- ORIGINAL ARTICLE -

Service Migration in a Distributed Virtualization System

Migración de Servicios en un Sistema de Virtualización Distribuido

Pablo Pessolani, Luis Santiago Re and Tomás Andrés Fleitas

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

Abstract

Cloud applications are usually formed by different 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 developers are forced to maintain and configure complex sets of infrastructure, platforms, and frameworks which are expensive to implement, operate and manage. Another approach would be to use a Distributed Virtualization System (DVS) that provides a 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 commercial-class distributed applications, such as replication support (active and passive) and process migration. This article describes the mechanisms used for the migration of server processes between nodes of a DVS cluster transparently for client and server processes, doing special focus on how to solve the problem of keeping client/server communications active even when the server process location has changed.

Keywords: Virtualization, Process Migration, Distributed Systems.

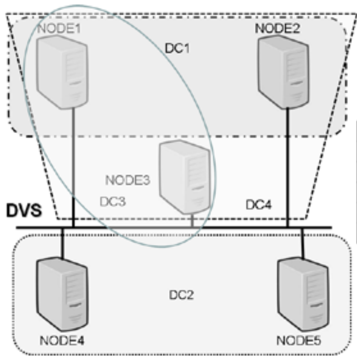
Resumen

Las aplicaciones desarrolladas para ejecutar en la nube suelen estar constituidas por múltiples componentes (microservicios) que se localizan en diferentes computadores físicos o virtuales. Para alcanzar los niveles de rendimiento, disponibilidad, escalabilidad y robustez, en este tipo de sistemas los desarrolladores se ven forzados a mantener y configurar complejos conjuntos de infraestructura, plataformas y marcos de trabajo que son costosos de implementar, operar y gestionar. Otra forma de resolver este problema es haciendo uso de un Sistema de Virtualización Distribuido (DVS) el que provee un mecanismo que permite comunicar entre sí los componentes de la aplicación en forma transparente a su localización, ocultando los problemas y complejidades añadidos por su ejecución distribuida. Este mecanismo de comunicaciones cuenta con características específicas para el desarrollo de aplicaciones distribuidas de clase-comercial tales como el soporte de Replicación (Activa y Pasiva) y de Migración de Procesos. En este artículo se describen los mecanismos que permiten realizar una migración de procesos servidores entre nodos de un cluster de un DVS en forma transparente para los procesos clientes y servidores, y la forma como se resuelve el problema de mantener activas las comunicaciones cliente/servidor aun cuando el proceso servidor haya cambiado de ubicación.

Palabras claves: Virtualización, Migración de Procesos, Sistemas Distribuidos

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 can be extended to a cluster of physical machines (nodes). Although there are several distributed processing technologies, those that offer simpler ways of implementation, operation, and maintenance are highly valued because they reduce costs. 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 and migration 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 a 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). A topological diagram of a DVS cluster is shown in Fig. 1.



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

Fig. 1. Example of a DVS topology.

An issue that must be considered when using a distributed application refers to the location of a service used by an external or internal client, or by another component of the application itself. One way to solve this problem, without using the DVS Inter-Process Communication (IPC) facilities, 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 considered when working with a cluster is that the network and its nodes may fail, preventing continuity in the delivery of a given service. A similar problem presents process migration, which is in general used in those cases in which a cluster node, where a service is running, is overloaded or its disconnection has been planned to maintain the node's hardware. The applications that use this service must not be disrupted by the migration to maintain service availability. The destination node of the migrated service will have another IP address (and other MAC address), which forces service clients to know about these new addresses to continue operating with it.

When virtual machines (VMs) are used on hypervisors such as VMware ESXi [2] or KVM [3], the migration at the VM level is solved using virtual networks where the source VM is connected to a distributed virtual switch, then migrated to the destination host keeping its MAC and IP addresses and connected to the same virtual switch.

The problem of maintaining communications between a client and a server after server migration can be faced with such as those used by VM migration. These solutions rely on network management and use two different approaches: to keep an IP address when performing a migration or changing it. One possible solution is to transform the problem of migrating VMs between independent subnets into a problem of migrating VMs in the same subnet. Tools such as OpenFlow [4] and VXLAN [5] can be used which are based on tunneling strategies, modified routing [6], and layer 2 expansion [5, 7].

Another solution is to use a load balancer, which, in short, operates as a reverse proxy by which clients connect to servers. At the time of migration, the load balancer could be in charge of preserving the information necessary to reestablish the communication of the clients with the migrated server transparently. This technology is widely used in certain scenarios such as web applications, where end-users send requests from their devices as clients, and the load balancer is the one that establishes a session with the corresponding server, thus distributing and balancing the load between them. But, in a scenario where both client and server are part of the same cluster, the use of a load balancer could be detrimental, since it centralizes communications among clients and servers, transforming it into a single point of potential failure and this could end up reducing service availability.

Using a DVS is a simpler and more transparent solution that handles the state and configuration of the cluster in a distributed way. A DVS maintains the identification of its nodes and the nodes which compose each DC. A highly valued feature a DVS has is the mechanisms to maintain communications between clients and a server process, even when changes in the location of the latter may occur as a result of its migration to another node in the cluster.

When a server changes its location from a source node to a destination node, the processes that communicate with it must be able to maintain their communications despite the change and in a transparent way, to simplify programming, management, and maintenance.

There are several ways to handle this problem using a DVS. First, by enabling a distributed service called RADAR [8], which manages the location of services that require fault tolerance or, as in this case, process migration. Second, by using DVS's specific commands meant to be used to notify the location of certain processes (generally servers) in the cluster, and to keep ongoing communications between the processes alive. Third, by using the DVS APIs to develop an application that manages migrations.

One of the main components of a DVS is the Distributed Virtualization Kernel (DVK), which is integrated into the Linux kernel as a module. All the mentioned utilities are available through the DVK APIs. In the test scenarios presented in this article, RADAR has not to be used, to focus on the mechanisms that the DVS can use to support process migration, but not in replication scenarios.

The process migration support available in a DVS is fully transparent to the underlying network because it does not require additional network configurations to those already established when configuring the DVS, and therefore, simplifies its deployment, use, and maintenance.

Another problem that process migration usually presents is the treatment of PIDs (Process IDentifier). When a process is migrated, it will have a PID assigned to it at the source node and another, generally different PID, will be assigned to it at the destination node. If the process, once migrated, makes a *getpid()* system call, it will obtain a different value than what it could have obtained in the source node. For this reason, a virtualization layer must be implemented, in which the process PID becomes a global attribute of the process, associated with a local PID in the node where it is being executed in a given moment. Several distributed virtual OSs which keep the process' PID unchanged after it has migrated have been implemented for the DVS.

This article presents how process migration is done in a DVS without disrupting active communications, as a proof of concept of one of the many features a DVS has. The main contribution of this work relies on explaining the particular approach used in a DVS with its underlying migration support which allows providing services with greater availability.

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 how DVS process migration works. Section 5 presents the scenarios used, and the results of the evaluation of process migration and finally, the conclusions and future work are summarized in Section 6.

2. Related Works

The problem raised is not unknown by the scientific community, so previous research and development works have been carried out to solve it.

Process migration is called homogeneous when it is carried out between machines (virtual and/or physical) with the same architecture (ISA) and same OS. Process migration is called heterogeneous when it is carried out between machines with different architecture or OS. Within these, process migrations done at the user-level are differentiated from the ones done at the kernel-level. The former is generally easier to implement and maintain but, it has certain disadvantages, such as the inability to migrate certain processes, and the need to use system calls, which are slow and expensive.

To carry out a homogeneous process migration, five well-known algorithms are mentioned and briefly explained below [9]:

- *Total-copy algorithm*: it consists of suspending the process, then transferring all its status information, and finally resuming it on the destination node. It is simple, easy, and does not have residual dependencies, but it has a long delay caused by having to transfer the whole state of the process.

- *Pre-copy algorithm*: it consists of transferring the whole state of the process, then suspending it and transferring the changes that may have occurred in its state, and finally resuming it in the destination node. Despite being a bit more complex than the Total-copy algorithm, this algorithm has a lower downtime.

- *On-demand pages algorithm*: the largest part of a process state is its virtual address space so, on this algorithm, the process is suspended, then transfers its state except for the virtual address space, which will be requested, if required in the destination node. This approach is fast, but it maintains dependencies with the source node, meaning that the source node must keep on the information on the migrated process until it finishes.

- *File server algorithm*: it is very similar to the on-demand pages algorithm, but it introduces a third node called the file server, which will precisely be in charge of storing the virtual address space of the migrated process, to avoid dependency between different possible nodes, and always keep dependencies only with the file server node.

- *Nonfreezing algorithm:* this algorithm proposes two important optimizations against the previous ones. On the one hand, about the virtual address space, it looks for the pages in use at the time of migration. Only migrates those together with the process state, the rest are sent later and in case of page faults, they are handled as in the on-demand page algorithm. On the other hand, this algorithm separates the communications module of a process from the rest of its state, to first migrate the process but not its communications, queuing the messages that arrive during this time, and then just migrates the communications module. This achieves to considerably reduce the communications freeze time.

Handling communications during a process migration is something that increases its complexity. Some techniques used by classic Distributed Operating Systems (DOS) to perform process migrations while maintaining their communications are analyzed below.

2.1. Mosix

Mosix [10, 11] can be used to build a DOS in a Linux cluster. Mosix is implemented as a kernel-level loadable module and has a set of tools and libraries. In Mosix, a process that has been migrated shares the runtime environment of its Unique Home Node (UHN), the node on which it was started.

Two algorithms are provided for sharing resources: load balancing and remote node memory usage. When a node's memory runs out, the remote node memory usage algorithm is triggered. A given process is migrated to a node that has enough free memory, but it maintains interaction with its original environment. The context of the process selected for migration is divided into two parts: deputy and remote. The attached context remains in UHN and cannot be migrated. The remote part of a process is a user context and can be migrated. Therefore, all processes that have been migrated to other nodes interact with the user's environment through the UHN and use the remote node's resources when possible.

This way of migrating processes represents leaving a residual dependency on the UHN that affects the availability of the service provided by the process, but, on the other hand, the migration has no consequences on the IP addressing (as an associated process of the migrated one remains in the kernel on the UHN) and communications remain uninterrupted. For this reason, Mosix provides transparent process migration and automatic load balancing across the cluster.

2.2. OpenSSI

OpenSSI [11] is a Single System Image DOS. To manage and balance loads, OpenSSI implements and uses a process migration mechanism. The migration scheme used in OpenSSI is derived from the one used by Mosix, but unlike it, it does not require a deputy process on the source node. This is achieved through the use of a virtualization layer implemented as a Linux kernel extension that manages Vprocs (virtual processes). For example, the PID of a process remains even after it has migrated to another node. This PID is virtual and is associated with the real PID of the local node (node where the process is running).

2.3. Kerrighed

Kerrighed [11] is another SSI DOS that additionally implements Distributed Shared Memory (DSM). It implements several useful mechanisms when migrating processes or threads (memory sharing supports thread migration). It first performs the *checkpoint* of a process to obtain relevant information about its status and then performs the migration itself through a stream exclusively dedicated to this, which guarantees a high-efficiency level. However, a disadvantage of Kerrighed is the inability to add or remove nodes to a cluster after it has been started and a failure of one node can cause the failure of the entire cluster.

2.4. Amoeba

Amoeba is an ancient SSI DOS developed by A. S. Tanenbaum based on a microkernel [12]. It uses a high-performance network protocol called FLIP [13] that supports RPC, group communications (GCS), support for process migration, and important security features. Each process is assigned a Network Service Access Points (NSAPs) that are independent of their location, so they can be located at any node in the cluster. This feature facilitates process migration.

The process migration implementation in Amoeba is based on three key points. First, separate the migration mechanism from the policy to be used, that is, separate how a migration is carried out from when and where it is carried out. On the other hand, achieve transparency for the processes, that is, they should not worry about where they are being executed, or about possible migrations of both themselves and other processes with which they are related. And finally, avoid residual dependencies preventing a process from continuing to depend on its original node.

3. Background Technology

This section presents the developments, products, and tools that have been studied and analyzed as technological support for the design, and implementation of process migration in a DVS prototype.

3.1. M3-IPC

The DVK provides programmers with an advanced IPC mechanism named M3-IPC [14] which is available at all nodes of the DVS cluster. M3-IPC provides APIs to carry out transparent communications between processes located in the same (local) or other (remote) node. To send messages and data between processes of different nodes, M3-IPC uses Communications Proxies processes. Those proxies can use different transport/network protocols which can include encryption, compression of data, and batching of messages. It also hides communication disruptions that involve replicated server processes in case of server failure or when it migrates to another node of the DVS cluster. On a process migration, communications are reestablished after the process has migrated or returned to its source node (failed migration).

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.

3.2. CRIU

One of the most used sequences of actions when migrating a process is: 1) carry out a checkpoint, which consists of stopping the process to capture and store its status at a given point in time; 2) transferring the stored state of the process from the source node to the destination node, and finally 3) restore the state of the process and execute it from this checkpoint.

The first implementations were carried out with the intervention of the kernel, which caused that this solution was not accepted by the Linux community [15].

In particular, the CRIU project [16] solves this problem, since it implements the user-space checkpoint and restores mechanism, using the available kernel interfaces through system calls or pseudo-filesystems such as */proc*. One of those most important interfaces to accomplish this task is the *ptrace* system call, which provides a means for one process to observe and control the execution of another process and examine and change its memory and registers.

Finally, in the restore, CRIU allows to re-identify the process with the PID that it had during the checkpoint. To achieve this, CRIU writes one number less than the desired PID to */proc/sys/kernel/ns_last_pid* and then validates that the newly created process has the correct PID, otherwise the process restore is aborted.

It was not possible to integrate CRIU to DVS in these instances due to incompatibilities in the architectures of both products. The stable versions of CRIU require 64-bit architectures, but the current version of the DVS prototype only supports the i386 and i686 architectures, both 32-bit.

3.3. DMTCP

DMTCP [17, 18] is a tool that provides checkpoint and restore (C/R) mechanisms at the user-level. Each process to be supervised by DMTCP must be associated with a coordinator process (Fig. 2) at the time of its execution. This allows the coordinator to later capture the status of these processes on demand. Then, when a process fails or is terminated for some reason, it can be resumed from one of the captured states. However, the resumption of these processes must necessarily be done using DMTCP.

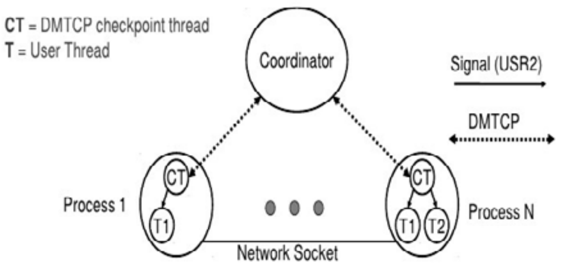


Fig. 2. DMTCP components (from [18]).

The DMTCP Coordinator communicates with the processes it controls through a thread called Coordinator Thread (CT). In this way, when the Coordinator requests that a process checkpoint be generated, it sends a command to the CT. Upon receiving this command, the CT generates a *SIGUSR2* signal so that each thread of the process is suspended and the image to be stored in a file can be generated.

DMTCP only captures the state of a process and then resumes it, possibly on another node. However, sending the process state between nodes and handling its communications must be done separately, using some other tools, because DMTCP does not provide these features. Once the process image file has been obtained, it must be transferred to the destination node. In the latter, the *dmtcp_restart* command is executed, and the process is restarted.

4. Process Migration in a DVS

Process migration is a technique used to dynamically load balance between the nodes of a cluster to relieve a node that requires repair. It is also used to consolidate services in a fewer number of active nodes during periods of low demand and thus reduce the energy consumption of the infrastructure.

The DVS supports that a process associated with an *endpoint* can be migrated from its source node to a destination node without having to modify the applications (neither the migrated process nor the processes that communicate with it). Its goal is for the communications to remain active but suspended while the migration process is being carried out. Then, once the migration is finished, the communications are reestablished without changing the behavior of the involved processes, except for perceiving an additional delay. This property of the communication system is known as Migration Transparency.

To migrate a process in a DVS, the DVK of each of the involved nodes (source, destination, and communication counterparts) must be notified that the process whose *endpoint* is in an active state will be migrated. It is assumed that there is a distributed process management system that runs in each of the nodes of the DVS cluster, although all this operation can be done manually or through scripts, as shown in the tests presented in the next section. Using a command line or a web application, the migration manager is instructed that a given process will be migrated (*MIGR_START*) which has as parameters the process to be migrated identified by the {*DC, endpoint*} tuple.

This *MIGR_START* command is broadcasted to all DVS nodes, and each node will take different actions depending on the state of the endpoint in that node. These states can be:

- *Active*: This means that the process (*PROC_RUNNING*) to be migrated is executing on that node (source), so the endpoint is active and upon receiving the migration start command, it will go to the *MIGRATING* state until the migration is finished or aborted. The DVK ensures that no messages will be received/sent by that *endpoint*.
- *Backup*: This means that the *endpoint* has been registered as a backup type process (*MIS_RMTBACKUP*) and the process to be migrated (Primary) is running on another node (*REMOTE*). The *endpoint* will go into a state waiting for migration (*MIGRATING*) and no messages will be received/sent by that *endpoint*.
- *Remote*: This means that the *endpoint* has been registered as a remote process and is running in another node (*REMOTE*) of the DVS cluster. The *endpoint* will go into a state waiting for migration (*MIGRATING*) and no messages will be received/sent by that endpoint.
- *Not registered*: This means that there is no record of the process to be migrated in this node's DVK, so there will be no changes.

Once communications with the *endpoint* to be migrated have stopped, the process can be migrated from the source node to the destination node with any tool designed to do so. DMTCP was the tool used to test DVS migration scenarios because it fully runs in user-space avoiding architecture compatibility issues and facilitating debugging. Furthermore, it is easy to use and does not require kernel modifications.

Although a single DMTCP Coordinator can be used in a DVS, one or multiple Coordinators by each node can be run (each one using its own TCP port), avoiding performance bottlenecks and scalability issues.

Once the migration is finished, a *MIGR_COMMIT* command must be executed on all the DVS nodes, which will execute different actions depending on the endpoint type on the node:

- *Source Node*: the *endpoint* will be registered as remote (*REMOTE*), pointing to the destination node.
- *Destination Node*: the *endpoint* will be registered as active (*PROC_RUNNING*) in the local node.
- *Nodes with endpoint registered as Remote*: the type of *endpoint* will be kept as remote, but the DVK will modify the node ID where the *endpoint* is located, from the source node to the destination node.
- *Unregistered*: There are no changes on these nodes.

Once the *MIGR_COMMIT* is received, the DVK of each node where the *endpoint* is bound reestablishes its communications. If the process migration fails; a *MIGR_ROLLBACK* command must be sent to all the DVS nodes to abort the migration. Then, the *endpoint* in each node returns to its state before starting the migration, and its communications previously stopped are reactivated.

5. Evaluation

This section describes the test scenarios used to verify the correct operation of the M3-IPC migration support in a DVS cluster.

It should be considered that the tests had to be carried out in a virtualized environment and not a physical environment as a consequence of the inability to access university labs during 2020 and 2021 due to the regulations established by the national government about COVID-19. This does not imply important consequences for the proof of concept that is to be demonstrated for the following reasons:

- Currently, most applications run in virtualized environments, so in some way, the test environment would be similar to a production environment.

- The goal of these tests is to demonstrate the correct behavior of the communications migrating the server counterpart. It is not intended to evaluate the performance of the migration itself because the major impact would be the migration tool used (in this case, DMTCP), the file transfer software (in this case, *rsync*), and the testing infrastructure (node hardware and networking).

The hardware used to perform the tests was a PC with an AMD Ryzen 5 5600x with 6 cores (12 threads) 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 3 nodes was configured, each node in a VM: NODE0, NODE1, and NODE2. 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. All tests were run 10 times.

For the test mockup, a specific client and server were developed in a way to allow recording the status of each one of them, a fundamental aspect to detect possible changes in their behavior due to the migration of the server. At the start, the server process allocates a specified amount of memory and fills it with random data. This is done to have processes with several memory sizes.

The client sends a request message to the server, transfers a block of random data, and then it waits for the reply message. These operations were repeated 100 times for each test. Based on this interaction between the programs, the test itself consisted of performing the migration of the server process while the communications were being carried out and then verifying that they were resumed correctly after the server process has migrated or performed a rollback if it can't. The migration scenarios are detailed below.

5.1. Migrating the Server from a Remote Node to the Client's Local Node

In this scenario, initially, the client process is running on NODE1 and the server process is running on NODE0, both in the DC0 environment. The DMTCP coordinator also runs on NODE0 but does not belong to any DC. It is important to highlight that the process to be migrated must be executed by a program that reports to the DMTCP coordinator. The steps to carry out the migration of the server to NODE1 are the following (Fig. 3):

- The DVK of NODE1 is notified that the server will initiate a migration (*dvk_migr_start*). In this way, the DVK stops the client's communication with the server and waits to be notified when the server is available again, either because it migrated successfully or because the migration failed.

- The DVK of NODE0 is notified that the server process will start a migration (*dvk_migr_start*). In this way, the DVS stops communications to and from the server.

- Using the *dmtcp_command* command, a checkpoint of the server process in NODE0 is carried out, which captures its status in an image file and then ends it.

- The image file is transferred from NODE0 to NODE1 with a tool such as *rsync*.

- The server process is resumed from its image file with DMTCP on NODE1 using the *dmtcp_restart* command.

- If the migration was successful, the DVK of NODE1 is notified so that it resumes the communications of the migrated process (*dvk_migr_commit*), which unlocks both the client and the server to continue to exchange messages.

- Otherwise (failed migration), the DVK is notified to restore the server process on NODE0 (*dvk_migr_rollback*) and then the DVK on NODE1 to unlock the client to continue to exchange messages.

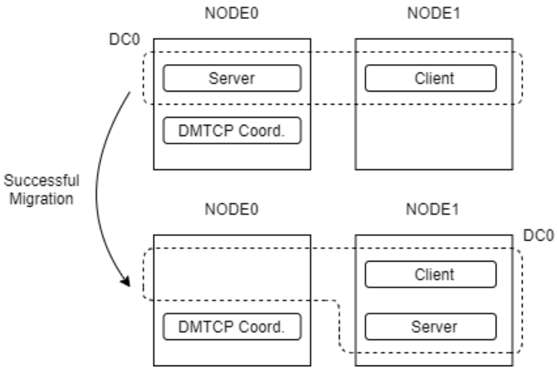


Fig. 3. Server migration from a remote node to the Client's local node.

5.2. Migrating the Server from the Client's Local Node to a Remote Node

In this scenario, initially, both the client process and the server are running on NODE1, both in the DC0 environment. The DMTCP coordinator also runs on NODE1 but does not belong to any DC. The steps to carry out the migration of the server to NODE0 are the following (Fig. 4):

- The DVK of NODE1 is notified that the server will initiate a migration (*dvk_migr_start*). In this way, the DVK stops the client's communication with the server and waits to be notified when the server is available again, either because it migrated successfully or because the migration failed.

- The DVK of NODE0 is notified that the client process is located in NODE1. This is required so that after the server is migrated, messages are routed correctly to the client.

- Through the *dmtcp_command* command, a checkpoint of the server process in NODE1 is carried out, which captures its status in an image file and then ends it.

- The image file is transferred from NODE1 to NODE0 with a tool such as *rsync*.

- The server process is resumed from its image file with DMTCP on NODE0 using the *dmtcp_restart* command.

- If the result of the migration was successful, the DVK of NODE1 is notified so that it resumes the communications of the migrated process (*dvk_migr_commit*), which unlocks both the client and the server to continue to exchange messages.

- Otherwise (failed migration), the DVK is notified to restore the server process on NODE1 (*dvk_migr_rollback*) and then unlock the client to continue to exchange messages.

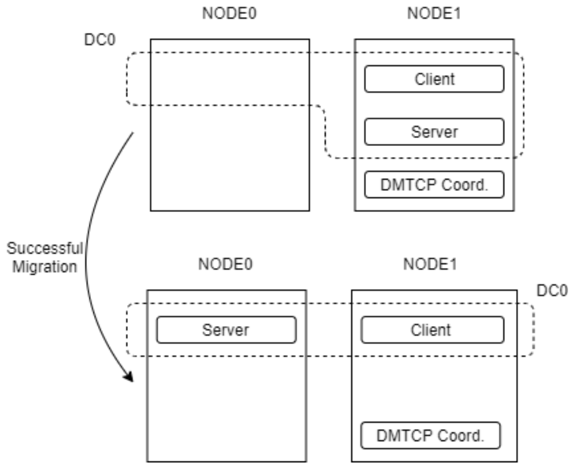


Fig. 4. Server migration from the Client's local node to a remote node.

5.3. Migrating the Server from a Remote Node to another Remote Node

In this scenario, initially, the client process is running on NODE2 and the server process is running on NODE0, both in the DC0 environment. The DMTCP coordinator also runs on NODE0 but does not belong to any DC. The steps to carry out the migration of the server to NODE1 are the following (Fig. 5):

- The NODE2 DVK is notified that the server will initiate a migration (*dvk_migr_start*). In this way, the DVK stops the client's communication with the server and waits to be notified when the server is available again, either because it migrated successfully or because the migration failed.

- The DVK of NODE1 is notified that the client process is located in NODE2. This is required so that after the server is migrated, messages are routed correctly to the client.

- The DVK of NODE0 is notified that the server process will start a migration (*dvk_migr_start*). In this way, the DVS stops communications to and from the client.

- Using the *dmtcp_command* command, a checkpoint of the server process in NODE0 is carried out, which captures its status in an image file and then ends it.

- The image file is transferred from NODE0 to NODE1 with a tool such as *rsync*.
- The server process is resumed from its image file with DMTCP on NODE1 using the *dmtcp_restart* command.
- If the result of the migration was successful, the DVK of NODE2 is notified so that it resumes the communications of the migrated process (*dvk_migr_commit*), which unlocks both the client and the server to continue to exchange messages.
- Otherwise (failed migration), the DVK is notified to restore the server process on NODE1 (*dvk_migr_rollback*) and then unlock the client to continue to exchange messages.

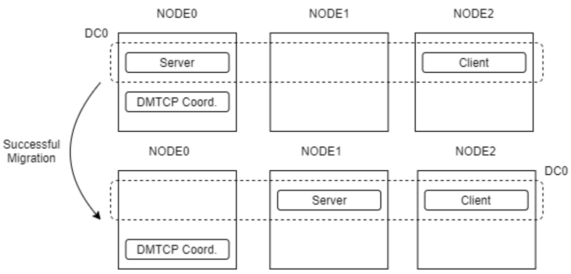


Fig. 5. Server migration from one remote node to another remote node.

5.4. Performance Results

For each scenario, two sets of tests were done. The first was to transfer the full process image from the source node to the destination node using *rsync*.

The second set of tests was to make a process checkpoint before its migration then, transfers a full process image then, starts the migration taking another checkpoint, comparing both process image generating a differential image file, transfers this file, build the migrating image using the pre-migrating image and the differential image.

The *rsync* tool was used to transfer process images with compression enabled.

The migration time is computed since the first process is suspended (the Client) until it resumes its execution.

The results of the first set of tests are presented in Fig. 6.

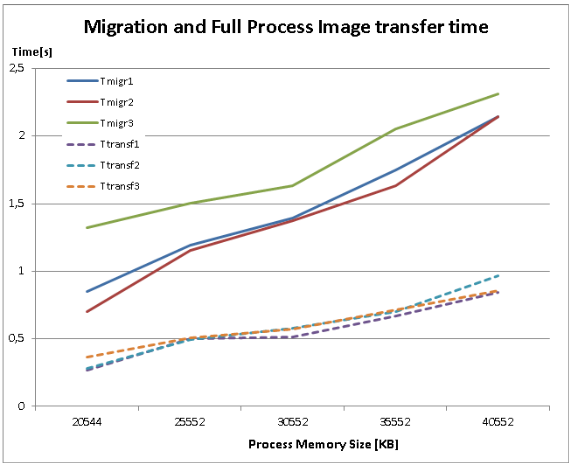


Fig. 6. Migration and Full Process Image Transfer Time.

The full process image transfer time consumes about 20%-30% of the total migration time. It should be considered that all tests were executed using user-space commands. Therefore, the resulting migration times include components such as the time to execute several remote commands via *ssh* which involves creating a session, identifying, and authenticating a user (automated), starting a shell, executing a shell script, and return the output.

To evaluate the performance of transferring a pre-migration process image file and then, during migration, transferring the difference image file, the *bsdiff* tool was used. Therefore, in the source node, the difference image file needs to be created using the pre-migration process image, and the migration process image. In the destination node, once the difference image file is received, the migration process image must be built using the pre-migration and difference image files.

It is important to note that the differences between images are about 4 Kbytes, which was the block of random data that the Client sends to the Server's buffer.

The results of the first set of tests are presented in Fig. 7.

Although the difference process image transfer time has a minimal impact in migration time, the total migration time is about 3 times the migration time transferring the full process image file. The marked increase in migration time is due to the execution of the *bsdiff* command on the source node creating the difference image file, and on the target node building the migrated process image file. Therefore, it seems that it is not convenient to use the pre-copy approach, at least with user-space tools.

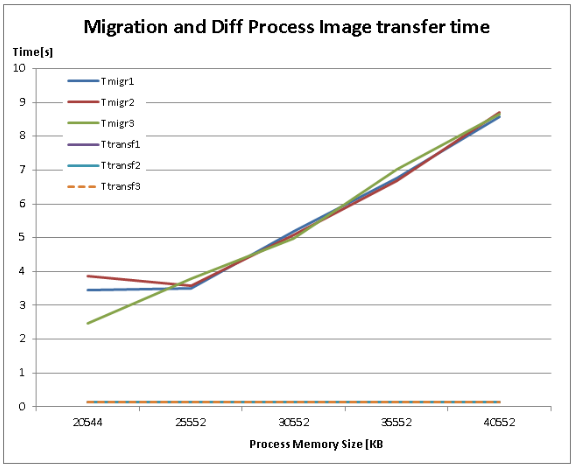


Fig. 7. Migration and Diff Process Image Transfer Time.

For the rest, both in the migration of the full process image, as in the pre-copy migration, the growth of the migration time increases linearly with respect to the process size.

The test scenarios presented refer to successful migrations but, in all of them, the same tests were performed simulating failed migrations. The results obtained showed a correct behavior in terms of continuing with the exchange of messages between the client and the server.

6. Conclusions and Future Works

There are several approaches to face process migration. Some of them are acceptable in terms of their scalability and availability, but they lack simplicity in terms of implementation and management. It is necessary to deal with multiple configurations, which are generally managed by different workgroups (developers, operators, IT security groups, service providers, etc.) and, although there are useful tools to deal with them, they increase even more the complexity related to manage and operate distributed applications.

A DVS provides scalability, reliability, and availability features, it is simple to implement, deploy and configure; and lightweight in terms of requirements, reducing the related costs.

The main contribution of this work relies on explaining the particular approach used in a DVS with its underlying migration support which allows the development of reliable distributed applications.

Current technologies used to support process migration are based on LAN and IP protocols, although they are well-known, they are more complex, and the applications must be written for the network (such as using sockets). On the other hand, a DVS provides APIs to write Client/Server applications without considering the location of the processes and another approach was used to support process migration. To reuse code of network applications, the DVK kernel module is being modified (in the scope of an ongoing project), so that M3-IPC can be used as a new socket domain (AF_M3IPC).

Several test scenarios of process migration were evaluated verifying that communications between processes keep active after successful migrations of failed ones. Maintaining communications after process migration increases the availability and performance of distributed and Cloud applications. The correct behavior of the client and the server in all the tests presented here demonstrates one of the several features that a DVS has. Process migration is easy to implement in a DVS, and transparent to the communicating processes. Client processes can continue receiving service after a server migration, even when the migration fails.

As future work, it is proposed to develop a distributed scheduling system that periodically evaluates the load on each DVS node and, if necessary, performs a redistribution of the whole load using process migration to balance the use of resources.

Competing interests

The authors have declared that no competing interests exist.

Authors' contribution

Pablo Pessolani is the leader of the project exposed here. He conceived the idea and presented the major components to the research team, assigned the roles to each member and supervised them. Luis Santiago Re and Tomás Andrés Fleitas searched for and tested the best tools available to carry out the experiments. The whole team worked on the development of the tests and interpretation of the results. All authors wrote the draft, read and approved the final manuscript.

References

[1] P. Pessolani, T. Cortes, F. Tinetti, S. Gonnet: "An Architecture Model for a Distributed Virtualization System"; Cloud Computing 2018; The Ninth International Conference on Cloud Computing, GRIDs, and Virtualization; Barcelona, España.2018.

[2] "VMware Infrastructure Architecture Overview". White paper. Available at: https://www.vmware.com/pdf/vi_architecture_wp.pdf .Accessed on 2021-04-01.

[3] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori: "KVM: the Linux Virtual Machine Monitor", In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07), 2007.

[4] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schioberg: "Live ¨ Wide-Area Migration of Virtual

Machines Including Local Persistent State," in SIGPLAN VEE. ACM, 2007.

[5] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal et al:, "VXLAN: A Framework for Over-laying Virtualized Layer 2 Networks over Layer 3 Networks," Internet Draft (Work in Progress), 2013.

[6] D. Erickson, G. Gibb, B. Heller, D. Underhill et al.: "A Demonstration of Virtual Machine Mobility in an OpenFlow Network," in SIGCOMM (Demo). ACM, 2008.

[7] K. Kompella and Y. Rekhter, "Virtual Private LAN Service (VPLS) Using BGP for Auto-Discovery and Signaling," RFC 4761 (Proposed Standard), 2007.

[8] Pablo Pessolani; David Gabriel Harispe; Octavio Garcia Aguirre: "Localizacion y Seguimiento de Servicios Replicados en un Sistema de Virtualizacion Distribuido", Revista Digital del Departamento de Ingenieria e Investigaciones Tecnologicas; vol.: 5 - nro. 1 (agosto-2020) ISSN: 2525-1333.

[9] R. Lawrence; "Introduction A Survey of Process Migration Mechanisms". Department of Computer Science University of Manitoba, 1998

[10] A. Barak and A. Shiloh. The MOSIX Cluster Management System for Distributed Computing on Linux Clusters and Multi-Cluster private Clouds white paper, 2016.

[11] P. OsiĚski, E. Niewiadomska-Szynkiewicz: "Comparative Study of Single System Image Clusters", Evolutionary Computation and Global Optimization 2009 / National Conference 2009 ; Zawoja, Poland.

[12] S. J. Mullender, G. van Rossum, A. S. Tananbaum, R. van Renesse, H. van Staveren: "Amoeba: a distributed operating system for the 1990s", in Computer, vol. 23, no. 5, pp. 44-53, May 1990

[13] M.F. Kaashoek, R. Renesse, H. van Staveren, and A.S. Tanenbaum: "FLIP: An Internet-work Protocol for Supporting Distributed Systems". ACM Transactions on Computer Systems, 11(2):73-106, 1993.

[14] P. Pessolani, T. Cortes, F. G. Tinetti, and S. Gonnet: "An IPC Software Layer for Building a Distributed Virtualization System", Congreso Argentino de Ciencias de la Computación (CACIC 2017) La Plata, Argentina, October 9-13, 2017

[15] "Criu - checkpoint/restore in user space". Available at: https://access.redhat.com/articles/2455211. Accessed on 2021-04-01.

[16] "CRIU" Available at: https://criu.org/Main_Page, Accessed on 2021-04-01.

[17] "DMTCP" Available at: http://dmtcp.sourceforge.net/index.html .Accessed on 2021-04-01.

[18] "DMTCP_Distributed_MultiThreaded_CheckPointin g_Works" Available at: https://es2.slideshare.net/jserv/implement-checkpointing-for-android/13-. Accessed on 2021-04-01.

Citation: P. Pessolani, L.S. Re and T.A. Fleitas. *Service Migration in a Distributed Virtualization System.* Journal of Computer Science & Technology, vol. 21, no. 2, pp. 177-187, 2021.
DOI: 10.24215/16666038.21.e16
Received: April 9, 2021 **Accepted:** October 6, 2021.