an Application Proxy (AP), also known as Reverse Proxy, is used to meet this requirement with auto-scaling and load balancing features added. Autoscaling is the mechanism that dynamically monitors the load of the backend servers and creates new server instances when the load is greater than a threshold of highest CPU usage (*high_CPU*) or it removes server instances when the load is lower than a threshold of lowest CPU us-

ISBN 978–987–633–574–4

age (*low_CPU*). Load-balancing is another related mechanism that distributes the load among backend server instances to avoid that some instances are saturated and others unloaded. Both mechanisms help to provide better performance and availability of critical services. This technology is widely used in certain scenarios such as web applications, where end-users send requests from their devices as clients, and the SP is the component that establishes sessions with backend servers, thus distributing, balancing, and orchestrates services between internal services and microservices [2]. To distinguish between these two usages scenarios, in this article Application Proxy (AP) refers to the former, and Service Proxy (SP) refers to the latter.

This article presents the design, implementation, and experimentally proves the capabilities of an SP with autoscaling and load balancing features for a DVS. Therefore, the project focused on building an operational prototype of an SP for the DVS (not a commercial-class one), relegating performance and high availability improvements for future works.

The rest of the article is organized as follows: Section 2 refers to related works. Section 3 provides an overview of background technologies and Section 4 describes the design and implementation of the SP for the DVS (referred to as DVS-SP). Section 5 presents the tests for the SP performance evaluation and finally, the conclusions and future works are summarized in Section 6.

2 Related Works

APs are not unknown by the scientific community, so a lot of research and development works have previously been carried out, but for IP environments. Therefore, only those with the most important features and more popular [3,4] are presented here for space reasons.

2.1 NGINX

A very popular HTTP server and reverse proxy is NGINX [5]. It is free, open-source, and well known for its high performance, stability, rich feature set, and low resource consumption.

NGINX can handle tens of thousands of concurrent connections and provides caching when using the *ngx_http_proxy_module* module and supports load balancing and fault tolerance. The *ngx_http_upstream_module* module allows for de *nginx groups* of backend servers to distribute the requests coming from clients.

2.2 HAproxy

HAproxy [6] (stands for High Availability Proxy) is an HTTP reverse-proxy. It is a free, open-source, reliable, high-performance load balancer and proxying software for TCP and HTTP-based applications. It is also an SSL/TLS tunnel terminator, initiator, and off-loader, and provides HTTP compression and protection against DDoS. It can

handle tens of thousands of concurrent connections by its event-driven, non-blocking engine.
HAproxy was designed for high availability, load balancing and provides redirection, server protection, logging, statistics, and other important features for large-scale distributed computing systems.

3 Background Technologies

This section presents the products and tools that have been studied and analyzed as technological support for the design and implementation of the DVS-SP prototype.

3.1 M3-IPC

The DVS provides programmers with an advanced IPC mechanism named M3-IPC [7] in its Distributed Virtualization Kernel (DVK) which is available at all nodes of the DVS cluster. M3-IPC provides tools to carry out transparent communication between processes located at the same (local) node or in other (remote) nodes. To send messages and data between processes of different nodes, M3-IPC uses Communications Proxies (CPs) processes. CPs act as communication pipes between pairs of nodes.

M3-IPC processes are identified by *endpoints* that are not related to the location of each process, and then it does not change after a process migration. This feature becomes an important property that facilitates application programming, deployment, and operation. An *endpoint* can be allocated by a process or by a thread and must be unique in each DC.

M3-IPC supports message transfers (which have a fixed size) and blocks of data between endpoints. If the sender and receiver endpoints are located in the same node, the kernel copies the messages/data between the processes/threads which own the endpoints. If the sender and receiver are located in different nodes, CPs are used to transfer messages and data between nodes, and the DVK of both nodes copies those messages/data between the CPs and the processes/threads.

3.2 Group Communication System (GCS)

To exchange information between a group of processes that run on several nodes or in the cloud, communication mechanisms with characteristics such as reliability, fault tolerance, and high performance are required. Several tools offer these features such as Zookeeper [8], Raft [9] or the Spread Toolkit [10].
The Spread Toolkit was chosen for the DVS-SP development because it is a well-known GCS used by the authors' research group in other projects. On Spread Toolkit, two kinds of messages are distinguished. Regular messages: sent by a group member, and; Membership messages: sent by the Spread agent running on each node.

Regular messages can be sent by members using the provided APIs for broadcast (multicast) them to a group. However, unicast messages could be sent to a particular

ISBN 978–987–633–574–4

member. Membership messages are sent by Spread to notify members about a membership change, such as the joint of a new member, the disconnection of a member, or a network change. Network changes can be the crash of a node (or a set of nodes), a network partition, or a network merge after a partition.

Spread provides reliable delivery of messages (even in the event of network or group member failures) and the detection of failures of members, or the network. It also supports different types of ordering in message delivery such as FIFO, Causal, Atomic, etc. making it an extremely flexible tool for the development of reliable distributed systems.

Spread Toolkit is based on the group membership model called Extended Virtual Synchrony (EVS) [11], tolerating network partition failures and network merge, node failures, process failures and restart.

4 Design and Implementation of the DVS-SP

The design of the DVS-SP started proposing its architecture, describing its components and the relations between them. The active components are (Fig. 2):

- The Main Service Proxy (MSP) reads a configuration file, initializes all data structures, and starts the other components threads.
- For each Frontend Client Node (specified in the configuration file), a pair threads are started for the CPs. The Client Sender Proxy (CSP) thread sends messages from the DVS-SP to the Client node. The Client Receiver Proxy (CRP) thread receives messages from the Client node.
- Similarly, for each Backend Server Node, a pair threads are started for the CPs. The Server Sender Proxy (SSP) thread sends messages from the DVS-SP to the Server node. The Server Receiver Proxy (SRP) thread receives messages from the Server node.
- A Load Balancer Monitor (LBM) thread receives notifications about changes in the load state from the Load Balancer Agents (LBA) running on each backend server node.
- Each Client and Server node uses the Node Sender Proxy (NSP) and the Node Receiver Proxy (NRP) to communicate with the DVS-SP proxies.

Proxy messages differ from application messages. Proxy messages are the transport of single application messages (like a tunnel), a batch of application messages, a block of raw data, an acknowledgment message, or a proxy HELLO message. The reader should consider that this architecture was not designed to serve user applications such as web browsers as clients. It should be used among application services, i.e. web servers (Clients) which need to read/write files from/to network filesystems (Servers).

4.1 Main Service Proxy (MSP)

As was mentioned earlier, the MSP reads the configuration file which describes the cluster, initializes all data structures, and starts the other components threads.
In the configuration file four types of items are specified:

a. *MSP*: specifies the node name where the MSP runs, the node ID, and the high-water and low-water load levels.
b. *Server*: specifies the server node name and its node ID.
c. *Client*: specifies the client node name and its node ID.
d. *Services*: describes the name of the service (i.e. fileserver), the external endpoint (*ext_ep*) in which the SP will receive requests from clients, the lowest (*low_ep*) and highest (*high_ep*) endpoint numbers which servers could use to serve the requests, and eventually the pathname of a server program to run on a server node.

Services could be running on server nodes (persistent service) or they could be started when the MSP receives a new request from a client (ephemeral service). Therefore, the MSP creates a *Session* for each pair of client-server processes. Once the MSP detects that a new client requests the same service on the same node using the same pair of endpoints, it removes the old *Session* from its database and terminates the old server process. Afterward, it creates a new *Session* for the new pair of processes. This behavior is a piece of the auto-scaling mechanism of the DVS-SP. A session is defined by the following tuple: {*dcid, clt_ep; clt_node, clt_PID, svr_ep, svr_node, svr_PID*}.

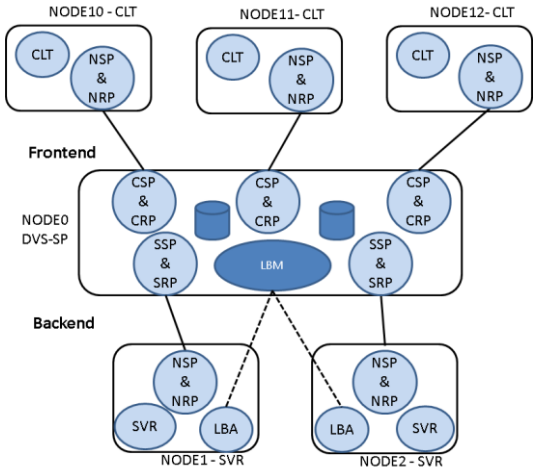


Fig. 2. DVS-SP Architecture.

As the LBM manages the load database of all Servers, the DVS-SP scheduling module can decide when it is time to allocate another Server node for new sessions or when it is time to dismiss it.

4.2 Client Receiver Proxy (CRP) and Server Sender Proxy (SSP)

When a new session starts, the client sends a message to the external endpoint (*ext_ep*) of the DVS-SP through its NSP to the CRP. The CRP compares several fields of the sessions to find an active session that matches. If it does not exist, it searches the server's database for the first non-saturated server node (server load < *high-water*) and allocates it for that session. The reader might ask: why not choose the server with the least load?. That policy would go against Autoscaling because an unloaded server quickly could acquire more work and could not be removed from the cluster (scaling-down).

If a program is specified for that service, the CRP sends a remote command to the server's node to execute the server program. Then, the proxy message is forwarded to the server NRP.

4.3 Server Receiver Proxy (SRP) and Client Sender Proxy (CSP)

When an SRP receives a message from its server's NSP it checks if a session exists. If all the session's parameters match but the server's PID, it removes it as an expired session. If a program was specified for that service, it sends a remote command to the server's node to terminate the server process. If the server's PID also matches, it gets the client endpoint field of the session and queues the proxy message into the CSP message queue. As the CSP is waiting for messages in its queue, it forwards the message to its Client NRP.

Several endpoint conversions are done into the header of proxy messages on CRP and SRP to hide the real architecture from clients and servers. Clients only request the DVS-SP as their single server, and servers only reply to the DVS-SP as their single client (service proxy behavior).

4.4 Load Balancer Monitor (LBM)

The LBM collects information about the load level of server nodes. The Load Balancer Agents (LBA), report their node load levels when they change. The load levels are defined as *LVL_UNLOADED, LVL_LOADED, and LVL_SATURATED.*
The LBM manages and keeps updated the node status database used by CRPs to allocate servers for new sessions. When a server node fails (reported by the GCS), the LBM deletes all the sessions with that node. When the load level of all active servers is *LVL_SATURATED* during a specified *START_PERIOD*, the LBM commands the hypervisor to start a new node (scaling-up). If a server node has no active sessions during a specified *SHUTDOWN_PERIOD*, it will be shut down (scaling-down).

4.5 Load Balancer Agents (LBA)

LBA periodically evaluates the load of its node. Currently, the load of a node is defined as the mean of CPU usage (reported by the pseudo-file */proc/stat*) in the specified *LBA_PERIOD* period. Although there are additional metrics that could be con-

sidered to describe the load of a node [12], such as memory usage, network traffic, disk I/O, etc., only CPU usage was considered to simplicity the prototype implementation.

Each server node keeps a *load_lvl* variable that stores its load level. In each period, if the new load level value differs from *load_lvl*, the LBA reports this new level to the LBM using the GCS, and updates *load_lvl*. Therefore, the dissemination of load information is event-driven. This mechanism consumes lower network bandwidth than a periodic one [13]. In the case that the LBM is doesn´t alive or is unreachable, the LBA doesn't report any message. Then, when LBM comes back, the LBA starts to report the load level again.

5 Evaluation

This section describes the tests and micro-benchmarks used to verify the correct operation of the DVS-SP in a DVS virtual cluster. It should be considered that the tests should have been carried out in a home virtualized environment and not a physical environment as a consequence of the inability to access the laboratories during 2020 and 2021 due to the regulations established by the national government in relation to COVID-19. This fact does not imply important consequences to demonstrate the correct behavior of the DVS-SP, but for performance measurements. It's known that CPU, memory, disk, network virtualization could distort the results.

The hardware used to perform the tests was a PC with a 6-core/12-threads AMD Ryzen 5 5600X CPU, 16 GBytes of RAM, and SATA disks. The virtualization was carried out using VMware Workstation version 15.5.0 running on Windows 10 and a cluster of 6 nodes was configured, each node in a VM: NODE{0-5}. Each VM was assigned a vCPU and 1 GB of RAM. The VMs were clones of each other running Linux kernel 4.9.88 modified with the DVK module. The DVS-SP runs on NODE0; servers run on NODE{1,2}, and clients run on NODE{3-5}. This virtual cluster only was used to test the correct behavior of the DVS-SP on allocating new sessions to new servers when the other servers are saturated and, exchange load information among the LBAs and the LBM and to test fault-tolerance on server crashes, node failures, or network partitions.

To evaluate the DVS-SP performance (taking into account the previously mentioned test environment) a minimal cluster of 3 nodes was used: DVS-SP run in NODE0, NODE1 was the server node, and NODE2 was client node. Two main metrics were measured:

- *Latency*: Two programs were used, a latency client and a latency server. The server program waits for a request and, when it receives one, it replies to the client. The client sends a request and then it waits for the reply, measuring the elapsed time among these two events. Another derived metric of these tests was the message transfer throughput.
- *Data transfer throughput*: A file transfer pair of programs was used. The client can request a GET operation to transfer a file from server to client, or a PUT operation to transfer a file from client to server. The server measures the time between the

first request received from the client and the last message sent to it then, it calculates the throughput.

In Fig. 3(A), the relative latency to a Local Ping is presented, where:

- *Local Ping:* Ping to the *localhost* interface address (average 0.036 ms).
- *Local Ping:* HTTPing to a web server on the same node.
- *Local Latency*: The client and server programs were executed on the same node.
- *Remote Ping*: Ping to Ethernet interface address of another node.
- *Remote HTTPping*: HTTPing to a web server on another node.
- *Remote Latency:* The client latency program was run on one node and the server latency program was executed on another node.
- *DVS-SP Latency*: The communications between the client and the server traverses the DVS-SP.

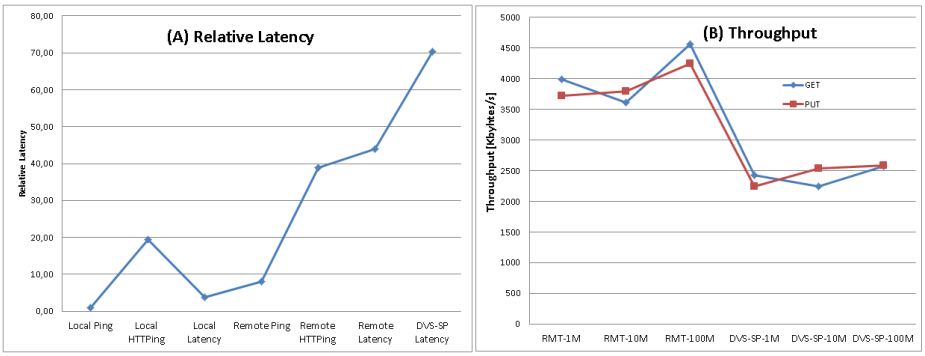


Fig. 3. (A) Relative Latency and (B) Data Transfer Throughput.

The use of remote DVS communications against a client/server HTTP communication imposes a latency penalty of 13%. Client/server using the DVS-SP adds 50% to the communication latency. This indicates that the DVS-SP is not suitable for use in high latency networks such as WANs or the Internet.

In Fig. 3(B), the data transfer throughput is presented for two scenarios: Client on one node and Server on another node (RMT) and then, with the DVS-SP between them. Three files sizes were used for transfers: 1, 10, and 100 Mbytes. Two well-known tools were used to compare file transfer performance; ***scp*** reports measurements from 13 to 20 [Mbytes/s] and ***wget*** from 10 to 44 [Mbytes/s] highlighting how resource virtualization affects performance metrics.

In the same way as latency, throughput is also affected by the use of both DVS and DVS-SP, however, there are still improvements to be made using data compression in data transfers planned for future works. Developers should consider these overheads as a tradeoff against the advantages and features provided by the DVS and the DVS-SP.

6 Conclusions and Future Works

A DVS provides scalability, reliability, and availability, it is simple to deploy and configure, and lightweight in terms of requirements which reduces the OPEX.

The contribution of this article is to present a Service Proxy with Load Balancing and Autoscaling features for a DVS as a proof of concept. Several tests were performed to demonstrate de SP capabilities as one of the several features that DVS architecture has. This DVS-SP uses a centralized and probabilistic approach to balance the computational loads and avoid backend server saturation.

The use of an AP is a common practice deploying Cloud Applications. However, a configuration with a single AP that centralizes all communications between clients and servers has in the AP a single point of failure and a performance bottleneck; therefore, reducing service availability and scalability. A future project will be to design and implement a DVS-SP cluster with replication support that can handle nodes and network failures and can tolerate higher performance demands.

For those architectures with clients and servers in the same network (in a trusted environment), a distributed Load Balancer with Autoscaling could be developed without an SP in the middle. The communications latency should be reduced and the single point of failure and performance bottleneck should be eliminated because each client will communicate with its allocated server directly.

References

1. P. Pessolani, T. Cortes, F. Tinetti, S. Gonnet: "*An Architecture Model for a Distributed Virtualization System*"; Cloud Computing 2018; Barcelona, España.2018.
2. M. Fowler. https://martinfowler.com/articles/microservices.html. Last accessed July 2021.
3. Usage statistics of Nginx. https://w3techs.com/technologies/details/ws-nginx. Last accessed July 2021.
4. Companies using HAproxy. https://enlyft.com/tech/products/haproxy. Last accessed July 2021.
5. Nginx. https://www.nginx.com/. Last accessed July 2021.
6. HAproxy. http://www.haproxy.org/. Last accessed July 2021.
7. P. Pessolani, T. Cortes, F. G. Tinetti, and S. Gonnet: "*An IPC Software Layer for Building a Distributed Virtualization System*", CACIC 2017, La Plata, Argentina, 2017.
8. Zookeeper. https://zookeeper.apache.org/. Last accessed July 2021.
9. Diego Ongaro, John Ousterhout. "*In Search of an Understandable Consensus Algorithm*", 2014 USENIX Annual Technical Conference. ISBN 978-1-931971-10-2. 2014.
10. The Spread Toolkit. http://www.spread.org. Last accessed July 2021.
11. L. E.Moser, et al., "*Extended Virtual Synchrony*", in Proceedings of the IEEE 14th International Conference on Distributed Computing Systems, Poznan, Poland, June 1994.
12. Siavash Ghiasvand, et al. "*An Analysis of MOSIX Load Balancing Capabilities*", International Conference on Advanced Engineering Computing and Applications in Sciences, November 20-25, 2011 - Lisbon, Portugal.
13. M. Beltrán and A. Guzmán, "*How to Balance the Load on Heterogeneous Clusters*," International Journal of High Performance Computing Applications , vol. 23, no. 1, pp. 99-118, Feb. 2009.