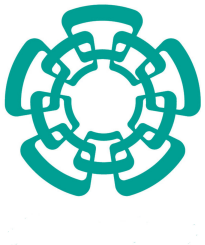


Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Cinvestav Tamaulipas

Método para la construcción de tuberías de procesamiento de datos para la nube

Tesis que presenta:

Hugo German Reyes Anastacio

Para obtener el grado de:

**Maestro en Ciencias
en Ingeniería y Tecnologías
Computacionales**

Dr. Víctor Jesús Sosa Sosa, Director
Dr. José Luis González Compeán, Co-Director

Cd. Victoria, Tamaulipas, México. Septiembre, 2017

La tesis presentada por Hugo German Reyes Anastacio fue aprobada por:

Dr. Hiram Galeana Zapién

Dr. Edwyn Javier Aldana Bobadilla

Dr. Víctor Jesús Sosa Sosa, Director

Dr. José Luis González Compeán, Co-Director

Cd. Victoria, Tamaulipas, México., 20 de Septiembre de 2017

A mi familia

Agradecimientos

- Agradezco a mi madre, que con esfuerzo y sacrificio me guió por el camino que me permitió cumplir esta meta.
- Así como a mis hermanos y amigos por su apoyo incondicional.
- A mis directores de Tesis, los Drs. Víctor Jesús Sosa Sosa y José Luis González Compeán por su dirección y atenciones a lo largo del trabajo de tesis.
- A mis revisores, los Drs. Hiram Galeana Zapién y Edwyn Javier Aldana Bobadilla por sus valiosos comentarios y sugerencias.
- A mis padres académicos, el M.C Santiago Gómez Carpizo y su esposa la Dra. Maria Esther Bautista Vargas por iniciarme en el camino de la investigación.
- A mis compañeros, por ser parte de este viaje compartiendo anécdotas y experiencias.
- A Deisy mi compañera de vida, por acompañarme a lo largo de este proyecto y los que están por iniciar.
- A Raúl André mi hijo, el motor más importante para concluir este proyecto y quien alegra mis días más pesados.
- Al CINVESTAV-Tamaulipas, por la oportunidad que me brindó para estudiar su posgrado y los recursos proporcionados para la realización del mismo.
- Y al CONACyT, por el apoyo económico con el cual pude concentrarme en mis estudios.

Índice General

Índice de Figuras

Índice de Tablas

Índice de Algoritmos

Publicaciones

Reyes Anastacio, Hugo German; Morales-Sandoval, Miguel; González-Compeán, José Luis. *Un prototipo para verificación remota de integridad de datos en la nube*, en tercer congreso nacional de ingernierí a, Ciudad Victoria, Tamaulipas, México, Septiembre 2016.

Resumen

Método para la construcción de tuberías de procesamiento de datos para la nube

por

Hugo German Reyes Anastacio
Unidad Cinvestav Tamaulipas
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, 2017
Dr. Víctor Jesús Sosa Sosa, Director
Dr. José Luis González Compeán, Co-Director

Las organizaciones que manejan grandes repositorios de acervos de información deben hacer frente a tres requerimientos: i) Flexibilidad en términos de *variedad* de procesos de codificación tales como corrección de errores, compresión, cifrado, etc., que se aplican a los archivos para cubrir restricciones de manejo impuestas por organizaciones y/o instancias gubernamentales. ii) Elasticidad en términos de procesamiento de acervos de grandes *volúmenes* de archivos de grandes dimensiones. iii) Eficiencia en términos de la *velocidad* de procesamiento de codificación y utilización de almacenamiento requeridos para preservar los archivos del acervo. Soluciones basadas en tuberías de procesamiento desplegables en infraestructuras de máquinas físicas o en máquinas virtuales en la nube han sido propuestas para hacer frente a los requerimientos antes mencionados. Sin embargo, ambas soluciones presentan inconvenientes. Por ejemplo, el despliegue de tuberías en infraestructuras de máquinas físicas es complejo debido a las dependencias con la infraestructura, lo cual compromete el requerimiento de flexibilidad. Las tuberías desplegadas en máquinas virtuales comprometen el requerimiento de eficiencia y producen subutilización de recursos. En el presente trabajo de Tesis se propone un método para la construcción de tuberías de procesamiento de datos basada en bloques de construcción encapsulados en contenedores virtuales. Los bloques de construcción permiten el acoplamiento de diversos módulos de procesamiento en forma dinámica, lo cual satisface el requerimiento de flexibilidad y *variedad*. La encapsulación de los bloques de construcción en contenedores virtuales permite no sólo desplegar tuberías bajo demanda sino desplegarlas en

diversos tipos de infraestructuras, lo cual evita la subutilización de recursos y permite satisfacer el requerimiento de elasticidad y procesamiento de grandes *volúmenes* de archivos. El esquema de despliegue llamado divide y encapsula propuesto en esta tesis, permite desplegar, en la nube, tuberías en paralelo lo cual mejora la *velocidad* de procesamiento de los archivos del acervo así como la utilización del almacenamiento, lo cual permite satisfacer el requerimiento de eficiencia. El método propuesto fue desarrollado en un prototipo que fue evaluado en diferentes infraestructuras y escenarios de experimentación. La evaluación reveló la factibilidad de aplicar tuberías de procesamiento en la nube basada en bloques de construcción encapsulados en contenedores virtuales, los cuales satisfacen los requerimientos de *variedad*, *volumen* y *velocidad* inherentes al manejo de grandes acervos de información.

Abstract

Method for the construction of data processing pipelines for the cloud

by

Hugo German Reyes Anastacio
Cinvestav Tamaulipas
Research Center for Advanced Study of the National Polytechnic Institute, 2017
Dr. Víctor Jesús Sosa Sosa, Advisor
Dr. José Luis González Compeán, Co-advisor

The organizations managing large repositories of digital heritage files must face up the meeting of three requirements: i) Flexibility in terms of *variety* of coding processes such as erasure coding, compression, encryption, etc., which are applied to files to meet management requirements imposed by either organizations or government instances. ii) Elasticity in terms of processing large *volumes* of large files iii) Efficiency in terms of *compute* and storage utilization required to preserve archives of repositories. Software pipeline-based processing solutions have been proposed for infrastructure-based physical machines and virtual machines in the cloud to address the aforementioned requirements. However, the deployment of pipelines in both types of infrastructure results in several issues. For instance, deploying pipelines by using physical machines is not trivial because of dependencies, which reduces the solution flexibility, whereas the deployment on the cloud impacts of the performance of solution and produces resource underutilization, which reduces the solution efficiency. In this thesis, a building block construction model for data processing pipeline based on containers is described. The building blocks allow coupling a *variety* of processing modules in dynamic manner, which meets the flexibility requirement. The encapsulating of building blocks into containers enables designers not only deploying pipelines on the cloud, but also to deploy pipelines in different types of infrastructure, which meets the elasticity requirement and resource underutilization when processing large *volumes* of files. A deployment scheme called divide and containerize deploys pipelines in parallel on the cloud to improve the *processing* of repository files, which meets the efficiency requirement.

A prototype based on the proposed method was developed and evaluated in different infrastructures through different experimentation scenarios. The evaluation revealed the feasibility of applying cloud processing pipelines based on building blocks encapsulated containers, which meet the requirements of *variety*, *volume* and *velocity* required for the management of repositories of large files.

Nomenclatura

VC	Contenedor Virtual
CI	Imagen de contenedor
VM	Máquina Virtual
VMM	Administrador de máquinas virtuales
LXC	Contenedores Linux
NIST	Instituto Nacional de Estándares y Tecnología
SaaS	Software como servicio
IaaS	Infraestructura como servicio
PaaS	Plataforma como servicio
BB	Bloque de construcción
PU	Unidad de procesamiento
PBB	Bloque de construcción paralelizado
API	Interfaz de programación de aplicaciones
IPC	Comunicación entre procesos
POSIX	Interfaz de sistema operativo portátil de unix
SMS	Segmento de memoria compartida

1

Introducción

En este capítulo se describen los antecedentes, la motivación y el problema que se abordó en esta tesis. También se presenta la pregunta de investigación que se desea resolver y se definen los objetivos que se plantearon para la tesis. Finalmente, se muestra la metodología de investigación seguida durante el desarrollo del proyecto.

1.1 Antecedentes y motivación

El universo digital esta compuesto por el promedio de los bits digitales creados, replicados y consumidos por año. La producción de contenido digital (documentos, archivos de audio, imágenes, etc.) en los últimos diez años ha incrementado en promedio un 25 % adicional por año [10, 29]. Estudios que analizan el contenido del universo digital realizado entre los años 2005 y 2012 proyectan que para el año 2020 los datos digitales crecerán en un factor de 300 en comparación al año 2005, creciendo de 130 exabytes (EB) a 40,000 EB [30]. La corporación internacional de datos [30] proyecta que para el 2020 el total de datos almacenados en la nube corresponderá al 13 % (5208 EB) del

universo digital y el 24 % (9788 EB) será procesado o transmitido por la nube, pero sin almacenarlo en ésta.

En los últimos cinco años, las organizaciones han adoptado por el modelo de servicio llamado cómputo en la nube para almacenar sus datos y hacer frente al crecimiento de la producción de datos nuevos. Esta migración ha reducido los costos de la adquisición y mantenimiento de la infraestructura requerida para esta tarea por dicha organización [46]. El almacenamiento y transporte de datos en la nube permite delegar la responsabilidad de almacenamiento a una entidad externa llamada proveedor de servicios, lo cual permite obtener la disponibilidad de los datos accediendo a estos a través de internet en cualquier lugar en cualquier momento.

El almacenamiento de datos en la nube es proporcionado por empresas tales como Google, Microsft y Amazon, mediante aplicaciones como Google Drive [35], OneDrive [47] y Amazon S3 [3] respectivamente. Además de estos servicios de almacenamiento, existen aplicaciones y soluciones de *software* que utilizan redes de entregas de contenido a través de la nube, los cuales generan grandes volúmenes de datos que deben de estar disponibles para los consumidores que requieran acceder a estos (e.g.: SkyCDs [34] , Sacbe [33], DepSky [8], CloudS [57] y CYRUS [16]).

Para el despliegue de aplicaciones en la nube se realiza su encapsulación dentro de *máquinas virtuales* que permiten el despliegue de múltiples maquinas virtuales en un solo equipo de cómputo. Gracias a la virtualización de *hardware* se evita que dos máquinas virtuales consuman los mismos recursos brindando la característica de aislamiento. Sin embargo, esta característica implica que pueden estar máquinas virtuales sin trabajar con recursos reservados mientras que otras que si están trabajando requieren más recursos de los que tienen acceso. Las máquinas virtuales son gestionadas por un monitor de maquinas virtuales encargado de iniciarlas, asignarles los recursos y detenerlas.

En los últimos tres años se ha optando por una tecnología que permite la encapsulación de aplicaciones y servicios así como sus dependencias llamada *contenedor virtual* [9, 20]. El contenedor virtual no utiliza la virtualización de *hardware*, lo que permite que todos los contenedores accedan todos los recursos del equipo si los necesita[1] lo que permite que se ocupen los recursos de *hardware* disponibles bajo demanda. Esta tecnología delega la asignación de recursos al *kernel* del sistema operativo anfitrión el cual permite acceder a sus recursos de *hardware* asignados ayudando a reducir los tiempos de procesamiento.

En los escenarios de almacenamiento y soluciones para la distribución de contenido a través de la nube se produce un efecto de acumulación de grandes volúmenes de datos digitales el orden de EB. Esta acumulación genera un nuevo fenómeno llamado *Big Data* [15, 22], este fenómeno presenta varias necesidades para realizar la gestión de los datos tales como: la administración de recursos, mecanismos para extracción de conocimiento mediante el análisis de texto y técnicas para el procesamiento de datos.

Para la administración de recursos existen soluciones que permiten realizar el almacenamiento de los datos con tolerancia a fallos [51] y el uso de sistemas distribuidos para la modularización de aplicaciones y agilización del procesamiento [59].

Actualmente para el análisis de grandes volúmenes de datos existen alternativas que funcionan en sistemas distribuidos como *Hadoop* que procesa grandes conjuntos de datos a través de clusters de computadoras usando modelos simples de programación [28] y *Spark* que es un motor para el procesamiento de datos a gran escala [27] basado en *Hadoop* con la opción de trabajar con la memoria. Ambos permiten realizar minería de texto para la extracción de datos e información.

[1] A menos que se les delimite el uso del *hardware* a unos componentes en especifico

Las soluciones para el procesamiento de volúmenes de datos en el orden de GB y EB proponen arquitecturas de *software* como:

- Tuberías de procesamiento: Son una serie de procesos (algoritmos) llamados filtros de procesamiento que reciben un dato de entrada los procesan y devuelven al filtro siguiente los cuales están interconectados desde un origen hasta un destino [36] formando un flujo continuo de datos.

- Micro-servicio: También conocidos como arquitectura de micro-servicio, es un estilo arquitectónico que estructura una aplicación como una colección de servicios que no tienen recursos compartidos entre estos y si uno falla los demás siguen funcionando [2, 23].

Algunas soluciones que implementan estas arquitecturas de *software* son: la arquitectura SkyCDs [34] y la de Sacbe [33] que toman como referencia los *pipelines* propuestos por Douglas McIlroy e implementados por el sistema operativo Unix [53] para la realización del procesamiento de los datos mediante entidades de procesamiento independientes que pueden transferir mensajes y contenidos entre ellas también conocidas como filtros de procesamiento. Estas soluciones se encuentran implementadas en arquitecturas extremo a extremo del lado del cliente y consumidor que se encuentran en ubicaciones diferentes (equipos de cómputo o carpetas). Estas propuestas tienen como objetivo la utilización de filtros de procesamiento para la compresión de datos, aseguramiento de la información mediante técnicas de cifrado y herramientas para la codificación de datos que permiten ofrecer las características de distribución y tolerancia a fallos[2]. Estos filtros fueron desarrollados considerando las siguientes características:

- *Velocidad*: El tiempo de ejecución requerido para la finalización de los filtros de procesamiento desde el filtro origen hasta el filtro destino.

[2]Recuperando el contenido distribuido aún y cuando algunos de los nodos de almacenamiento no se encuentren disponibles (el nivel de tolerancia a fallos es representado por el número de nodos que pueden estar fuera de linea sin afectar la obtención del archivo)

- *Variedad*: Los diferentes procesos realizados por los filtros en la tubería.

- *Volumen*: La cantidad de datos que pueden ser procesados por los filtros.

En la Figura 1.1 se muestran la situación actual del *Big Data* así como los enfoques que se están utilizando para abordar el problema y las soluciones estudiadas para cada uno. Se hace un énfasis (relleno de un color más oscuro) en el procesamiento de datos utilizando tuberías de procesamiento debido a que este enfoque con esa solución serán abordados a lo largo de este trabajo de tesis.

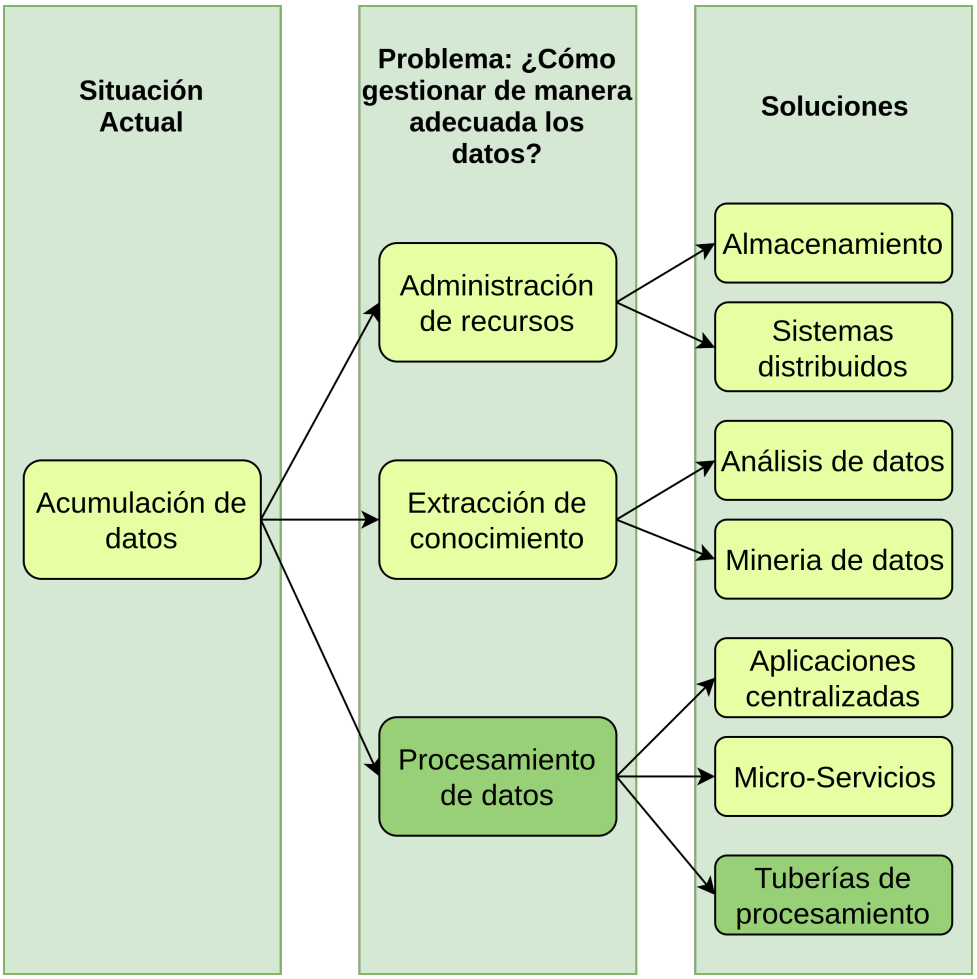


Figura 1.1: Problemática y enfoques para abordar el problema.

1.2 Planteamiento del problema

Algunas organizaciones que trabajan con datos de acceso restringido cubriendo restricciones de manejo impuestas por organizaciones y/o instancias gubernamentales deben aplicar procesos (filtros) adicionales para garantizar la confidencialidad, disponibilidad y autenticidad de los datos debido a que, el almacenamiento de los datos en entornos de nube pública implica darle al proveedor de servicios el completo control sobre los datos que en este se almacenan comprometiendo la confiabilidad (los datos pueden ser modificados) y confidencialidad (el proveedor de servicios tiene acceso total a los datos almacenados en este). En el caso de nube privada la organización es la que administra los recursos y los datos almacenados pero aún así deben de ser asegurados para evitar robos de información.

Los filtros usados para el aseguramiento de los datos forman parte de tuberías de procesamiento donde solo el resultado del último filtro es almacenado. Este tipo de escenarios son mayormente abordados por soluciones que encapsulan las tuberías de procesamiento en soluciones extremo a extremo como aplicaciones cliente-servidor y cliente-consumidor. Sin embargo, estas soluciones se ven restringidas severamente por los recursos limitados de los equipos de cómputo disponibles en dichos extremos ocasionando una saturación de los recursos disponibles, incrementando los tiempos de respuesta percibidos por el usuario final debido al tiempo de espera para la liberación y utilización de recursos. Las soluciones mencionadas en la sección 1.1 aprovechan el almacenamiento en la nube sin utilizar los recursos de cómputo (CPU, memoria) que este ofrece.

Los componentes involucrados en el procesamiento, distribución y almacenamiento de dato son:

- *Fuente de datos*: Es el origen, encargado de producir o generar los datos. Puede ser un cliente que emite los datos a través de las interfaces de salida a una tubería de procesamiento o enviar el contenido sin procesar para su almacenamiento.

- *Almacén de datos*: Espacio reservado para el almacenamiento de los datos producidos por el origen o el resultado de las tuberías de procesamiento del lado del cliente (quien carga los datos). Este componente puede ser un servicio de almacenamiento en la nube o en un equipo del cliente con espacio de almacenamiento disponible.
- *Consumidores*: Entidades que solicitan los datos colocados en el almacén de datos o a la tubería encargada de obtener los datos.
- *Filtros de procesamiento*: Procesos que reciben un dato de entrada, realizan alguna modificación o proceso a estos datos y lo devuelven al siguiente filtro, almacén o consumidor de datos que los solicite.
- *Tuberías de procesamiento*: Es un conjunto de filtros de procesamiento interconectados a través de sus interfaces de comunicación (sistema de archivos, memoria o red) que tienen como objetivo realizar múltiples procesos a los datos de entradas desde su origen hasta el almacén o consumidor.

Dichos componentes forman parte del *ciclo de vida del procesamiento de los datos* que está conformado por la tubería de procesamiento a través de la cual los datos son procesados desde el origen hasta el almacén o el consumidor.

En la Figura 1.2 se muestra un ejemplo de tuberías de procesamiento que involucran los componentes mencionados anteriormente. Estas tuberías realizan los procesos encargados del aseguramiento de datos y descarga de estos por los consumidores. El ejemplo cuenta k fuentes de datos, que del lado del cliente (en el mismo origen) realizan los procesos de cifrado (E), compresión (C) y marcado (M) antes de colocar los datos en el almacén. Por otro lado, para la descarga de los datos por l consumidores donde cada uno debe realizar los procesos inversos del aseguramiento comenzando por la verificación del marcado (VM), la descompresión (U) y el descifrado (D) de los datos para finalmente almacenarlos en el consumidor.

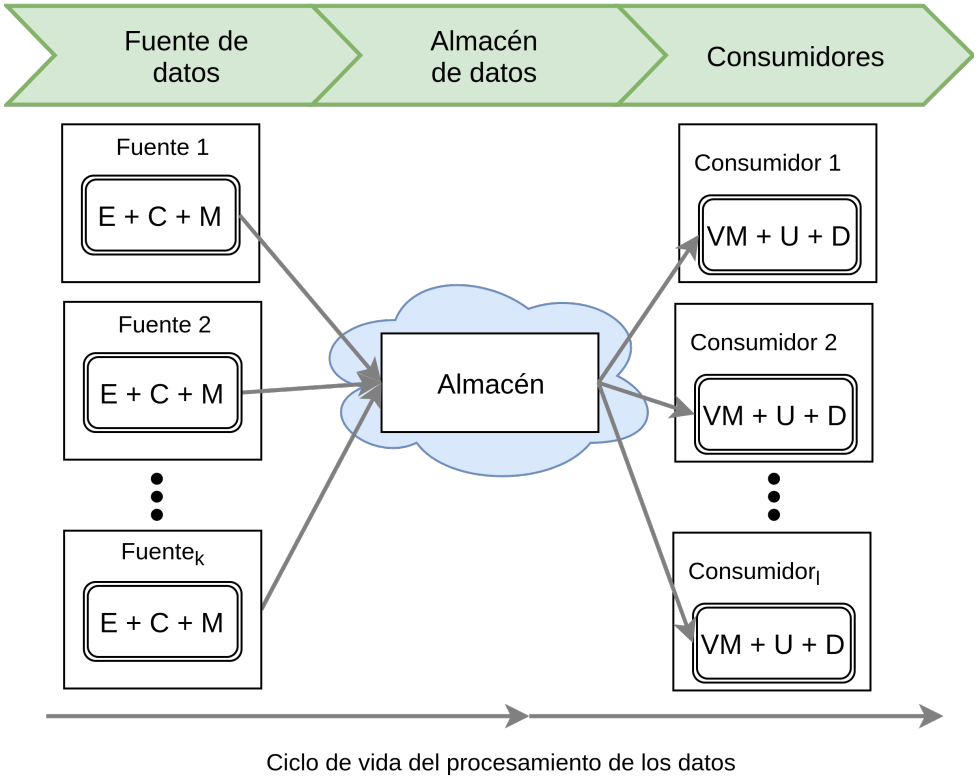


Figura 1.2: Ejemplo de tuberías de procesamiento para asegurar, almacenar, descargar y verificar datos.

Las áreas de oportunidad en este tipo de escenarios es que las tuberías de procesamiento se encuentran en ambos extremos (fuente y consumidor) limitándose a los recursos de *hardware* que estos disponen y no en la nube donde existen mejores prestaciones de *hardware*[3]. También se debe de considerar el tiempo que le toma a las tuberías de procesamiento realizar el procesamiento y transporte de los datos desde el origen al almacén (carga) y del almacén al consumidor (descarga) para archivos de gran tamaño (hasta un GB por archivo), el tiempo de respuesta (conformado por el tiempo de transporte, procesamiento y almacenamiento) incrementa en función del tamaño del archivo.

Considerando que una de las características de las tuberías de procesamiento es el añadir/remover

[3]Tantas como la organización pueda y desee pagar.

filtros es necesario generar tuberías de procesamiento configurables, es decir, que puedan ser modificadas para seguir una nueva trayectoria dependiendo del proceso que se desea realizar o que un filtro pertenezca a varias tuberías.

1.3 Pregunta de investigación

En esta tesis se busca i) Minimizar los problemas de saturación de recursos de cómputo delegando la responsabilidad de procesamiento a VCs con tuberías de procesamiento embebidas desplegados en la nube o en equipos con mayores prestaciones de *hardware* para reducir los tiempos de espera para realizar el procesamiento de los datos. ii) El almacenamiento de datos codificados en la nube mediante la codificación basada en corrección de errores haciendo que el proveedor de servicios no pueda interpretar el contenido que tiene almacenado y brindando tolerancia a fallos. iii) Disminuir los altos tiempos de respuesta que perciben usuarios interesados en adquirir, procesar y consumir grandes cantidades de datos al utilizar múltiples tuberías de procesamiento en paralelo. Tomando como referencia estas áreas de oportunidad surge la pregunta de investigación motivo de este trabajo de tesis.

¿Pueden las estructuras de tuberías de procesamiento basadas en flujos de datos continuos hacer frente al desafío de procesamiento de datos en la nube con las características de *velocidad, variedad* y *volumen*?

1.4 Objetivos

1.4.1 Objetivo general

Obtener un método para la construcción de tuberías configurables para el procesamiento de datos en la nube que permita manejar características de flexibilidad en términos de variedad de procesos, elasticidad en términos de manejo de grandes *volúmenes* de datos y eficiencia en términos de *velocidad* de procesamiento.

1.4.2 Objetivos específicos

1. Obtener una arquitectura de *software* para la construcción de tuberías de procesamiento basada en contenedores que reduzca el tiempo de respuesta observado mejorando la experiencia de servicio del usuario final.

2. Definir un esquema de despliegue de contenedores de la arquitectura propuesta que permita la ejecución en paralelo de tuberías de procesamiento para mejorar la utilización del almacenamiento de datos.

3. Obtener un planificador para diseñar y desplegar tuberías de procesamiento mediante la interconexión de contenedores que mejoren los tiempos de servicio.

1.5 Metodología

La metodología seguida para alcanzar los objetivos planteados y llevar a cabo este trabajo de tesis se ha dividido en las siguientes etapas:

- Introducción y estado del arte.

  En esta etapa se presenta la información que sustenta la introducción y estado del arte, mediante la recolección y búsqueda de bibliografía de propuestas similares. Además, se presenta la definición de los objetivos general y específicos, la pregunta de investigación.

- Diseño de la solución.

  En esta etapa se consiguió elaborar la carpeta de diseño con los componentes del método propuesto. Los diseños propuestos son:

  - Una arquitectura de *software* para la construcción de tuberías de procesamiento basada en contenedores.

  - Un constructor de imágenes de contenedor.
  - Un planificador de conexiones que permite establecer las tuberías de procesamiento mediante la interconexión de contenedores.
  - Un esquema que permite la segmentación de los datos de entrada en el origen y enviarlo de forma paralela a los contenedores siguientes. Este esquema lleva por nombre *Divide y Encapsula*.
  - Un mecanismo de lanzamiento de contenedores para la arquitectura de *software* propuesta.

- Implementación de la solución y evaluación experimental.

  Esta etapa se dividió en dos sub etapas:

  - Implementación de los componentes de la carpeta de diseño y la realización de una prueba de concepto.

    Durante esta sub etapa se realizó la implementación de la carpeta de diseño y utilización de sus componentes para realizar una prueba de concepto y el desarrollo de un prototipo funcional. Además, se realizaron las pruebas requeridas para evaluar la funcionalidad de los componentes implementados.

  - Despliegue del prototipo y evaluación experimental.

    Durante esta sub etapa se realizó el diseño de la evaluación experimental basada en prototipos, el despliegue del prototipo en los escenarios planteados en la evaluación: codificación y decodificación de archivos. Ambos escenarios fueron probados realizando variaciones en los tamaños de los archivos de entrada así como del número de tuberías utilizadas para el procesamiento.

- Evaluación y documentación de resultados.

  Esta etapa también se dividió en dos sub etapas:

- Análisis de resultados.

  En esta subetapa se realizó un análisis de los resultados recolectados. Además, la realización de pruebas complementarias y su análisis.

- Finalización del documento de tesis.

  En esta sub etapa final se redactaron los capítulos de la tesis relacionados con la implementación, prueba, análisis y conclusiones del proyecto. También se incluyeron las áreas de oportunidad y el trabajo futuro que se identificaron para este trabajo de tesis.

1.6 Organización de la tesis

Esta tesis está conformada de 6 capítulos, los cuales se encuentran organizados de la siguiente manera:

El Capítulo 1 presenta el contexto de este trabajo de tesis. En el Capítulo 2 se realiza una descripción del fundamento teórico requerido para el desarrollo de esta tesis. Además, se presenta una revisión del estado del arte respecto a arquitecturas de *software* para el procesamiento de datos, implementaciones de dichas arquitecturas y el uso de contenedores virtuales para la encapsulación de aplicaciones. En el Capítulo 3 se describe el diseño del método propuesto así como sus componentes. En el Capítulo 4 se muestra la implementación de los componentes diseñados para alcanzar los objetivos propuestos en el Capítulo 1. En el Capítulo 5 se describe la metodología a seguir para realizar la evaluación experimental. Además, se discuten los resultados obtenidos de la experimentación realizada. Finalmente, en el Capítulo 6 se muestran las conclusiones obtenidas durante el desarrollo de este trabajo concluyendo este documento de tesis.

2

Marco teórico y estado del arte

En este capítulo se presentan las distintas tecnologías y conceptos involucrados en el desarrollo de este trabajo de tesis (marco teórico) así como algunas propuestas existentes en la literatura para el diseño de arquitecturas de software que permiten el procesamiento y análisis de datos en diferentes infraestructuras como cliente, servidor, cluster de computadoras y nube (estado del arte).

2.1 Marco teórico

Los conceptos clave para entender los componentes y elementos utilizados durante este trabajo de tesis son descritos en las siguientes secciones.

2.1.1 Sistemas monolíticos

Es el sistema donde sus componentes son separados unicamente de forma lógica pero son implementados como uno solo dando como resultado un programa enorme (en cuanto al tamaño

del *software* final y número de procesos en su interior). Este enfoque hace que sea difícil reemplazar o adaptar un componente sin afectar a todo el sistema debido a la dependencia que existe entre ellos. Esta característica hace que los sistema monolíticos sean cerrados a modificaciones y/o actualizaciones una vez puestos en producción (cuando ya están en uso por el cliente) en vez de estar abiertos para el mantenimiento y corrección de errores [59].

2.1.2 Tuberías de procesamiento

Una tubería de procesamiento es una sucesión encadenada de f filtros de procesamiento conectados de forma adyacente a través de sus interfaces de comunicación desde un origen o almacén de datos formando un flujo de datos continuo hasta un destino, almacén o consumidor, donde el valor de $f \geq 1$ [36], [13]. Las tuberías de procesamiento forman un flujo de datos continuo como se muestra en la Figura 2.1.

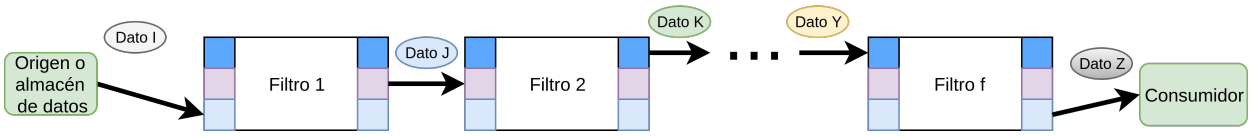


Figura 2.1: Ejemplo de tubería de procesamiento.

Las definiciones de filtros de procesamiento e interfaces de comunicación utilizadas por las tuberías de procesamiento son:

- *Filtro de procesamiento*: Unidad de procesamiento que recibe un dato de entrada (Dato I) desde un origen o almacén de datos, aplica un proceso (un algoritmo), modifica los datos de entrada de acuerdo al proceso y devuelve como resultado el dato modificado (Dato J) a algún consumidor u otro filtro de procesamiento como se muestra en la Figura 2.2

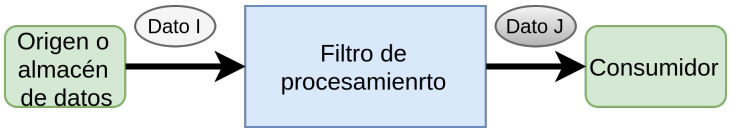


Figura 2.2: Ejemplo de filtro de procesamiento.

- *Interfaz de comunicación*: Mecanismo que permite la interconexión entre diferentes entidades como los filtros de procesamiento y los componentes del método propuesto descritos en la sección 3.1 habilitando las interfaces de E/S del sistema operativo anfitrión. Las interfaces de comunicación establecidas son: red, sistema de archivos y memoria compartida. En la Figura 2.3 se muestra un filtro de procesamiento con las tres interfaces de comunicación habilitadas.

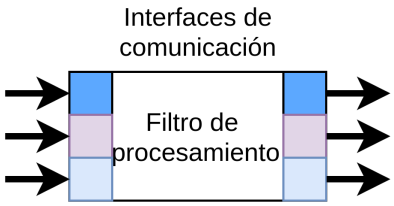


Figura 2.3: Ejemplo de interfaces de comunicación.

Cada filtro de procesamiento puede ser agregado, modificado, reemplazado o eliminado de una tubería sin afectar el funcionamiento de los demás filtros de la tubería. Por lo tanto, las tuberías de procesamiento son una alternativa a los sistemas monolíticos en escenarios donde se requiera la característica de *variedad* de procesos. El uso de diversas interfaces de comunicación permiten agilizar el envío de datos entre filtros (al usar la memoria) consiguiendo incrementar la *velocidad* de procesamiento reflejada al disminuir los tiempos de respuesta de la tubería como se muestra en la sección 5.3.

2.1.3 Redes de entrega de contenidos

La distribución de contenido a través de la red es indispensable para el desarrollo de la arquitectura propuesta ya que permite realizar el transporte y compartición de documentos y contenido digital a múltiples usuarios. Esta distribución es requerida para realizar la el transporte de contenidos procesados o sin procesar entre un origen y un consumidor que forma parte de nuestra problemática. En la literatura se ha trabajado con la distribución de contenidos mediante el uso de redes de entrega de contenido (CDN, por sus siglas en inglés).

Una CDN es una red auto-organizada de nodos de distribución de contenidos que están dispuestos para la entrega eficiente de los contenidos digitales (por ejemplo, contenido *web*, *streaming media* y aplicaciones) en nombre de los proveedores de contenidos de terceros. Una petición de un usuario final que solicita un contenido dado se dirige a la "mejor" réplica, en la que "mejor" por lo general significa que el elemento se entrega al cliente de forma rápida en comparación con el tiempo que le llevaría a buscarlo desde el servidor de origen del proveedor del contenido. Una entidad que proporciona una CDN se refiere a veces como un proveedor de servicios de red de entrega de contenido [14, 49, 60].

2.1.4 Cómputo en la nube

El cómputo en la nube de acuerdo al Instituto Nacional de Estándares y Tecnología (NIST, por sus siglas en inglés) es "un modelo que permite acceso ubicuo conveniente bajo demanda a un conjunto de recursos de cómputo configurables (e.g., redes, servidores, almacenamiento, aplicaciones, y servicios) que pueden ser provisionados y liberados rápidamente con el mínimo esfuerzo de gestión o interacción del proveedor de servicio". Este modelo se compone de cinco características esenciales (servicio bajo demanda, amplio acceso a la red, puesta en común de recursos, rápida elasticidad y servicio medido), tres modelos de servicio *software* como servicio (SaaS), plataforma como servicio (PaaS) e infraestructura como servicio (IaaS) y cuatro plataformas de despliegue (pública, privada, híbrida y comunitaria) [46].

Este modelo permite realizar el consumo de estos servicios utilizando internet como medio de comunicación. El *cloud computing* se apoya de una infraestructura tecnológica dinámica que se caracteriza por un alto grado de automatización, una rápida movilización de recursos, una capacidad de atender una demanda variable así como la virtualización avanzada a un precio flexible en función de su consumo.

2.1.5 Virtualización para la nube

La virtualización es la simulación del *software* o *hardware* sobre el que se ejecuta otro software. Es una forma de particionar de forma lógica un equipo físico en distintos entornos simulados. Dichos entornos simulados se les denomina **máquinas virtuales** (VMs, por sus siglas en inglés) [37, 55].

Hay dos tipos de tecnologías para la virtualización de servidores que son comunes en entornos de virtualización de centros de datos: virtualización a nivel de *hardware* y la virtualización a nivel de *sistema operativo*.

- Virtualización a nivel de *hardware*

  La virtualización del *hardware* implica virtualizar el *hardware* en un servidor y crear VMs que proporcionan la abstracción de una máquina física. La virtualización de *hardware* implica ejecutar un hipervisor, también denominado monitor de máquina virtual directamente sobre el servidor, es decir sin la necesidad de instalar previamente un sistema operativo (bare metal server). El hipervisor emula hardware virtual como la CPU, memoria, dispositivos de E/S de red para cada VM. Cada VM, a continuación, ejecuta un sistema operativo independiente y las aplicaciones en la parte superior de ese sistema operativo. El hipervisor también es responsable de multiplexar los recursos físicos subyacentes a través de las VMs residentes [56].

- Virtualización a nivel de sistema operativo

  La virtualización a nivel de sistema operativo también conocida como virtualización basada en contenedores implica la virtualización del *kernel* del sistema operativo en lugar del *hardware* físico. En la virtualización de nivel sistema operativo las VMs se denominan **contenedores virtuales** (VCs, por sus siglas en inglés). Cada VC encapsula un grupo de procesos que están aislados de otros VCs o procesos en el sistema. El *kernel* del sistema operativo es responsable de implementar la abstracción del VC, además, es el encargado de organizar la compartición de recursos de hardware tales como: CPU, memoria y la red de Entrada/Salida (E/S) a cada VC y

también puede proporcionar aislamiento al sistema de archivos [56], [9], [4]. Los elementos de la virtualización basado en contenedores son: Imagen de contenedor virtual (CI, por sus siglas en inglés) y el VC.

Una CI y es un modelo de solo lectura que es usado para crear contenedores de la plataforma de despliegue de contenedores Docker [50] descrita más adelante en la sección 2.1.7.1. Una imagen es una colección ordenada de cambios en el sistema de archivos raíz y los parámetros de ejecución correspondientes para su uso en el tiempo de ejecución de un contenedor. Una imagen típicamente contiene una unión de sistemas de archivos en capas apilados uno encima del otro. Una imagen no tiene estado y nunca cambia.

Un VC es una instancia en tiempo de ejecución de una CI. Un VC está compuesto de: una CI, un entorno de ejecución y un conjunto estándar de ejecuciones requeridos por las aplicaciones definidas en el entorno [9], [5].

En la Figura 2.4 muestran los elementos de la virtualización a nivel de *hardware* (a) y a nivel de sistema operativo (b). Se puede observar que al utilizar los VCs se reduce el número de elementos involucrados en el proceso de virtualización, esto permite su reducción en tamaño. Además, al no usar *hardware* virtualizado los contenedores pueden acceder a los recursos del sistema operativo huésped incrementando el desempeño de los CVs.

2.1.6 Diferencias entre máquinas virtuales y contenedores virtuales

El análisis de utilización de *Docker* para el despliegue de contenedores es abordado en [52]. En este trabajo se muestra que la utilización de aplicaciones encapsuladas en contenedores *Docker*

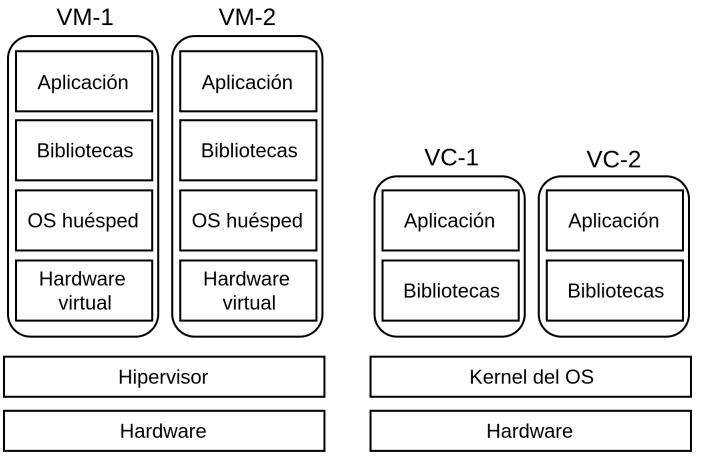


Figura 2.4: Tipos de virtualización.

permite tener acceso a más del 90 % de los recursos disponibles por el sistema operativo anfitrión y el desempeño de una aplicación encapsulada en VC consumen tiempos similares que utilizarla sin encapsular.

Las diferencias entre los VMs y VCs mostradas en [17, 44, 45] son: el tamaño de la imagen por utilizar (las imágenes de contenedor son de menor tamaño), el tiempo de despliegue de la imagen en entornos de nube (3 segundos para VCs y [895–918] segundos en promedio para una VM sin pre-configurar y [335–340] segundos para una VM pre-configurada), el *overhead* generado por el hipervisor de VM [39] y el tiempo de acceso a las interfaces de E/S.

Estas características muestran que los contenedores *Docker* son ligeros (tamaño en MB), altamente portables (son multiplataforma) y escalables (permiten la ejecución de múltiples VCs en base a una CI permitiendo realizar el mismo proceso en diferentes instancias). Por lo tanto, se seleccionaron a los VCs como la herramienta de virtualización para encapsular los componentes y aplicaciones desarrolladas durante este trabajo de tesis.

2.1.7 Plataformas de contenedores, virtualización y de gestión de recursos virtualizados

Para la creación, ejecución y administración de VCs es necesario la instalación de una plataforma de despliegue y algunos frameworks de administración de recursos, como los que se listan a continuación.

Docker

Docker es una plataforma basada en contenedores que proporciona una forma de ejecutar aplicaciones con sus dependencias de software y bibliotecas requeridas encapsuldas de forma segura en un VC. Debido a que la aplicación se puede ejecutar con el entorno que espera en la CI que la contiene, el despliegue de un VC se puede realizar con un solo comando. El despliegue de los VCs es portable debido a que las CIs están listas para ejecutarse en cualquier entorno compatible con la plataforma *Docker* como: Linix, Windows y Mac [20]

Debido a que los contenedores son ligeros y se ejecutan sin la carga adicional de un hipervisor, se pueden ejecutar varias aplicaciones que dependen de diferentes bibliotecas y entornos en un solo equipo administradas por el mismo *kernel*. Cada VC se encuentra aislado y no interfiere con procesos de los demás VC o del sistema operativo anfitrión. Esto le permite sacar más provecho de su *hardware* cambiando la unidad de escala de su aplicación de una VM o máquina física a una instancia de VC [20], [39].

Docker proporciona una forma sistemática de automatizar y eficientar el despliegue de aplicaciones Linux dentro de un VC. Básicamente, *Docker* extiende los contenedores Linux [41] con una interfaz de programación de aplicaciones (API, por sus siglas en inglés) a nivel de kernel y a nivel de aplicación que permiten la ejecución de procesos aislando: CPU, memoria, interfaces

de E/S, red, etc. *Docker* también utiliza espacios de nombres para aislar completamente la vista de una aplicación del entorno operativo subyacente, incluidos los árboles de proceso, la red, los identificadores de usuario y los sistemas de archivos [7, 24].

Docker cuenta con su propia línea de comando que permiten el despliegue de contenedores. La línea de comandos de *Docker* puede ser consultada en [18]. Los comandos considerados importantes para el despliegue de los contenedores son:

1. `-v`: Enlaza un directorio del sistema de archivos del sistema operativo anfitrión que ejecuta el contenedor. Al utilizar este comando se debe de tomar en cuenta que se requieren la ruta de dos directorios: El directorio del sistema de archivos del anfitrión el cual es externo al contenedor y el directorio que se encuentra dentro del contenedor. Estos dos directorios son enlazados mediante un *volume* de *Docker* que hará que las modificaciones en cualquiera de estos dos directorios pueda modificar el contenido del *volume* compartido [19].

2. `-ti`: Después de ejecutar el contenedor concede el acceso a la línea de comandos interactiva del VC lo que permite trabajar sobre el VC.

3. `-d`: Ejecuta el contenedor en segundo.

4. `-p`: Habilita uno o múltiples puertos de escucha de red para acceder al VC.

5. `-name`: Asigna de un nombre al VC.

Algunas recomendaciones para la utilización de *Docker* para el desarrollo de investigación reproducible mencionadas en [11] son:

- Usar VCs *Docker* durante el desarrollo del proyecto. Permite generar una o múltiples versiones de CIs para cada etapa del desarrollo dando como resultado diferentes puntos de restauración en caso de que se dañe una CI durante la adición de una nueva dependencia de software.

- Usar y proveer CIs apropiadas. Utilizar y generar nuevas CIs que contengan solo las dependencias y bibliotecas de software necesarias para el VC que se desea desarrollar.

- Compartir las CIs Docker y Dockerfiles. El repositorio *Docker Hub* [38] permite la compartición de CIs y Dockerfiles previamente generados y probados por diferentes usuarios agilizando el desarrollo de componentes de software. Por ejemplo: se pueden descargar CIs con un sistema operativo Linux que contenga un servidor apache y mysql funcional listo para su ejecución.

Kubernetes

Kubernetes es una plataforma de código abierto para automatizar el despliegue, la ampliación y las operaciones de los contenedores de aplicaciones entre *clusters* de *hosts*, proporcionando una infraestructura centrada en contenedores compatible con *contenedores Docker* [1, 7, 12] y otras tecnologías de virtualización.

Apache Mesos

Apache Mesos es una plataforma para compartir recursos de grano fino en el centro de datos abstrayendo la CPU, la memoria, el almacenamiento y otros recursos de computación lejos de las máquinas (físicas o virtuales), permitiendo que los sistemas distribuidos elásticos y tolerantes a fallos sean fácilmente construidos y ejecutados con eficacia.

Apache Mesos proporciona varios mecanismos de aislamiento en los esclavos para realizar diferentes tareas. La asignación de recursos a un *framework*/tarea o usuario no debe tener ningún efecto no deseado en los trabajos en ejecución. Permite el despliegue de VCs que actúan como VMs ligeras, proporcionando el mecanismo de aislamiento necesario sin la sobrecarga de las VMs. Utilizando VCs, podemos limitar la cantidad de recursos a los que puede acceder el proceso y todos sus procesos secundarios [42]. *Apache Mesos* permite el despliegue de contenedores Docker.

Docker swarm

Swarm permite gestionar un cluster de servidores *Docker*. Exporta las API estándar de *Docker* y permite gestionar el ordenamiento de las tareas y la asignación de recursos por contenedor dentro del pool de recursos de las máquinas físicas. Su interés reside en que permite gestionar el cluster como una única máquina *Docker*.

Las máquinas *Docker* también llamadas nodos se dividen en dos tipos: administrador y trabajador. Los nodos administradores se encargan de gestionar el cluster y distribuyen las tareas o *tasks* (unidades básicas de trabajo) mientras que los nodos trabajadores reciben las tareas y las ejecutan. Los administradores asignan tareas a los nodos trabajadores de acuerdo al número de réplicas definidas por el servicio. Una vez que la tarea es asignada a un nodo trabajador ya no se puede mover a otro, tan sólo puede ejecutarse o morir. Ante la caída de una tarea, *Swarm* es capaz de crear otra similar en esa u otro nodo la genera para cumplir con el número de réplicas definido.

Debido a que *Docker Swarm* sirve a la API de *Docker* estándar, cualquier herramienta que ya se comunique con un *daemon de Docker* puede utilizar *Swarm* para escalar de forma transparente a varios equipos. Las herramientas soportadas incluyen, pero no se limitan a: *Dokku, Docker Compose, máquinas Docker, Jenkins* y el cliente *Docker*.

Al igual que otros proyectos *Docker*, *Docker Swarm* sigue el principio "*swap, plug and play*". A medida que se asienta el desarrollo inicial, se desarrollará una API para habilitar *backends* conectables. Esto significa que se puede intercambiar el *backend* de programación de *Docker Swarm* por un *backend* que se prefiera por el desarrollador. El diseño *Swappable* de *Swarm* proporciona una experiencia sin complicaciones para la mayoría de los casos de uso, y permite que los despliegues de producción a gran escala se intercambien para *backends* más potentes como Mesos [21].

2.2 Estado del arte

En la literatura se ha abordado la arquitectura de *software* de *tuberías de procesamiento de Unix* para la validación de integridad de datos, aplicación de confidencialidad y almacenamiento de archivos usando el sistema de archivos del sistema operativo [36]. También hay propuestas para generar tuberías de procesamiento con distintos procesos colocados del lado del cliente para la publicación y suscripción de contenido. Además, existen trabajos sobre el uso de micro-servicios para evitar el uso de sistemas monolítico haciendo uso de la modularización de componentes así como propuestas de arquitecturas de software basada en micro-aplicaciones para el desarrollo de servicios extremo a extremo para almacenamiento de archivos en la nube [33, 34, 40].

Por otra parte para el despliegue de aplicaciones en entornos de nube, local o distribuido se requiere que estas puedan ser trasladadas de una infraestructura a otra sin que su funcionamiento se vea afectado. Esta característica es conseguida al utilizar VCs que permiten la encapsulación de aplicaciones con sus dependencias y bibliotecas que pueden ser almacenadas, trasladadas y ejecutadas en diferentes entornos con el requerimiento de tener la plataforma de contenerización utilizado para su creación. A continuación se describen las propuestas mencionadas.

2.2.1 Tuberías de procesamiento para almacenamiento usando el sistema de archivos del sistema operativo

En [36] se presentan los resultados de evaluar el desempeño de las tuberías de Unix contra su implementación de tuberías desarrollada usando C++, con el objetivo de mostrar que una implementación distinta a las tuberías de Unix *shell* por medio de las herramientas convencionales proporcionadas por el sistema operativo puede aumentar el desempeño.

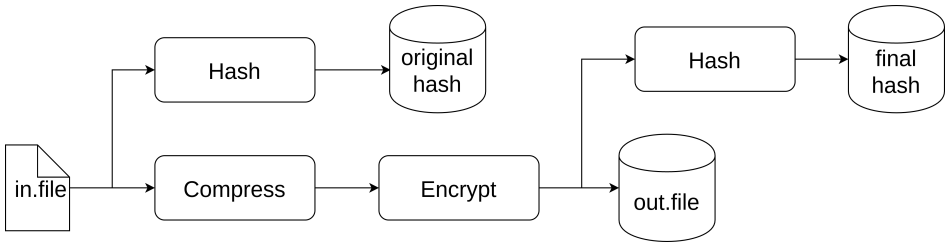


Figura 2.5: Tubería de procesamiento utilizada en [36].

En la Figura 2.5 se muestran las etapas de procesamiento y resultados involucrados en la creación de la tubería de procesamiento evaluada. Para la selección de los algoritmos de hash, compresión y cifrado por usar en la implementación final, se realizaron pruebas con distintos algoritmos y configuraciones (*single thread*, *multithreading*) con una implementación de tuberías de Unix *shell*. Se implementó un Hash-Compress-Encrypt *pipeline*, con los algoritmos que presentaron un mejor rendimiento en MB/s (Blake2sp-LZ4-AES128-*pipeline*) y se realizaron pruebas con distinto número de *pipelines*. Los resultados muestran un incremento lineal del rendimiento en MB/s de la cantidad de datos procesados mientras mayor sea el número de *pipelines* utilizados. La diferencia notable entre la carga (*load*) y almacenamiento (*store*) se debe a que la carga solo se realiza una vez y de ahí es distribuida, a diferencia de los sistemas monolíticos que requieren una lectura y escritura para cada filtro de procesamiento. Las pruebas fueron realizadas dentro de un servidor, lo cual limita este desarrollo, implementación y evaluación a una sola arquitectura hardware de forma aislada.

2.2.2 Tuberías de procesamiento para servicios de seguridad de datos extremo a extremo

El uso de unidades de procesamiento en aplicaciones extremo a extremo para el almacenamiento seguro en la nube ha sido abordado por la literatura, Yanez-Sierra *et al*. [40] proponen una arquitectura para el cifrado, control de acceso, realización de firmas digitales y empaquetamiento de objetos. Los servicios de seguridad son implementados en las unidades de procesamiento

pertenecientes a una tubería de procesamiento, estas se comunican mediante las interfaces I/O como lo son: red, memoria y sistema de archivos como se muestra en la Figura 2.6.

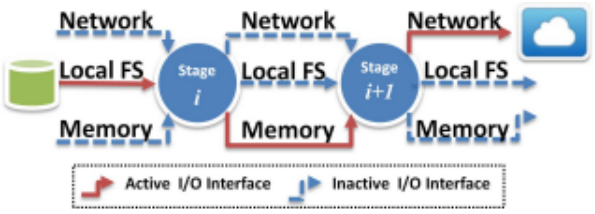


Figura 2.6: Comunicación de unidades de procesamiento propuesta en [40].

La arquitectura propuesta se muestra en la Figura 2.7 en la cual se observa que las tuberías de procesamiento se realizan del lado de los equipos de los usuarios antes de realizar el envío a la nube.

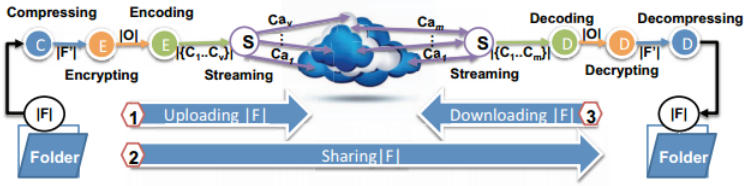


Figura 2.7: Estructura del flujo de trabajo basado en unidades de procesamiento encadenadas mediante interfaces I/O [40].

2.2.3 Tuberías de procesamiento utilizadas para el almacenamiento y compartición de contenido en la nube

Gonzalez *et al.* [34] presentan SkyCDS, que es un servicio de entrega de contenido basado en componentes y subcomponentes para la publicación y suscripción segura de datos de manera diversificada en la nube. Los elementos de SkyCDS garantizan distintos comportamientos y flujo de trabajo a partir de la configuración de sus componentes. Cada componente contiene un comportamiento esencial para el flujo correcto de los datos a través del servicio. En la Figura 2.8 se observan los componentes y subcomponentes de SkyCDS.

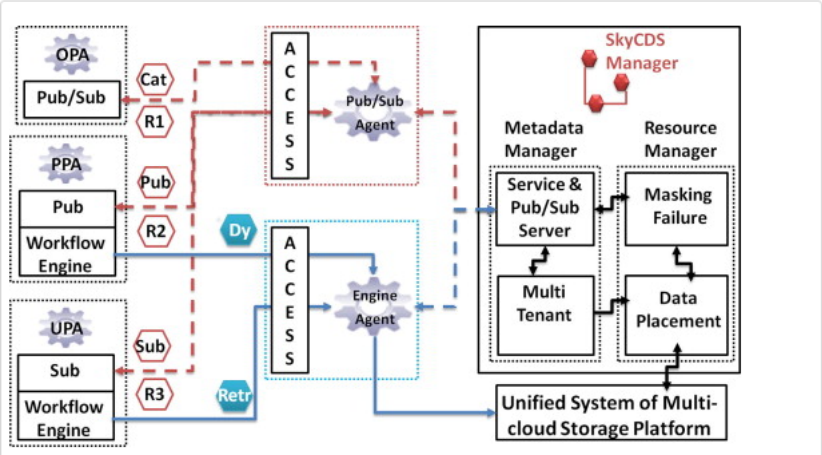


Figura 2.8: Componentes y subcomponentes de SkyCDs [34].

2.2.4 División de un sistema de información utilizando micro-servicios

Por otra parte Abrams *et al.* [2] presentan una revisión al diseño conceptual y la implementación técnica del ambiente de micro-servicios de la infraestructura desarrollada por el centro de curación de la Universidad de California, la cual está basada en la idea de micro-servicios [13, 48], que consiste en la descomposición de una función de depósito en un conjunto altamente granular y ortogonal de componentes independientes pero interoperables que se pueden componer libremente en combinaciones estratégicas hacia fines útiles [23, 39]. La base para el enfoque de micro-servicios son las tuberías de Unix [53] que dieron origen a la arquitectura de *software* de tuberías de procesamiento.

En esta propuesta se basa en un sistema centralizado, donde cada micro-servicio incluye las direcciones web accesibles de los otros servicios que requiere para su funcionamiento, la comunicación entre estos se realiza mediante peticiones RESTful. En la Figura 2.9 se muestra el flujo de trabajo para la ingesta de datos.

Como se ilustra en la Figura 2.10, dentro de cada micro-servicio es posible establecer mecanismos de comunicación entre sus componentes subsidiarios. Por lo tanto, es posible generar mecanismos

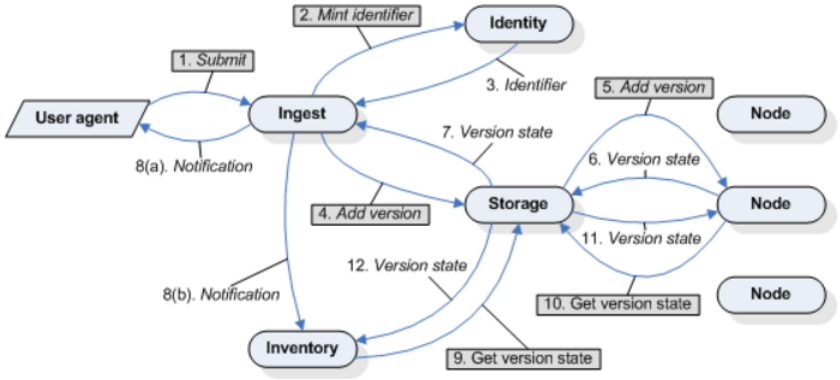


Figura 2.9: Flujo de trabajo para la ingesta de datos [2].

para la gestión de datos que arriban a un micro-servicio mediante el uso de un flujo de datos y un flujo de metadatos.

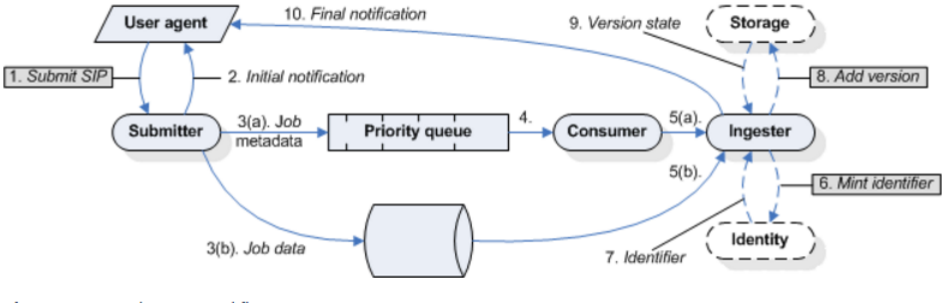


Figura 2.10: Flujo de trabajo interno de la ingesta de datos [2].

Esta arquitectura carece de generalidad, ya que esta propuesta resuelve un solo problema, utilizando patrones de software para su solución, es decir, para un propósito específico.

2.2.5 Micro-aplicaciones para el desarrollo de aplicaciones extremo a extremo

En el proyecto Sacbe [33] se propone el uso de una arquitectura de *software* modular que permite encapsular módulos de *software* en sus contenedores[1] llamados *building blocks* (BBs) que están interconectados por las interfaces de comunicación I/O como la red para el almacenamiento, compartición y seguridad de datos en la nube. A diferencia de las tuberías de procesamiento de Unix, esta propuesta fue realizada para aplicaciones extremo a extremo de almacenamiento en la nube en vez de herramientas del sistema operativo para sistemas de aplicaciones de almacenamiento como [36].

En Sacbe los módulos de cobertura y gestión son encapsulados por micro-aplicaciones independientes en BBs. Estos BBs pueden realizar tareas como adquisición, entrega y procesamiento. En la etapa de procesamiento los datos son procesados por instancias de software llamadas unidades de procesamiento (PUs, por sus siglas en inglés). Los contenedores que encapsulan PUs son llamados *data coverage build blocks* (DC-BB) mientras que los contenedores que encapsulan PUs *deploying management modules* son llamados *management building blocks* (M-BB).

El uso de Sacbe permite tener una (un-PU) o múltiples PUs (N-PU) dentro de un BB. Estas configuraciones se pueden observar en la Figura 2.11.

Al conectar n DC-BBs se puede generar un flujo de trabajo continuo en un extremo del proceso, donde $n \geq 1$. En la Figura 2.12 se muestra un flujo de trabajo donde $n = 4$.

Cada PU es independiente y realiza una operación, sus PUs adyacentes reciben los datos

[1] Se refiere a la abstracción de sus aplicaciones dentro de un elemento que las contiene y no de los VCs mencionados anteriormente.

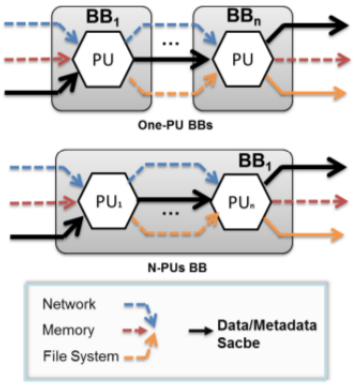


Figura 2.11: Configuraciones de BBs con un-PU y N-PU [33].

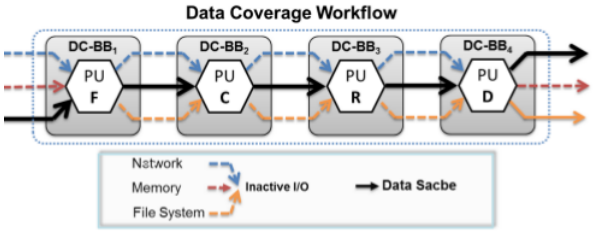


Figura 2.12: Ejemplo de cobertura tubería PUT y data Sacbe construida mediante Pipe-BB chaining policy [33].

procesados sin saber lo que la PU anterior realizó.

Los M-BB son instancias para la gestión de los meta datos generados por los Pipe-BB. Estas M-BB se encuentran distribuidas en la nube y permiten el transporte de datos de un extremo a otro.

2.2.6 Contenedores y micro-servicios para infraestructuras de nube usando DevOps

En [43] se realiza una exploración de las oportunidades y retos al utilizar contenedores al aplicar el diseño basado en micro-servicios para operar y manejar servicios de infraestructuras en la nube usando como caso de estudio el *framework OpenStack*. En este trabajo se presenta el diseño, implementación y despliegue de micro-servicios encapsulados en contenedores en una instancia de

OpenStack. Típicamente, la infraestructura de *OpenStack* está compuesta de tres tipos de nodos: nodos de plano de control (mensajes, base de datos y control), cómputo y red.

La definición de contenedores de *OpenStack* requiere colocar estos tipos de nodos en contenedores. Por ejemplo, el contenedor del plano de control debería estar informado de dónde se encuentran los servicios de base de datos y mensajes, cuando estos servicios son creados recientemente, recuperados de una falla o apagados. Para solventar esta necesidad, en [43] se presenta una arquitectura que facilita la operación de escalamiento, alta disponibilidad y balanceo de carga. En base a estos requerimientos se propone una arquitectura que incluye: Un servidor proxy, un administrador de estado de configuración y un orquestador de servicios. En la Figura 2.13 se muestra la arquitectura así como sus componentes. El despliegue de la arquitectura está compuesta por tres etapas (1) lanzamiento de un contenedor controlador (2) registrar el servicio y (3) descubriendo el servicio.

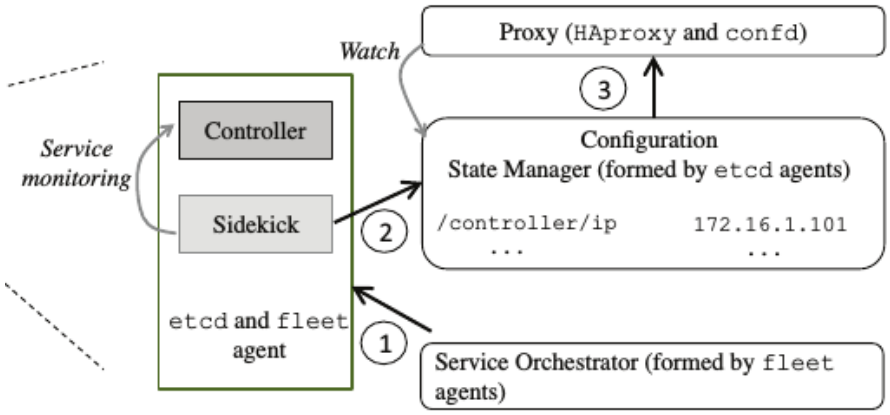


Figura 2.13: Componentes y etapas de la arquitectura propuesta por [43].

El despliegue de la arquitectura propuesta fue realizado en dos fases: inicialización y escalamiento. En la fase de inicialización, crearon una instancia de cada tipo de nodo en cada host (creando una

VM o VC) para levantar una instancia de plano de control de OpenStack. En la fase de escalamiento se añadieron dos instancias más de cada nodo para formar un cluster de alta disponibilidad. En comparación el despliegue de las dos fases de OpenStack usando VCs es 1.5X más rápido que al usar VMs.

2.2.7 Entornos basados en contenedores para la creación de flujos de trabajo científicos

En Skyport [32] se presenta la encapsulación de aplicaciones del flujo de trabajo dentro de VCs. Esta encapsulación permite brindarles el *runtime* requerido para la ejecución de los pasos del flujo de trabajo. Al obtener el *runtime* se obtiene la capacidad de generar un contenedor igual a uno ya en ejecución disponible para el mismo flujo de trabajo u otro distinto.

Para lograr la encapsulación de los contenedores Skyport presenta su propia arquitectura diseñada para manejar sus flujos de trabajo. Además, cuenta con su repositorio de imágenes de contenedor Docker para la ejecución de nuevos trabajadores.

En la Figura 2.14 se presenta la arquitectura desarrollada por Skyport y en la Figura 2.15 se presentan las imágenes de contenedor generadas para su funcionamiento.

Como resultado de la utilización de contenedores en [32] mencionan que el uso de los flujos de trabajo sin contenedores y con Skyport tienen rendimientos similares. Skyport es un poco más costoso debido al *overhead* generado por la administración de los contenedores.

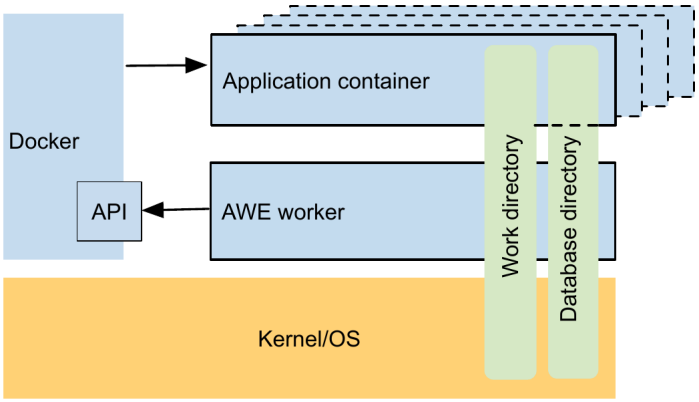


Figura 2.14: Arquitectura de Skyport [32].

repository:tag	base image	total	diff
ubuntu:14.04	–	213.0	213.0
mgrast/base:latest	ubuntu:14.04	586.9	373.9
mgrast/superblat:latest	mgrast/base:latest	591.2	4.3
mgrast/rna_search:latest	mgrast/base:latest	589.3	2.4
mgrast/qc:latest	mgrast/base:latest	739.6	152.7
mgrast/genecall:latest	mgrast/base:latest	643.6	56.7
mgrast/cluster:latest	mgrast/base:latest	587.8	0.9
mgrast/bowtie:latest	mgrast/base:latest	625.8	38.9
mgrast/blat:latest	mgrast/base:latest	596.7	9.8

Figura 2.15: Imágenes de contenedor usadas por Skyport [32].

2.2.8 Resumen del estado del arte

Como se muestra en la Tabla 2.1, existen varias propuestas para la segmentación del proceso principal de un sistema de información en múltiples pasos de programación secuenciales. En [36] presentan los resultados obtenidos al utilizar múltiples *pipelines* en paralelo usadas para dividir la tarea principal de su sistema. Los resultados que obtuvieron muestran que el uso de *pipelines* permite un incremento lineal en el rendimiento en MB/s obtenidos en la agilización de la carga y almacenamiento de archivos por número de *pipelines* utilizados. Sin embargo, su solución es implementada dentro de un solo servidor haciéndola una aplicación centralizada limitada a los recursos que este posea.

Yanez-Sierra *et al.* [40] proponen el uso de unidades de procesamiento en aplicaciones extremo a extremo para el almacenamiento seguro de datos en la nube. Esta propuesta puede ser implementada del lado del cliente o servidor limitándose a los recursos con los que estos cuenten.

Gonzalez *et al.* [34] presentan un servicio de entrega de contenido en una o varias infraestructuras de nube. Este servicio utiliza varias capas de procesamiento para la realización de su tarea principal. Estas capas están implementadas en arquitecturas cliente o servidor limitándose también a los recursos disponibles en estos.

Abrams *et al.* [2] presentan un entorno basado en micro-servicios orientado a la curación de documentos digitales, utilizando la metáfora de *pipelines* de Unix para la segmentación de la tarea principal de su sistema en micro-servicios. La segmentación fue desarrollada en un ambiente distribuido y utiliza peticiones *restful* para el seguimiento del flujo de datos entre los micro-servicios a través de servidores *web*. El inconveniente con esta propuesta es el uso de archivos de configuración en los servidores que permiten la la transferencia de mensajes y datos entre los micro-servicios, dichos archivos de configuración deben de ser actualizados y mantenidos de forma manual.

En el trabajo presentado en [33] los autores presentan Sacbe una arquitectura de software modular para la construcción segura, fiable, y flexible en un entorno extremo a extremo para el almacenamiento de datos en la nube, Sacbe propone crear caminos virtuales seguros para transportar datos y metadatos desde sus dispositivos a la nube a través de flujos de datos continuos. Sacbe contiene una arquitectura modular que permite a los diseñadores desplegar módulos de cobertura y gestión como instancias de software en contenedores interconectados mediante las interfaces de comunicación de entrada y salida tales como red, memoria y sistema de archivos.

En los trabajos presentados en [32] y [43] los autores hablan de la contenerización de aplicaciones

utilizando patrones de diseño para facilitar el desarrollo de flujos de trabajo y clusters de alta disponibilidad respectivamente.

Tabla 2.1: Comparativa de artículos del estado del arte y método propuesto.

Trabajo	**Observaciones**		
	Tipo de aplicación	Uso de la nube	Entidad de procesamiento
2015 [36]	Centralizado	Ninguno	Servidor
2015 [40]	Distribuido	Almacenamiento	Cliente y servidor
2015 [34]	Distribuido	Almacenamiento	Cliente y servidor
2011 [2]	Distribuido	Almacenamiento	Cliente y servidor
2016 [33]	Distribuido	Almacenamiento	Cliente y servidor
2014 [32]	Distribuido	Procesamiento	Cliente, servidor y nube
2016 [43]	Distribuido	Análisis	Cliente, servidor y nube
Método propuesto	Distribuido	Almacenamiento y procesamiento	Cliente, servidor, nube y cluster

Algunas de las propuestas mencionadas realizan el procesamiento de datos del lado del cliente y/o servidor en sistemas centralizados o distribuidos limitándose al uso del almacenamiento de datos en la nube, mientras que otras utilizan la nube para el despliegue de componentes para el análisis de datos. A diferencia estas propuestas se espera que el método presentado en este trabajo de tesis permita el despliegue, gestión y comunicación de tuberías de procesamiento de datos eficientes (en tiempos de respuesta) de forma distribuida en diferentes infraestructuras como dispositivos finales (cliente, servidor), clusters de computadoras y la nube, mediante la encapsulación de aplicaciones en VCs cumpliendo con los objetivos planteados en la sección 1.4.

3

Método para la construcción de tuberías de procesamiento de datos para la nube

En este capítulo se detallan los componentes diseñados para el desarrollo del método propuesto que permite el despliegue de tuberías de procesamiento de datos basadas en contenedores utilizando distintas infraestructuras de *hardware* tales como servidores y *clusters* de computadoras en la nube.

3.1 Definiciones de componentes del método propuesto

Para entender el funcionamiento y componentes del método propuesto es necesario conocer las siguientes definiciones:

- *Meta-Filtro*: Unidad de construcción que abstrae una tubería de procesamiento en su interior que permite la E/S de datos a través de sus interfaces de comunicación de E/S.

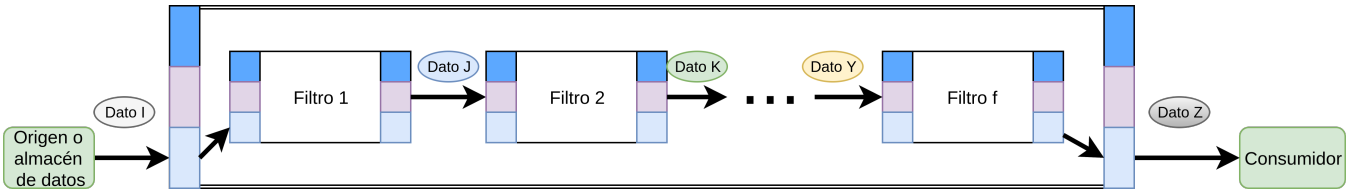


Figura 3.1: Ejemplo de Meta-Filtro.

- *Meta-Tubería* : Sucesión encadenada de *n* Meta-Filtros de procesamiento conectados de forma adyacente a través de sus interfaces de comunicación desde un origen o almacén de datos hasta un destino o consumidor donde $n \geq 1$. Esta Meta-Tubería forma un flujo de datos continuo.

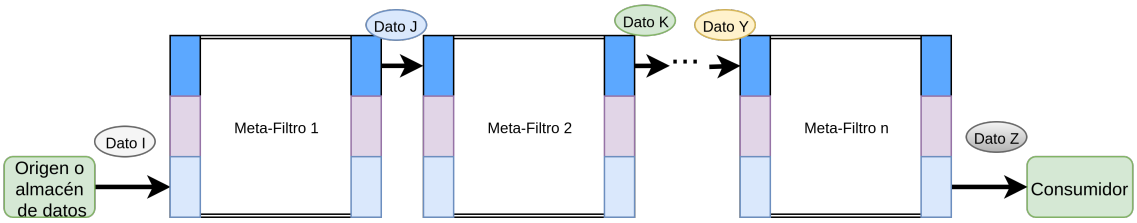


Figura 3.2: Ejemplo de Meta-Tubería.

- *Bloque de construcción*: Un bloque de construcción (BB, por sus siglas en inglés) es una representación abstracta de la estructura básica de la que derivan los componentes de la arquitectura de Meta-Tuberías propuesta. Consta de una capa de acceso y una de procesamiento. Para nuestro caso de estudio, el BB básico constará de una capa de acceso con interfaces de E/S y con una capa de procesamiento en la que se encapsula un Filtro o un Meta-Filtro de procesamiento.

- *Bloque de construcción paralelo*: Un bloque de construcción paralelo (PBB, por sus siglas en inglés) es un filtro de procesamiento que implementa el esquema *divide y encapsula* propuesto por este trabajo de tesis. Tal como se muestra en la sección 3.4, el diseño de este filtro permite la utilización de múltiples Meta-Tuberías de procesamiento en forma paralela para el procesamiento de los datos para reducir el tiempo de respuesta observado por el usuario final.

- *Imagen de contenedor base*: Es la imagen de contenedor (CI, por sus siglas en inglés) que contiene las bibliotecas y dependencias de *software* requeridas por el BB diseñada.
- *CI de Meta-Filtro*: Es la CI creada a partir de una CI base que añade las dependencias de software del Meta-Filtro contenido en su interior.
- *Contenedor virtual de Meta-Filtro*: Es el contenedor virtual (VC, por sus siglas en inglés) obtenido al ejecutar una CI de Meta-Filtro.
- *Silo de CIs y CIs de Meta-Filtros* : La definición de *silo* dada por la Real Academia Española es: "Lugar seco en donde se guarda el trigo u otros granos, semillas o forraje" [25]. Las características de un silo son: tener una puerta entrada o acceso que permita ingresar algún contenido para realizar su almacenamiento, un espacio reservado para almacenar el contenido que entra por la puerta de acceso y una puerta de salida que permita realizar la extracción del contenido. Por estas características se escogió el nombre *silo* para el almacén de las CIs base y CIs de Meta-Filtro.

Una vez definidos estos componentes se puede realizar la descripción del método el cual es presentado en la siguiente sección.

3.2 Método propuesto

El método propuesto se compone de las siguientes etapas:

- Generar el BB y PBB,
- encapsular filtros y tuberías de procesamiento dentro de BBs y PBBs dando como resultado filtros y Meta-Filtros,
- realizar la encapsulación de los Meta-Filtros en VCs lanzados a partir de CIs con sus dependencias instaladas,

- exportar el VC generado en el paso anterior generando una nueva CI llamada CI de Meta-Filtro,
- recopilar las CIs de Meta-Filtros en un almacén,
- crear archivos de configuración para cada CI de Meta-Filtro por utilizar,
- realizar el despliegue de las CIs de Meta-Filtros con sus respectivos archivos de configuración generando VCs de Meta-Filtros que forman parte de una Meta-Tubería de procesamiento.
- Un BB basado en una arquitectura de software modular compatible con contenedores que permite la encapsulación de filtros y tuberías de procesamiento para la generación de filtros y Meta-Filtros de procesamiento con la capacidad de comunicarse a través de diferentes interfaces de comunicación de entrada y salida (E/S) del sistema operativo anfitrión[1].
- Un constructor de CIs de Meta-Filtros encargado de generar nuevas CIs de Meta-Filtros a partir de la filtros y Meta-Filtros en CIs que contienen sus dependencias instaladas o los comandos para su instalación. Este proceso genera nuevas CIs de Meta-Filtros que son portables y se encuentran listas para su despliegue en diferentes infraestructuras que contengan la misma plataforma contenedores utilizada para su creación.
- Un silo de CIs y CIs de Meta-Filtros disponible para la carga y descarga de CIs y CIs de Meta-Filtros para la compartición de Meta-Filtros funcionales para la generación de Meta-Tuberías.
- Un planificador de conexiones que permita generar los archivos de configuración para cada CI de Meta-Filtro por utilizar. Dicho archivo será usado para habilitar las E/S de los VCs de Meta-Filtros lanzados, permitiendo la interconexión y transferencia de datos entre múltiples VCs de Meta-Filtros para formar Meta-Tuberías.
- Un constructor de Meta-Tuberías encargado de solicitar CIs de Meta-Filtros desde el silo, usar los archivos de configuración generados por el planificador de conexiones para cada CI de

[1]El sistema operativo que contiene la plataforma de contenerización y lanza los VCs y VCs de Meta-Filtros

Meta-Filtro, lanzar los VCs de Meta-Filtro usando las CIs de Meta-Filtro descargadas desde el silo usando las configuraciones establecidas en su archivo de configuración para habilitar sus interfaces de comunicación generando Meta-Filtros de procesamiento que forman parte de la Meta-Tubería.

- Un esquema encargado de realizar la división de carga de trabajo en múltiples Meta-tuberías incrementando el rendimiento del procesamiento de los datos encapsulado dentro de un BB generando un PBB.

Los diseños de estos componentes son mostrados en la siguiente sección.

3.3 Diseño del método propuesto

En esta sección se describen los elementos involucrados en el diseño de los componentes desarrollados para la creación del método para la construcción de tuberías de procesamiento de datos para la nube basado en contenedores.

Los diseños mostrados en esta sección permiten la creación, configuración de filtros y Meta-Filtros basados en BBs y PBBs dentro de VCs de Meta-Filtros que forman parte de Meta-Tuberías de procesamiento de datos que ofrecen las características de *flexibilidad* en términos de *variedad* de procesos, *elasticidad* en términos de manejo de grandes *volúmenes* de datos y *eficiencia* en términos de *velocidad* de procesamiento.

El diseño de la arquitectura se encuentra dividido en cinco entidades: el BB, el constructor de CIs de filtros y Meta-Filtros, el planificador de conexiones, el constructor de Meta-Tuberías y el esquema de división de carga de trabajo basado en BB llamado PPB. Los diseños de estas entidades son descritos a continuación.

3.3.1 Bloque de construcción

El bloque de construcción o BB (por sus siglas en inglés) propuesto se encuentra dividido en dos capas: la capa de acceso y la capa de procesamiento. Las funciones realizadas por la capa de acceso son la recepción y transmisión de datos, mientras que la capa de procesamiento encapsula los filtros de procesamiento que debe aplicar el BB. En la Figura 3.3 se muestra el diseño en capas del BB así como los elementos que contienen en su interior.

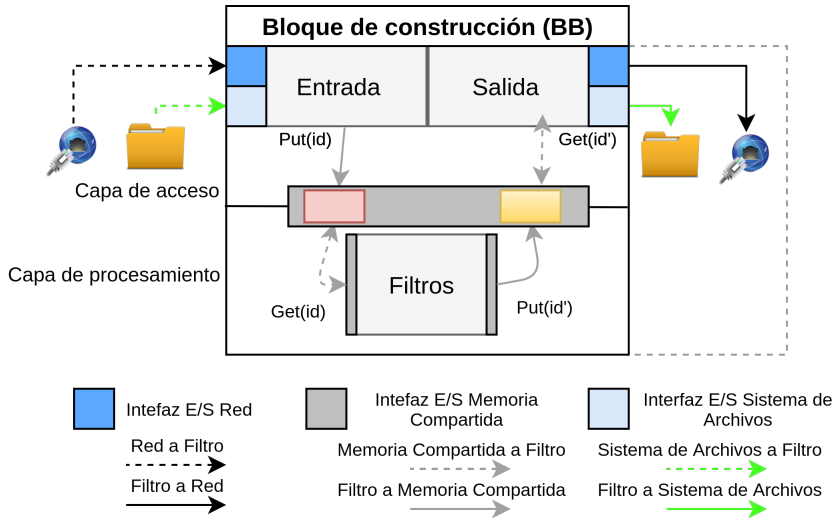


Figura 3.3: Diseño en capas del BB.

Las capas presentadas en el diseño del BB son descritas a continuación.

Capa de acceso: Es la capa encargada de la recepción y transmisión de datos a través de las interfaces de E/S habilitadas del BB. Para realizar las tareas de recepción y transmisión de datos de esta capa se establecen dos módulos en su interior: el módulo *Entrada* de datos encargado de habilitar las interfaces de comunicación del BB que permiten que otros BBs puedan comunicarse y realizar el envío de datos a este. El módulo *Salida* de datos permite realizar la comunicación y envío de datos a través de alguna de las interfaces de salida de este BB al módulo de recepción de datos de otro BB. Estos módulos habilitan las interfaces de E/S respectivamente siguiendo los parámetros establecidos dentro del archivo de configuración de este BB. Las interfaces E/S consideradas para el

BB son:

- *Red*: Interfaz de comunicación usada para la transmisión de mensajes y datos a través de una red de computadoras.

- *Memoria compartida*: Interfaz de comunicación usada para compartir contenido entre los filtros de procesamiento colocados en el interior del BB.

- *Sistema de archivos*: Interfaz de comunicación usada para leer y escribir contenido en el sistema de archivos del sistema operativo anfitrión.

Capa de procesamiento: En esta capa se realiza el trabajo de procesamiento del BB. Es la encargada de encapsular y ejecutar los filtros o tuberías de procesamiento del BB como se muestra en la Figura 3.3. La capa de procesamiento permite la encapsulación de *variedad* de filtros de procesamiento, por ejemplo en la Figura 3.4 se presenta una tubería de procesamiento con f filtros de procesamiento encapsulados en la capa de procesamiento del BB. El proceso del BB mostrado en la Figura 3.4 realiza la lectura de un contenido usando la interfaz de entrada del sistema de archivos, este contenido es procesado por cada uno de los f filtros de procesamiento y es transferido a través de la interfaz de salida al sistema de archivos para su almacenamiento, al contener una tubería de procesamiento esta BB se le considera un Meta-Filtro.

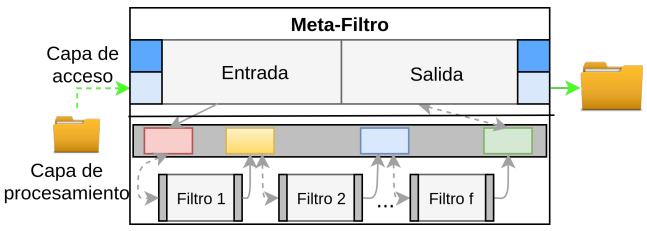


Figura 3.4: Ejemplo de Meta-Filtro.

3.3.2 Constructor de Imágenes de contenedor de Meta-Filtros

Una vez definido el BB es necesario realizar el proceso encargado de la construcción de las CIs con Meta-Filtros en su interior. Este proceso será realizado de forma asíncrona de los demás componentes. Por lo tanto el tiempo de ejecución utilizado por este componente se le considera *offline*.

El proceso de construcción de CIs de Meta-Filtros está conformado de dos fases: La fase uno corresponde a la selección de una CI base [2] y el Meta-Filtro de procesamiento que se desea encapsular y la fase dos consiste en generar una CI de Meta-Filtro insertando el Meta-Filtro requerido dentro de la CI seleccionada.

En la fase uno se debe de obtener a la CI base y se debe seleccionar el Meta-filtro que se desea integrar a ésta. La CI base se obtiene desde el Silo de CIs y el Meta-Filtro por utilizar debe de ser proporcionado por su desarrollador, ambos elementos deben encontrarse en el sistema de archivos del sistema operativo que ejecutará el constructor. La fase dos inicia después de que ambos elementos son obtenidos incrustando el Meta-Filtro a la CI base generando una CI de Meta-Filtro, para finalizar esta CI de Meta-Filtro es transportada al sistema de archivos del sistema operativo anfitrión o transferido al Silo. El proceso para la creación de CIs de Meta-Filtros es mostrado en la Figura 3.5.

[2] Imagen de contenedor con las dependencias de software del BB pre-instaladas

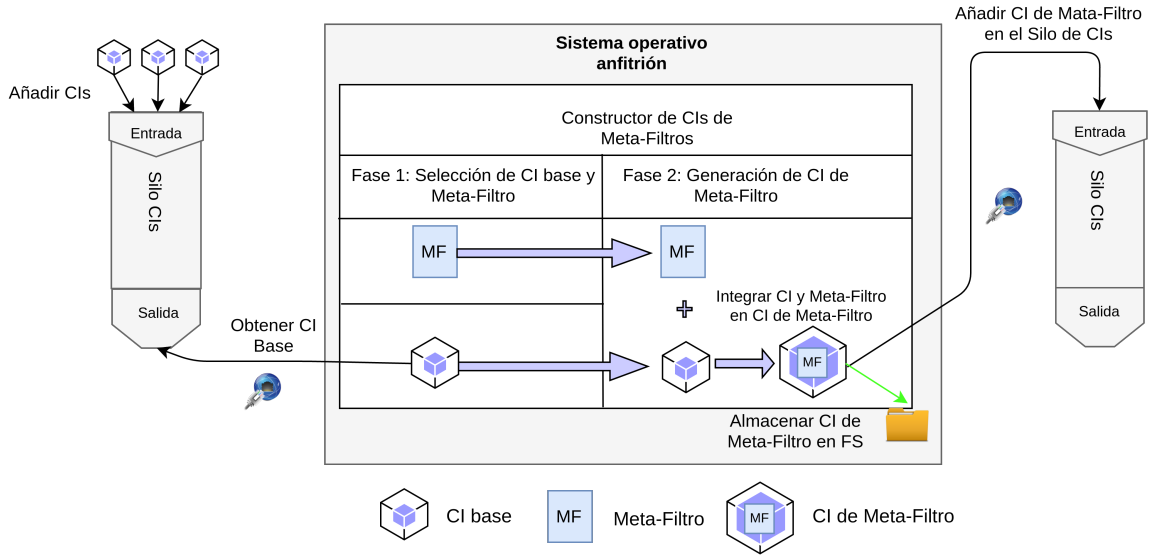


Figura 3.5: Proceso de construcción de CI de Meta-Filtro.

Los componentes requeridos para el desarrollo de esta aplicación son: el silo de CIs y CIs de Meta-Filtros, un recolector de CIs y CIs de Meta-Filtros, la función de integración de la CI obtenida por el recolector y el Meta-Filtro desarrollado, un repartidor encargado de exportar la CI de Meta-Filtro generada de forma local o remotamente al silo.

En caso de que no se establezcan los datos de recursos compartidos al lanzar las CI de Meta-Filtro y convertirse en VC de Meta-Filtros estos se quedarán aislados y no podrán enviar o recibir datos por lo que no pueden procesar datos y formar parte de una Meta-Tubería.

3.3.3 Planificador de conexiones

En la sección anterior se definió el proceso para la creación de CIs de Meta-Filtros. Esta sección corresponde a la generación de una herramienta que permita crear archivos de configuración que habiliten las interfaces de E/S de los VCs de Meta-Filtros desplegados generando la "trayectoria" de la Meta-Tubería a partir de las CIs de Meta-Filtros generadas por el constructor.

Esta herramienta es necesaria debido a que al lanzar una CI de Meta-Filtro y convertirse en VC de Meta-Filtro se requiere previo conocimiento de los recursos de *hardware* compartidos con el sistema operativo anfitrión para que este les conceda el acceso a sus recursos y permita el flujo de datos a la capa de acceso del VC de Meta-Filtro. En la Figura 3.6 se puede observar que hay m VCs de Meta-Filtros lanzados sin utilizar los archivos de configuración, estos VCs de Meta-Filtros están aislados por lo que no forman parte de una Meta-Tubería.

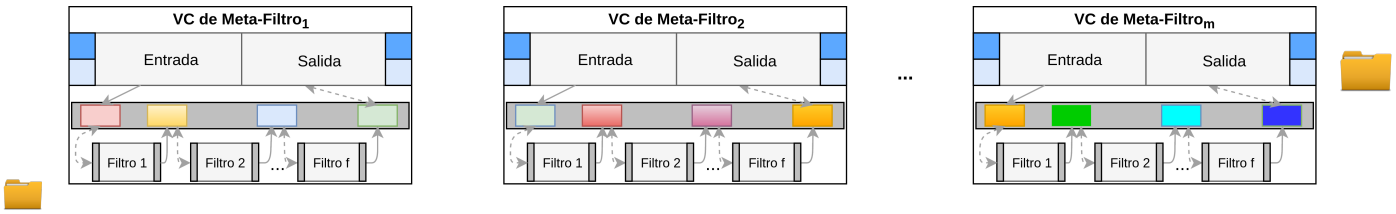


Figura 3.6: Ejemplo de VC de Meta-Filtros aislados.

Para establecer las *trayectorias* entre los VCs de Meta-Filtros desplegados se presenta el diseño de un planificador de conexiones. Este planificador permite generar un archivo de configuración para cada una de las CIs de Meta-Filtros por utilizar para las Meta-Tuberías que se desean desplegar. Cada archivo de configuración generado contiene la información necesaria para habilitar las interfaces de comunicación E/S entre Meta-Filtros para enviar y/o recibir datos.

La información contenida en el archivo de configuración contiene los datos del VC de Meta-Filtro actual y los VC de Meta-Filtros adyacentes a éste. Los datos de configuración incluidos en el VC de Meta-Filtro local son: i) El nombre de la CI base que será usada para su creación. ii) El identificador del Meta-Filtro. iii) El nombre que se le asignará a la CI de Meta-Filtro. iv) La etiqueta del VC de Meta-Filtro que será usada para el control de versiones, v) Las *rutas* del sistema de archivos utilizadas como origen o destino si se utilizará el sistema de archivos como interfaz de E/S. En el caso de tener VCs de Meta-Filtros adyacentes se requieren los siguientes datos: i) La dirección IP del VC de Meta-Filtro. ii) El número de puerto de escucha menor y el puerto de escucha mayor que

pueden usarse para la comunicación entre VCs de Meta-Filtros.

Al integrar el archivo de configuración a los VC de Meta-Filtros se pueden establecer múltiples configuraciones que permitan su interconexión y así generar diferentes Meta-Tuberías. En la Figura 3.7 se muestra un ejemplo de una Meta-Tubería compuesta por *m* VC de Meta-Filtros que contienen *f* filtros de procesamiento en su interior lanzados utilizando archivos de configuración que establecen el flujo de datos desde un origen a un destino.

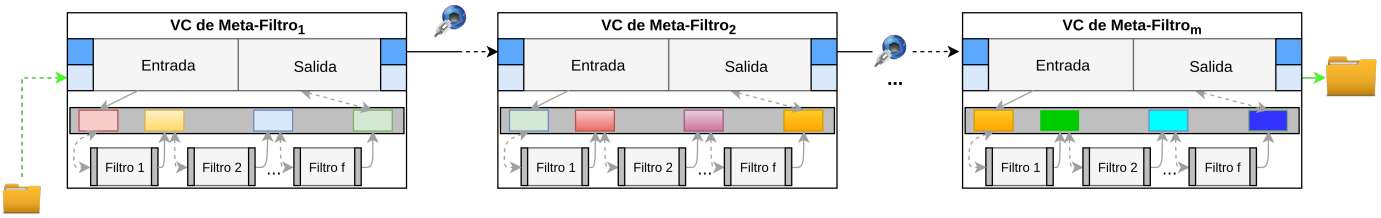


Figura 3.7: Tubería de procesamiento configurada.

3.3.4 Constructor de Meta-Tuberías

Este componente requiere que se encuentre habilitado el *silo* de CIs de Meta-Filtros y el planificador de conexiones. Esto se debe a que se realizará una serie de peticiones al *silo* para obtener las CIs de Meta-Filtros necesarias para la construcción de las Meta-Tuberías de procesamiento. Al ejecutar las CIs de Meta-Filtros descargadas se generan VC de Meta-Filtros, a estos se les deben insertar los archivos de configuración generados por el planificador de conexiones para que se les habiliten las interfaces de E/S permitiendo la interconexión de dichos VC de Meta-Filtros generando una Meta-Tubería. En la Figura 3.8 se muestran los pasos a seguir para la creación de una Meta-Tubería, los cuales se describen a continuación:

1. Recuperar CIs de Meta-Filtros: Realiza la petición de descarga de las CIs de Meta-Filtros requeridas por el desarrollador para la creación de la Meta-Tubería al *silo* de CIs de Meta-Filtros.

2. Generar los archivos de configuración: En esta etapa se deben de generar los archivos de configuración utilizando el planificador de conexiones para cada CI de Meta-Filtro recuperada con los datos requeridos para esta.

3. Despliegue de VC de Meta-Filtros: Partiendo de los archivos de configuración creados, cada CI de Meta-Filtro es ejecutada habilitando las interfaces de E/S del VC de Meta-Filtro correspondientes a los datos de su archivo de configuración, después de crear el VC de Meta-Filtros el archivo de configuración usado es copiado en su interior.

4. Conexión: Tiene como objetivo ejecutar la aplicación de configuración dentro de cada VC de Meta-Filtro que habilita las interfaces de E/S utilizando como parámetros de entrada los datos del archivo de configuración. Este proceso permite establecer la "trayectoria" que deben de seguir los datos ingresados al VC de Meta-Filtro obteniendo un flujo de datos continuo generando una Meta-Tubería.

3.4 Esquema de procesamiento *divide y encapsula*

El método propuesto para la generación de Meta-Tuberías posibilita el manejo de *variedad* de filtros de procesamiento y el manejo de grandes *volúmenes* de datos usando los Meta-Filtros que permiten construir Meta-tuberías a través de la interconexión de VC de Meta-Filtros. Sin embargo, en la presente tesis también se estableció la característica de *velocidad* de procesamiento como un requerimiento de la problemática identificada. Para cumplir con este requerimiento se diseñó el bloque de construcción paralelo(PBB, por sus siglas en inglés).

PBB es un Meta-Filtro que integra al diseño del BB propuesto con filtros adicionales que implementan las técnicas de segmentación e integración de datos. Estas técnicas permiten la división de carga de trabajo de un BB en múltiples Meta-Tuberías de forma local o distribuida dependiendo

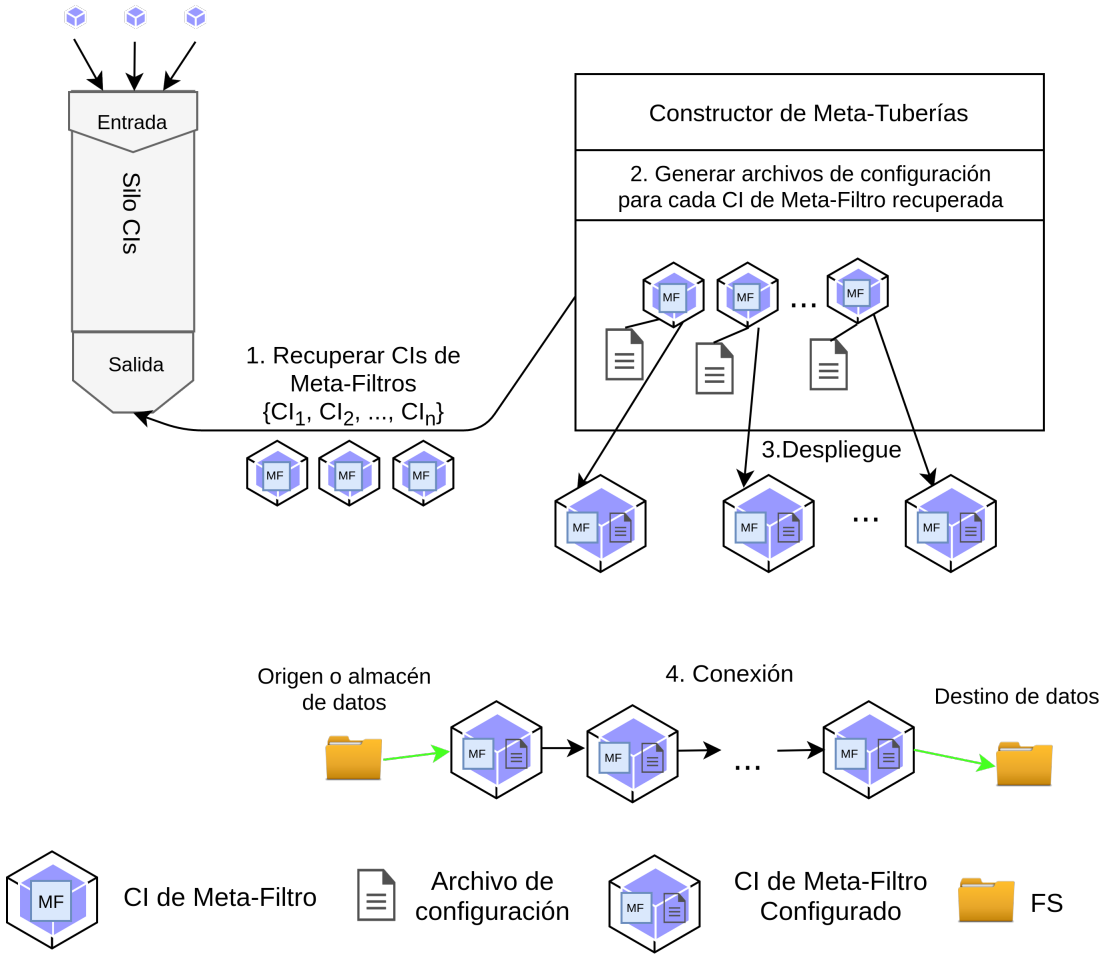
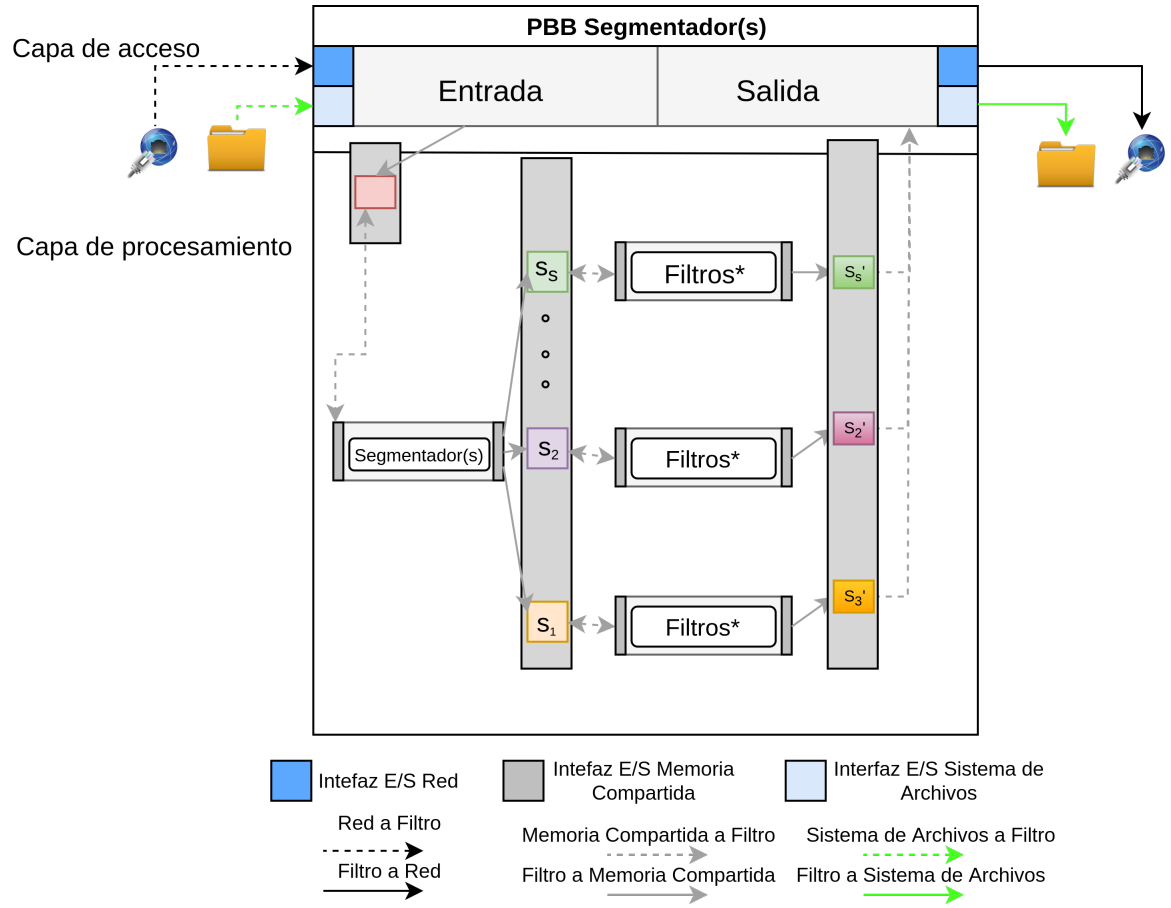


Figura 3.8: Generación de una Meta-Tubería.

de la configuración de sus interfaces de entrada.

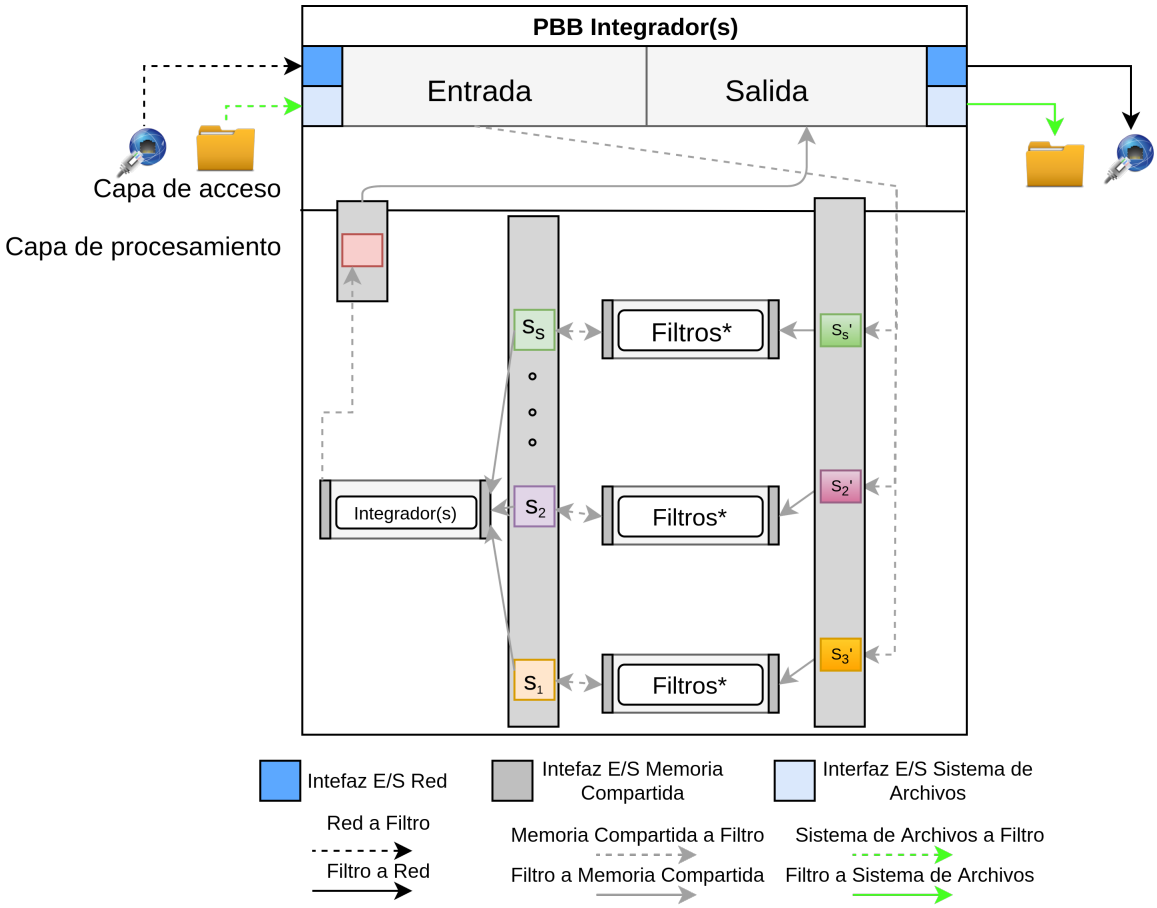
PBB Segmentador: En la Figura 3.9 se puede observar el diseño de una PBB de segmentación que recibe los datos por la capa de entrada por la interfaz de red o sistema de archivos y lo pasa a un filtro de segmentación a través de un segmento de memoria compartido (SMS, por sus siglas en inglés), este filtro genera s número de segmentos que son transferidos de forma paralela a otros filtros de procesamiento o enviados a otros VC de Meta-Filtros usando las interfaces de salida al sistema de archivos o la red.



* No es indispensable, los segmentos pueden ser enviados sin procesar a otro VC de Meta-Filtro a través de las interfaz de salida de red o almacenado usando el la interfaz del Sistema de Archivos.

Figura 3.9: Diseño del PBB de segmentación.

PBB Integrador: En la Figura 3.10 se puede observar el diseño de un PBB de integración que realiza la petición de los segmentos a los VCs de Meta-Filtros con los segmentos correspondientes en forma paralela y los coloca en SMS. Al terminar de descargar los segmentos, estos son integrados y almacenados usando la interfaz de salida del sistema de archivos o enviados a otro VC de Meta-Filtro de almacenamiento usando la interfaz de salida de red.



* No es indispensable, los segmentos pueden ser recuperados sin procesar desde VC de Meta-Filtros fuera de la infraestructura utilizada a través de las interfaz de entrada de red o recuperado desde la interfaz entrada del Sistema de Archivos.

Figura 3.10: Diseño del PBB de integración.

4

Implementación de prototipo funcional basado en la arquitectura propuesta

En este capítulo se describirán las técnicas utilizadas para implementar un prototipo de la propuesta arquitectónica para construir tuberías de procesamiento de datos en la nube, descrita en el capítulo anterior.

4.1 Componentes del prototipo funcional

La forma en la que será evaluado el método propuesto es mediante una evaluación basada en prototipos. Por esta razón se requiere realizar la implementación de los diseños descritos en el capítulo anterior con el fin de generar un prototipo funcional. A continuación se describen los elementos utilizados para la implementación de los cuatro componentes diseñados.

4.1.1 Bloque de construcción

Para la implementación del Bloque de Construcción básica o BB, se tomaron en cuenta los siguientes requerimientos para la selección de herramientas y el desarrollo de bibliotecas que permiten utilizar las interfaces de comunicación de E/S.

Requerimientos

Para realizar la implementación del BB fue necesario contemplar sus componentes y el comportamiento esperado para cada uno. Para realizar la correcta implementación de todos los componentes se establecieron los siguientes requerimientos: Contar con una Interfaz de Programación de Aplicaciones (API, por sus siglas en inglés) para el acceso y transferencia de datos en red, una API para el uso de memoria compartida y una API que permita la lectura/escritura de datos a través del sistema operativo anfitrión.

Lenguaje de programación y sistema operativo utilizados

Actualmente se cuenta con diferentes lenguajes de programación que nos ofrecen APIs para transferencia de datos en la red, comunicación entre procesos (IPC, por sus siglas en inglés) a través de memoria compartida y la lectura/escritura de datos a través del sistema de archivos del sistema operativo anfitrión. Sin embargo, para este trabajo se decidió utilizar el lenguaje C, dada la madurez del lenguaje, la existencia de librerías a bajo nivel que realizan de manera eficiente la ejecución de procesos, así como su acoplamiento estándar (POSIX) con diversos componentes que conforman a los sistemas operativos.

La conexión a través de la red se realiza utilizando *Sockets* los cuales son una herramientas que permite la comunicación entre dos entidades a través de la red [31]. El protocolo de comunicación utilizado para los sockets de la capa de entrada fue el protocolo de control de transmisión (TCP, por

sus siglas en inglés) que garantiza que los datos son recibidos en el mismo orden en el que fueron enviados.

Para el uso de memoria compartida se utilizó la IPC que consiste en mecanismos que permiten a los procesos intercambiar datos a través de segmentos de memoria compartida (SMS). Normalmente, estos datos se formatean en mensajes de acuerdo con un protocolo predefinido[6].

Por último, la lectura/escritura de datos se realiza a través del sistema de ficheros se consiguieron al utilizar la clase *File I/O* del lenguaje C que permite realizar la lectura/escritura de datos (texto, bytes) usando el sistema de archivos del sistema operativo huésped.

En este proyecto se prefirió trabajar con una plataforma de sistema operativo abierto y libre, por lo que se decidió utilizar alguna distribución de Linux, en particular la distribución Linux Ubuntu versión 16.04 como imagen base para los contenedores utilizados.

Retomando las capas que conforman nuestra propuesta de Bloque de Construcción básica (o BB), mostrada en la Figura 3.3 en el capítulo anterior, a continuación se describen las implementaciones que se llevaron a cabo en esta tesis.

Elementos de la capa de acceso

Fueron desarrolladas las bibliotecas de funciones que permitan establecer el uso de *Sockets, IPC* y *File*. En total se desarrollaron tres bibliotecas:

- *Sistema de Archivos*: Biblioteca desarrollada que contiene las funciones requeridas para realizar operaciones relacionadas con la Lectura/Escritura de datos al sistema de archivos del sistema operativo huésped. Las funciones contenidas dentro del archivo de esta biblioteca permiten: verificar si un archivo existe, obtener el tamaño de un archivo, realizar la lectura de archivos

almacenados en el sistema de archivos del sistema operativo huésped y colocarlos en una variable del tipo *File* en la M del sistema operativo, realizar la lectura de un segmento de un archivo y colocarlo en una variable del tipo *File* en la M del sistema operativo y realizar la escritura de datos colocados en una variable del tipo *File* en sistema de archivos del sistema operativos.

- *Memoria Compartida*: Biblioteca que contiene las funciones requeridas para realizar operaciones utilizadas para Recuperar/Colocar datos a la memoria compartida del sistema operativo. Las funciones contenidas dentro del archivo de esta biblioteca permiten: generar un Id para el SMS, reservar un SMS acorde al tamaño del contenido que se requiere compartir, colocar el contenido guardado en la memoria del sistema operativo a SMS, recuperar el contenido guardado en SMS y colocarlo en la memoria del sistema operativo.

- *Red*: Biblioteca con las funciones encargadas de establecer la comunicación, envió, recepción de mensajes y contenidos a través de la red usando *Sockets* del tipo cliente y servidor. Las funciones contenidas dentro del archivo de esta biblioteca permiten: Iniciar Socket del servidor, iniciar escucha de peticiones, establecer conexión desde el socket del cliente, enviar/recibir mensajes entre cliente y servidor, enviar/recibir contenido que son funciones basadas en el código presentado en [58] utilizado cumpliendo los términos establecidos por la licencia de software libre GNU [26] y por último cerrar conexión del cliente con el servidor.

Los archivos desarrollados con las bibliotecas descritas son mostrados en la Figura 4.1 donde se puede observar que se incluyó la biblioteca externa UNVP[58] usada en *Red.h*.

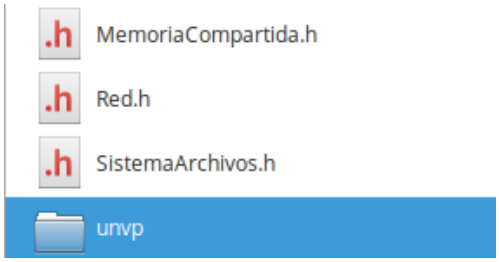


Figura 4.1: Bibliotecas desarrolladas y carpeta con la biblioteca descrita en [58].

Una vez terminadas las bibliotecas se procedió al desarrollo de las aplicaciones que las utilizan para establecer el acceso y la salida de una BB. El total de aplicaciones desarrolladas para la capa de acceso fueron cuatro. Las aplicaciones desarrolladas son descritas a continuación.

1. *EntradaR2MC*: Esta aplicación contiene la implementación requerida para establecer un *Socket* del tipo servidor que al terminar la comunicación permite la ejecución del primer filtro de la capa de procesamiento.

2. *EntradaSA2MC*: Esta aplicación contiene la implementación que permite realizar la lectura de un archivo almacenado en el sistema de archivos del sistema operativo, después de leer el contenido realiza la ejecución del primer filtro de la capa de procesamiento.

3. *SalidaMC2R*: Esta aplicación contiene la implementación requerida para establecer un *Socket* del tipo cliente. El último filtro de la capa de procesamiento ejecuta esta aplicación para transferir un contenido al siguiente Meta-Filtro.

4. *SalidaMC2SA*: Esta aplicación contiene la implementación que permite realizar la escritura de un contenido almacenado en un SMS al sistema de archivos del sistema operativo. El último filtro de la capa de procesamiento ejecuta esta aplicación para almacenar un contenido.

Componentes de la capa de procesamiento

A continuación se describe la parte de la implementación que corresponde a la capa de procesamiento del BB. Recordemos que un BB con un solo módulo de procesamiento corresponde

a la definición de filtro de procesamiento que se presentó en el capítulo anterior. De la misma manera tenemos que cuando un BB integra más de un módulo de procesamiento, entonces el BB corresponderá con nuestra definición de Meta-Filtro. A continuación se describen los componentes de software que realizan las actividades de E/S (en la capa de procesamiento) del BB, pudiendo trabajar en su modalidad de Filtro o Meta-Filtro.

- FiltroMC2MC: Aplicación que recibe datos a través de un SMS accedido por el BB. Los datos normalmente vendrán de las aplicaciones que gestionan la entrada desde la capa de acceso del BB, a través de un puerto de red (aplicación EntradaR2MC) o a través del sistema de archivos (aplicación EntradaSA2MC). Una vez que los datos se encuentran en control de FiltroMC2MC, podrán ser enviados a otro Meta-Filtro o Meta-Tubería a través de las aplicaciones SalidaMC2R o SalidaMC2SA según sea el tipo de interfaz de E/S que maneje el receptor.

- EntradaR2MC: Es la implementación del socket del servidor encargado de escuchar las peticiones de múltiples clientes que se conectan por la red, recibiendo los datos que llegan por esa interfaz y guardándolos en un SMS. La aplicación ejecuta a FiltroMC2MC pasándole los datos del SMS para que los pueda leer y procesar.

- EntradaSA2MC: Realiza la lectura de datos a través del sistema de archivos del sistema operativo host, coloca los datos leídos en un SMS y ejecuta el FiltroMC2MC pasándole los datos del SMS para que pueda recolectar los datos leídos para que los pueda leer y utilizar.

- SalidaMC2R: Obtiene los datos del SMS enviado por FiltroMC2MC y los transfiere por red al siguiente Meta-Filtro de la Meta-Tubería.

- SalidaMC2SA: Obtiene los datos del SMS enviado por FiltroMC2MC y los almacena en el sistema de archivos del sistema operativo huésped.

En la Figura 4.2 se observan las aplicaciones y bibliotecas que conforman el BB (ya sea como Filtro

o Meta-Filtro) y en la Figura 4.3 se ilustra la ubicación donde operan las aplicaciones previamente descritas dentro de nuestra propuesta arquitectónica para un Meta-Filtro.

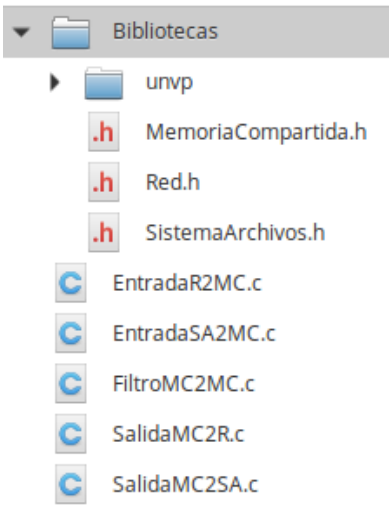


Figura 4.2: Bibliotecas y aplicaciones desarrolladas.

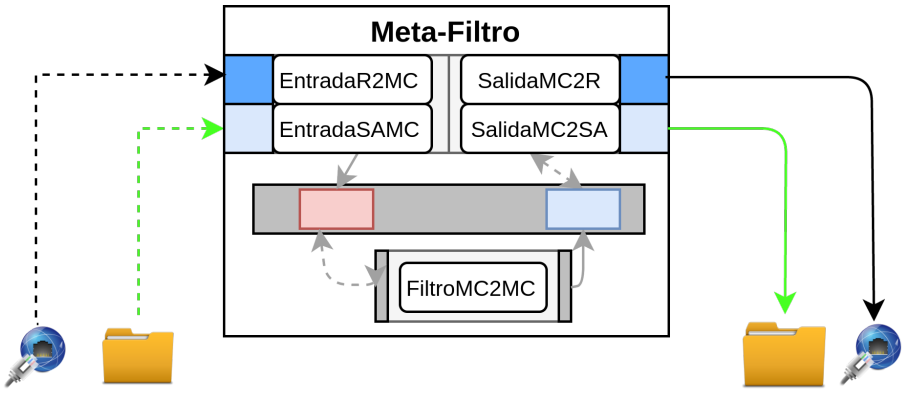


Figura 4.3: Vista del Meta-Filtro con las aplicaciones desarrolladas.

Con la creación de Meta-Filtros con múltiples filtros de procesamiento en su interior se consigue la *variedad* requerida por el método propuesto, además de que permite el manejo de grandes *volúmenes* de datos.

4.1.2 Constructor de imágenes de contenedor con tuberías embebidas

Para la implementación del constructor de CIs se debe de seleccionar una plataforma para el despliegue de contenedores. Para el desarrollo de nuestro constructor se desarrollaron los siguientes componentes:

Silo de imágenes de contenedor

Es un BB configurado con la aplicación EntradaR2MC habilitada para recibir peticiones PUT y GET a través de la dirección IP del sistema operativo huésped usando como puerto de escucha 8080. En la Figura 4.4 se pueden observar los componentes del silo, además, se muestran los procesos de PUT y GET. Es importante notar el polimorfismos que presenta la estructura del BB, como filtro o Meta-Filtro, que para el caso del silo de CIs se comporta como un Meta-Filtro (por contener más de un filtro), realizando procesos (filtros) específicos (PutCI, GetCI) que utilizan SMS para el intercambio de datos entre estos y las aplicaciones de entrada (R2MC o SA2MC) y salida (MC2R y MC2SA) para crear el flujo y transporte de datos desde la capa de acceso del Meta-Filtro al sistema de archivos o el recolector de CIs.

Los procesos realizados por las peticiones PUT y GET se muestran en la Figura 4.4 y son descritos a continuación:

- *PUT*: Petición encargada de realizar la carga de una CI desde el repartidor al silo. El silo recibe el contenido de una CI junto con sus metadatos (P1), la CI es colocada en un SMS (P2), el filtro *PutCI* realiza la lectura del SMS (P3), agrega los metadatos de la CI a la base de datos, lo aprueba y pasa el SMS sin modificar a la interfaz de salida (P4), *SalidaMC2SA* lee el SMS (P5) y almacena el contenido del SMS en el sistema de archivos del sistema operativo del contenedor (P6). El uso del filtro *PutCI* se debe a que se requiere realizar un proceso para la

adición de registros a la base de datos y estos no se pueden agregar en la capa de acceso ya que esta solo contiene las aplicaciones requeridas para la comunicación entre Meta-Filtros.

- *GET*: Después de que el recolector establece la conexión con el silo, realiza la petición de una CI mediante el nombre con el que fue almacenada. La aplicación *GetCI* realiza la lectura de la CI solicitada (G1), la coloca en un SMS (G2), *GetCI* añade la petición a la base de datos y pasa el SMS a la salida (G4), SalidaMC2R realiza la lectura del SMS (G5) y lo envía por la red al recolector (G6). El uso de *GetCI* es para la búsqueda de la CI solicitada dentro de la base de datos, por lo que se le considera un procesamiento y no debe incluirse en la capa de entrada del Meta-Filtro.

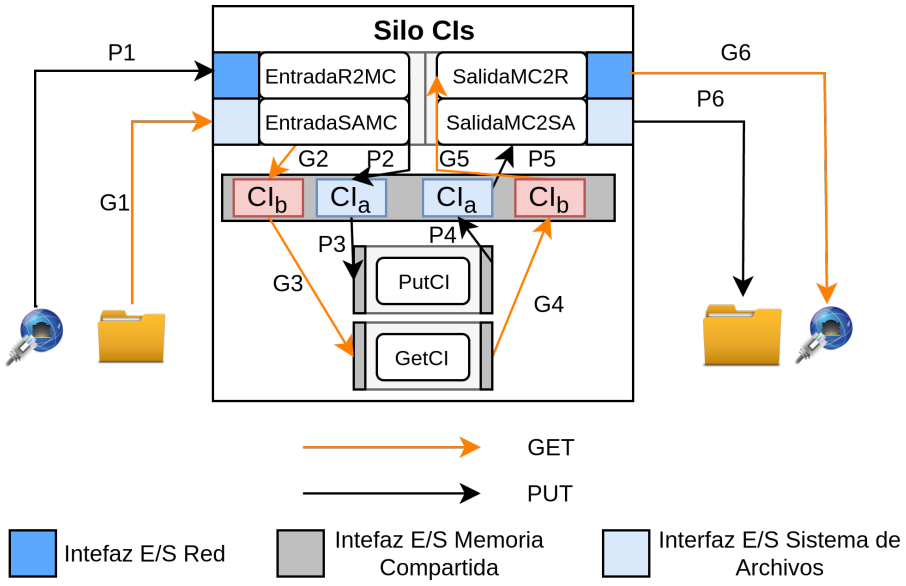


Figura 4.4: Componentes y flujos de datos del silo de CIs.

Repartidor de imágenes de contenedor

Es un filtro de procesamiento que permite almacenar la CI generada por el Integrador de CIs en el sistema de archivos del sistema operativo o realizar la petición PUT al silo de CIs para almacenarlo remotamente.

Recolector de imágenes de contenedor

Es un filtro de procesamiento encargado de recuperar CIs del silo. Contiene el filtro de procesamiento *Integrador de CIs con Aplicaciones* y el filtro de procesamiento *Repartidor de CIs*. Recibe como parámetros de entrada el nombre o ruta de la CI, la ruta de la tubería de procesamiento por incluir en la CI, el nombre de la CI equipada que se creará y el tipo de almacenamiento local y red. Una vez ingresados los parámetros estos son usados para invocar al *Integrador de CIs con tuberías de procesamiento*.

Integrador de imágenes de contenedor con Meta-Filtro

Es una aplicación (con arquitectura de Meta-Filtro) encargada de realizar la integración de una CI y un Meta-Filtro, para que posteriormente sean utilizados como unidad de procesamiento de una Meta-Tubería. Los parámetros requeridos para iniciar este proceso son: el nombre o la ruta de una CI, la ruta del Meta-Filtro de procesamiento que se le desea ingresar, nombre de la CI de salida y el tipo de salida local o red (en caso de que se vaya a guardar en el silo).

El proceso realizado por esta aplicación consiste en la lectura de una CI desde la plataforma de contenedores local u obtener una nueva CI del silo usando la aplicación *Recolector*, después se utiliza la aplicación *IntegradorCIMetaFiltro* que utiliza la CI obtenida en el paso anterior para lanzar un VC y copiar las aplicaciones del Meta-Filtro dentro de este VC para poder generar una nueva CI con el nombre proporcionado al inicio del proceso. La CI puede ser almacenada en la plataforma de contenerización local o ser transferida al silo usando la aplicación *Repartidor* dependiendo del tipo de salida requerido. El comportamiento de integración de CIs es mostrado en el Algoritmo 1

Algoritmo 1 Construcción de CIs con tuberías embebidas.

Entrada: Nombre CI origen $NombreCIOrigen$, Ruta de la Meta-Tubería $PathMetaTuberia$, Nombre CI Salida $nombreCISalida$, Tipo de Salida $tipoSalida$

1: **si** ExisteLocalmente($NombreCIOrigen$) **entonces**
2: $CIRecuperada = Recolector("Local", NombreCIOrigen)$
3: **si no**
4: $CIRecuperada = Recolector("Red", "GET", NombreCIOrigen)$
5: **fin si**
6: $CISalida = IntegradorCITuberia(CIRecuperada, PathMetaTuberia)$
7: **si** $tipoSalida == "Local"$ **entonces**
8: $Repartidor("Local", nombreCISalida, CISalida)$
9: **si no**
10: $Repartidor("Red", "PUT", nombreCISalida, CISalida)$
11: **fin si**

4.1.3 Planificador de conexiones

Para realizar la recolección de los datos de configuración usados para habilitar las interfaces de los Meta-Filtros contenerizados (CI+Meta-Filtro), se diseñó una página web que permite rellenar un formulario con los datos de configuración mencionados en la sección 3.3.3, en la que al presionar "generar archivo" se descarga el archivo de configuración correspondiente a los datos ingresados. El archivo de configuración contiene los datos en formato *JavaScript Object Notation* (JSON, por sus siglas en inglés). En la Figura 4.5 se puede observar el formulario de ingreso de datos de la página web desarrollada. En este formulario se ingresan los datos de configuración de una nueva BB-Contenerizada (ver definición en sección 3.3.3) que será utilizada como uno de los eslabones de una Meta-Tubería. En la figura 4.6 se muestra un ejemplo de archivo de configuración generado por la página web.

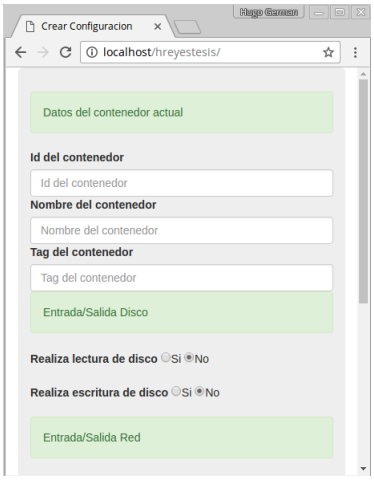


Figura 4.5: Página web para generar archivos de configuración.


```
{
  "contenedorActual": {
    "idContenedor":"123456789",
    "nombreContenedor":"Origen de datos",
    "tagContenedor":"data_origin",
    "puertoMinimo":"80",
    "puertoMaximo":"85",
    "lecturaDisco": {
      "habilitado":"si",
      "rutaLocal":"\/home\/rean\/Documentos\/Comparti
      dos",
      "rutaContenedor":"\/home\/Compartidos\/Comparti
      dosObtenidos"
    },
    "escrituraDisco": {
      "habilitado":"si",
      "rutaLocal":"\/home\/rean\/Documentos\/Comparti
      dosObtenidos\/",
      "rutaContenedor":"\/home\/Compartidos\/Comparti
      dosObtenidos"
    }
  },
  "contenedorSiguiente": {
    "habilitado":"si",
    "direccionIp":"192.168.0.1",
    "puertoMinimo":"86",
    "puertoMaximo":"95"
  },
  "contenedorAnterior": {
    "habilitado":"si",
    "direccionIp":"192.168.0.1",
    "puertoMinimo":"96",
    "puertoMaximo":"105"
  }
}
```


Figura 4.6: Contenido del archivo de configuración.

Al generar los archivos de configuración para nuestros VCs con BBs embebidas podemos establecer las conexiones entre ellos y habilitar las interfaces requeridas para establecer un flujo de datos continuo formando una Meta-Tubería de procesamiento. En la figura 3.7 se mostró un

ejemplo de una Meta-Tubería de procesamiento obtenida mediante la interconexión de contenedores usando el archivo de configuración propuesto.

El archivo de configuración es leído dentro del VC y una aplicación en su interior se encarga de habilitar las interfaces de E/S. Para indicar al contenedor qué ruta de enlace tendrá con la máquina que lo hospeda se requiere establecer un *data volume*[19] que es un directorio designado especialmente dentro de los VCs que pasa por alto el *Sistema de Archivos Unión* propio de los contenedores [1] y utiliza el proporcionado por el sistema de archivos del sistema operativo huésped.

4.1.4 Constructor de Meta-Tuberías

Es un Meta-Filtro que debe ubicarse en la infraestructura[2] donde serán desplegados los contenedores que contienen las BB-Contenerizadas que formarán las Meta-Tuberías.. Este Meta-Filtro recibe como parámetros de entrada los datos de la(s) CIs que conforman parte de la Meta-Tubería que deben ser ejecutados en la infraestructura donde este se encuentra, cada CI debe tener su respectivo archivo de configuración generado previamente con la página web desarrollada. El Constructor de Meta-Tuberías ejecuta cada CI y crea un contenedor habilitando los puertos y Volúmenes requeridos por el archivo de configuración, en seguida este archivo es almacenado dentro del contenedor en ejecución. Después de almacenar el archivo de configuración, desde el interior del contenedor se ejecuta la aplicación que actualiza los datos por defecto de las aplicaciones de la capa de entrada permitiendo así habilitar el contenedor como Meta-Filtro funcional de la Meta-Tubería. El proceso de construcción de Meta-Tuberías es mostrado en el Algoritmo 2.

[1] Sistema de archivo Unión, o UnionFS, es un sistema de archivos que opera bajo la creación de capas, haciéndolas muy ligeras y rápidas. Docker utiliza UnionFs para proveer los bloques de construcción para contenedores.

[2] Equipos de cómputo con la plataforma de despliegue de contenedores instalada

Algoritmo 2 Construcción de Meta-Tuberías.

Entrada: Lista de NombresCIs $NombresCIs = \{NombreCi_1, NombreCi_2, \ldots, NonbresCi_n\}$,

Parámetros de configuración CIs $ParametrosConfCIs = \{Param_1, Param_2, \ldots, Param_n\}$

1: $i = 1$

2: **mientras** $i <= n$ **hacer**

3: **si** ExisteLocalmente($NombreCI_i$) **entonces**

4: $CIRecuperada = Recolector("Local", NombreCI_i)$

5: **si no**

6: $CIRecuperada = Recolector("Red","GET", NombreCI_i)$

7: **fin si**

8: $archivoConfiguracion = GenerarArchivoConfiguracion(Param_i)$

9: $idContenedor = EjecutarContenedor(archivoConfiguracion)$

10: $copiarArchivoAContenedor(idContenedor, archivoConfiguracion)$

11: $ingresarAlContenedor(idContenedor)$

12: $realizarConfiguracionCapaEntrada(archivoConfiguracion)$

13: $habilitarCapaEntrada(archivoConfiguracion)$

14: $i+ = 1$

15: Salir del contenedor

16: **fin mientras**

4.2 Técnica divide y encapsula

Durante el desarrollo de este trabajo de tesis se propuso el esquema *divide y encapsula* para conseguir la característica de *velocidad* de procesamiento mediante el uso de Meta-Tuberías de procesamiento paralelas. En esta sección se realiza la descripción de la implementación de este esquema utilizando los componentes de la arquitectura propuesta.

4.2.1 Implementación de los filtros para el esquema divide y encapsula

Las técnicas de segmentación e integración de datos en forma paralela utilizando Meta-Tuberías es conseguida por los filtros de procesamiento de segmentación e integración descritos a continuación.

Filtro de Segmentación

El filtro de segmentación realiza la división del contenido de los datos de entrada $|F|$ en s segmentos que son distribuidos a diferentes Meta-Tuberías. Para cada segmento de s se crea un hilo de ejecución el cual es parte de una Meta-Tubería que aplica el mismo proceso a cada segmento.

Los tamaños de los segmentos generados corresponden a la cantidad resultante de realizar la división entera de $|F|/s$. En caso de que la división produzca un residuo este se encontrará en el rango de $1 \leq s \leq n-1$ por lo que se les podría agregar un byte de más desde el primer segmento hasta el penúltimo.

El segmentador propuesto realiza el cálculo del tamaño requerido para cada segmento de forma individual por lo que se le indica cual es el segmento deseado de un contenido, así como salida que será utilizada para transportar a dicho segmento. Para generar cada segmento se crea un *hilo* que se ejecuta de forma paralela a los demás el cual realiza la segmentación y envío del segmento. El proceso del filtro de segmentación es mostrado en el Algoritmo 3.

Filtro de Integración

El proceso de integración toma como parámetros de entrada la información de un dato de entrada $|F|$ segmentado s veces usando el filtro de segmentación propuesto. El filtro de integración realiza

Algoritmo 3 Proceso de segmentación de archivos.

Entrada: Datos de entrada $|F|$, número de segmentos s, lista de salidas $salidas = \{salida_1, salida_2, \ldots, salida_s\}$

1: $segmentoAcual = 1$
2: **mientras** $segmentoAcual <= s$ **hacer**
3: $crearHiloEjecucion()$
4: $residuo = |F| \mod s$
5: **si** $(residuo == 0)$ **entonces**
6: $bytesPorLeer = |F|/s$
7: **si no**
8: **si** $(segmentoAcual \leq residuo)$ **entonces**
9: $bytesPorLeer = |F|/s + 1$
10: **si no**
11: $bytesPorLeer = |F|/s$
12: **fin si**
13: **fin si**
14: **si** $(segmentoAcual > 1)$ **entonces**
15: $posicionInicialLectura = bytesPorLeer * (segmentoAcual - 1)$
16: **si no**
17: $posicionInicialLectura = 0$
18: **fin si**
19: $contenidoSegmento = leerSegmento(|F|, posicionInicialLectura, bytesPorLeer)$
20: $enviarSegmento(contenidoSegmento, salida_{segmentoAcual})$
21: $terminarHilo()$
22: **fin mientras**

la solicitud y descarga los s segmentos desde las Meta-Tuberías usadas para el procesamiento o los almacenes donde fueron dispersados utilizando el identificador del contenido $|IdArchivo|$ que se desea reconstruir. Una vez que los s segmentos son obtenidos se procede realizar su integración en el orden usado en la segmentación y así generar el contenido original $|F|$. El proceso de integración se muestra en el Algoritmo 4.

4.3 Filtros y Meta-Filtros de procesamiento utilizados

Para la realización de la pruebas fueron desarrollados y utilizados los siguientes filtros de procesamiento y propuestas.

Algoritmo 4 Proceso de integración de archivos segmentados.

Entrada: Identificador del archivo $IdArchivo$, Tamaño del archivo $|T_F|$, número de segmentos s, lista de orígenes $origenes = \{origen_1, origen_2, \ldots, origen_s\}$, ruta de salida $rutaSalida$

1: $i = 1$
2: $segmentosRecuperados = [s]$
3: **mientras** $i <= s$ **hacer**
4: $i+ = 1$
5: $crearHiloEjecucion()$
6: $segmentosRecuperados[i-1] = recuperarSegmento(i, idArchivo)$
7: $terminarHilo()$
8: **fin mientras**
9: $contenidoIntegrado = reservarMemoria(T_F)$
10: **mientras** $j < s$ **hacer**
11: $contenidoTemporal = integrarContenido(segmentosRecuperados[j], segmentosRecuperados[j+1])$
12: $contenidoIntegrado = integrarContenido(contenidoIntegrado, contenidoTemporal)$
13: **fin mientras**
14: $almacenarContenido(contenidoIntegrado, idArchivo, rutaSalida)$

4.3.1 Confidencialidad

Se utilizó el componente *Confidentiality DC-BB* obtenido de Sacbe[33] compuesto por tres unidades de procesamiento (PU, por sus siglas en inglés): Privacidad (PU_P), Integridad (PU_I) e Integración (PU_{IP}) que permiten realizar los procesos de cifrado, firma digital y empacado de datos respectivamente.

4.3.2 Compresión

Fue utilizado la unidad de procesamiento de compresión utilizada en Sacbe[33]. Esta unidad de procesamiento realiza la compresión de los datos utilizando la biblioteca de compresión Zip que está incluida en el lenguaje de programación Java.

4.3.3 IDA

IDA es un algoritmo, basado en técnicas de corrección de errores, que se utiliza para separar paquetes de datos evitando que estos últimos puedan ser entendidos de manera independiente por un lector no deseado. Este algoritmo permite generar n piezas de un archivo F y reconstruirlo utilizando cualesquiera de m piezas, donde $m < n$. Lo importante de este algoritmo es que cada pieza es de tamaño $|F|/m$, donde $|F|$ corresponde al número de bytes de F. Por lo tanto el total de bytes requerido por todas las n piezas es $(n/m) * |F|$. Puesto que se debe escoger n de tal forma que n/m sea cercano a 1, IDA no requiere gran cantidad de espacio extra. IDA se puede catalogar como un método de corrección de errores, en el cual se agregan bytes extras a cada pieza antes de ser dispersadas, de tal forma que después de la ocurrencia de cualesquiera k errores en $n-m$ piezas, el archivo F puede ser reconstruido [51].

IDA está compuesto por dos procesos. Los procesos que lo componen se llaman codificación y decodificación. A continuación se describen los dos procesos individualmente.

- Codificación: Realiza el proceso de codificación de un archivo de entrada $|F|$ generando las n piezas a partir de este y las dispersa en las n salidas correspondientes. En la Figura 4.7 se muestra el funcionamiento del proceso de codificación. El proceso inicia cuando se le indica el nombre del archivo que se desea codificar y las n salidas (C_1), el contenido del archivo $|F|$ es leído del sistema de archivos del sistema operativo local (C_2), a $|F|$ se le aplica el proceso de codificación (C_3), las n piezas generadas son enviadas a los n almacenes(C_4) terminando el proceso de IDA codificación. En la Figura 4.8 se muestra el funcionamiento del proceso de codificación.

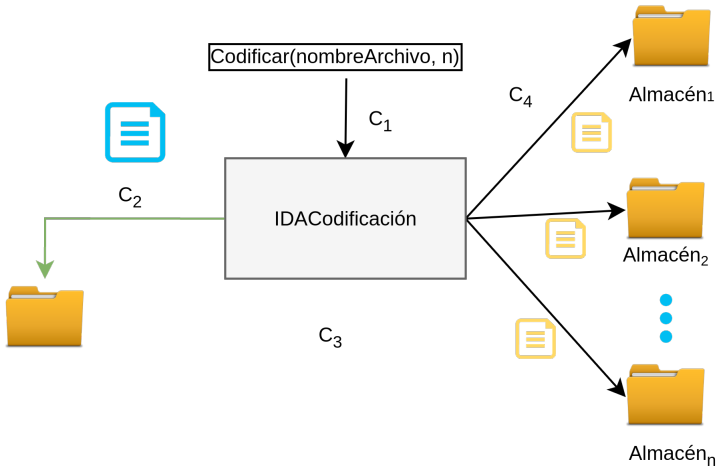


Figura 4.7: Proceso IDA Codificación.

- Decodificación: Obtiene m piezas cualesquiera de las s generadas por la codificación, aplica la función de decodificación y almacena el archivo decodificado $|F|$. El proceso de decodificación es mostrado en la Figura 4.8 Este proceso inicia cuando se le indica el id del archivo codificado y los s lugares de almacenamiento (D_1), se recuperan m piezas de las s disponibles (D_2),donde $m < s$, se aplica la decodificación con las m piezas recuperadas (D_3) y se almacena el archivo decodificado (D_4).

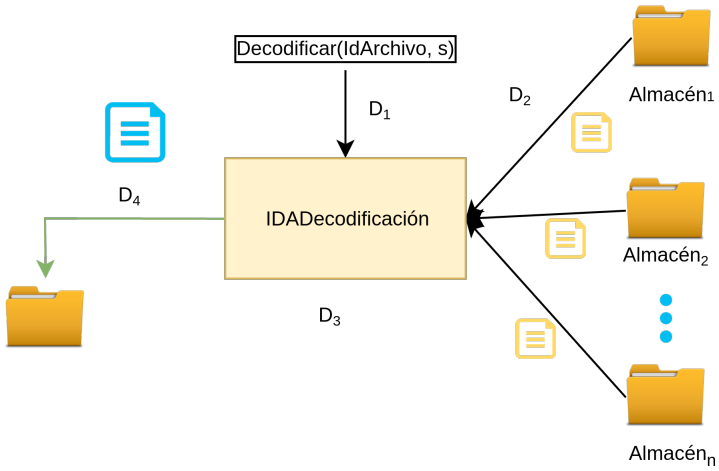


Figura 4.8: Proceso IDA decodificación.

Configuración IDA. El algoritmo IDA se utilizó con una configuración de $n = 5$ y $m = 3$. Esta configuración consume un 66.67 % adicional de almacenamiento $((5/3 * 1) - 1)$. Para ejemplificar el espacio total requerido al codificar un archivo con IDA se muestra la Tabla 4.1 en la que se observan los valores obtenidos al aplicar IDA(5,3) a distintos archivos.

Tabla 4.1: Ejemplo de archivos a los que se les aplicó IDA(5,3).

Tamaño del archivo ($\|F\|$)	Número de piezas (n)	Piezas requeridas para reconstruir $\|F\|$ (m)	Tolerancia a fallos ($n - m$)	Tamaño de la pieza ($\|F\|/m$)	Espacio requerido para almacenar las n piezas ($n/m * \|F\|$)
1MB	5	3	2	0.33MB	1.67MB
10MB	5	3	2	3.33MB	16.67MB
100MB	5	3	2	33.33MB	166.67MB
1000MB	5	3	2	333.33MB	1666.67MB

Los algoritmos utilizados para realizar la codificación y decodificación con la configuración IDA(5,3) son mostrados en el apéndice A.

4.3.4 IDA usando el Sistema de Archivos

Filtros de procesamiento que encapsulan los procesos de codificación y decodificación con una configuración IDA(5,3), utilizando el sistema de archivos como medio de comunicación entre procesos y mecanismo de almacenamiento/recolección de las piezas generadas/utilizadas por IDA. Los nombres de los filtros son: IDASACodificación y IDASADecodificación.

4.3.5 IDA usando la Memoria

Filtros de procesamiento que encapsulan los procesos de codificación y decodificación dentro del BB propuesto. Estos filtros utilizan la configuración de IDA(5,3) y usan los SMS como medio de comunicación entre procesos y las interfaces de salida de red y sistema de archivos para su dispersión/almacenamiento. La salida de las piezas se realiza de forma secuencial, es decir realiza el envío de las piezas de la primera hasta la última una tras otra. Los nombres de los filtros utilizados son: IDAMCodificación y IDAMDecodificación.

4.3.6 Almacén codificación

Filtro desarrollado usando el BB propuesto que recibe los metadatos y contenido enviado por múltiples clientes a través de la red mediante la petición *PUT* y los almacena dentro del sistema de archivos del sistema operativo anfitrión[3] a través de un volumen compartido de datos con el contenedor.

4.3.7 Almacén decodificación

Filtro desarrollado usando el BB propuesto que recibe los metadatos de archivos a través de la red mediante la petición *GET*, los localiza dentro sistema de archivos del sistema operativo huésped y los envía a los clientes que los solicitaron. Este BB debe enlazar al volumen utilizado para el almacenamiento de los datos codificados por *Almacén codificación*.

Se decidió separar la carga y descarga de datos de los almacenes en filtros diferentes debido al número de peticiones que deben ser atendidas al utilizar el esquema *divide y encapsula*. El separar los comportamientos de carga y descarga permite el lanzamiento de múltiples contenedores dedicados para cada proceso ejecutados en cores distintos, evitando así el costo producido por acceder al mismo

[3]El sistema operativo que ejecuta el contenedor

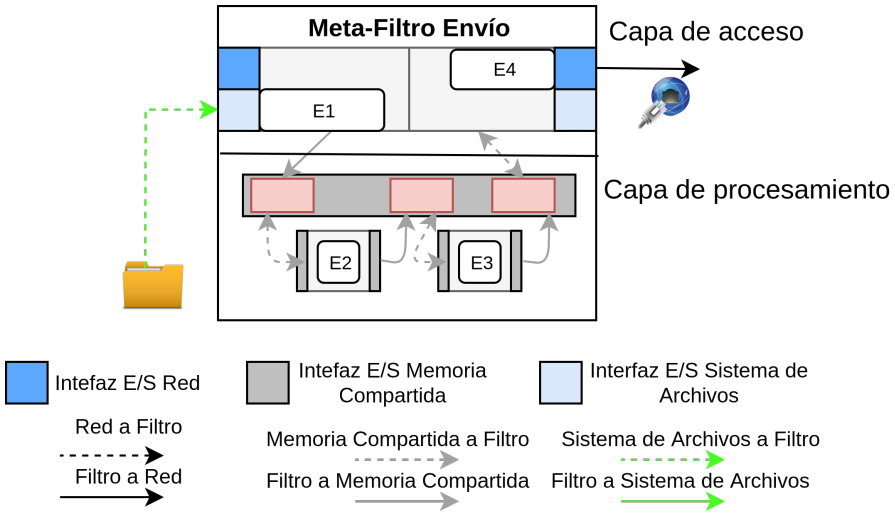


Figura 4.9: Proceso realizado por el Meta-Filtro Envío.

recurso (core) agilizando el tiempo de respuesta de las Meta-Tuberías que lo utilizan.

4.3.8 Meta-Filtro Origen

Meta-Filtro generado con el propósito de realizar la lectura de archivos de distintos tamaños desde el sistema de archivos del sistema operativo anfitrión y transferirlos a través de la red utilizando las APIs de las interfaces de comunicación desarrolladas para este trabajo de tesis. Las operaciones realizadas por este Meta-Filtro son mostradas en la Figura 4.9 y descritas a continuación: Leer el contenido del sistema de archivos y colocarlo en un SMS en la capa de acceso del Meta-Filtro (E1), en la capa de procesamiento el primer filtro toma el contenido del SMS y lo coloca en la memoria para realizar un resumen MD5 de los datos (E2), el contenido del SMS es leído y enviado a la interfaz de salida correspondiente de la capa de acceso (E3), se obtiene el contenido del SMS y es transferido a través de la red mediante el *socket* del cliente haciendo una petición PUT (E4).

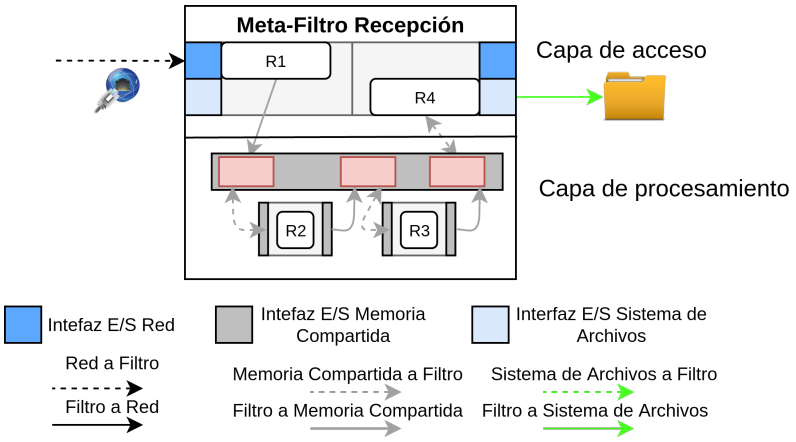


Figura 4.10: Proceso realizado por el Meta-Filtro Recepción.

4.3.9 Meta-Filtro Recepción

Meta-Filtro generado con el propósito de aceptar peticiones utilizando la aplicación *EntradaR2MC* que habilita un *socket* de servidor que acepta peticiones y datos de múltiples clientes y los almacena en el sistema de archivos del sistema operativo huésped. Las operaciones realizadas por este Meta-Filtro son mostradas en la Figura 4.10 y descritas a continuación: Aceptar la petición PUT del Meta-Filtro *Envío* y colocarlo en un SMS desde la capa de acceso para que pueda ser accedido desde la capa de procesamiento (R1), obtener el SMS y colocarlo en la memoria para que el filtro pueda calcular el resumen MD5 y después colocarlo nuevamente en un SMS (R2), se realiza la verificación de integridad comparando el resumen MD5 calculado y el recibido por el VC de Meta-Filtro anterior y es enviado a la interfaz de salida correspondiente de la capa de acceso (R3) y se lee el SMS y se escriben los datos verificados en el sistema de archivos del sistema operativo anfitrión (R4).

4.4 Soluciones implementadas en el prototipo

Para la evaluación del método propuesto se realizó la implementación de cuatro Meta-Tuberías que aplican el mismo proceso de codificación a los datos pero utilizan técnicas de procesamiento distintas. Las Meta-Tuberías utilizadas son descritas a continuación.

4.4.1 Bloque de construcción paralelo

Son los Meta-Filtros PBB Segmentador IDAM y PBB Integrador IDAM que implementan los algoritmos de segmentación e integración de la técnica *divide y encapsula* añadiéndoles el filtro de IDAMCodificación e IDAMDecodificación respectivamente colocándolos dentro de VCs de Meta-Filtro que generan s número de tuberías de procesamiento en diferentes hilos de ejecución, donde $i \geq 2$. Cada hilo creado será ejecutado utilizando uno de los cores disponibles para el VC de Meta-Filtro utilizado.

La solución está compuesta de cuatro CIs de Meta-Filtros, dos usadas para el proceso de codificación: PBB Segmentador IDAM y almacén codificación; y dos para el proceso de decodificación: PBB Integrador IDAM y almacén decodificación. Estas CIs de Meta-Filtros son utilizadas para la generación de diez VCs de Meta-Filtros, cinco para la codificación: un PBB Segmentador IDAM y cuatro almacenes codificación; y cinco para la decodificación: un PBB Integrador IDAM y cuatro almacenes decodificación.

Para el proceso de codificación se generó un PBB que encapsula en su interior el filtro segmentador y IDAMCodificación produciendo el Meta-Filtro PBB Segmentador IDAM. El proceso realizado PBB Segmentador IDAM es mostrado en la Figura 4.11 y descrito a continuación: se realiza la lectura de los datos (1) se indica el número de segmentos s en los que será dividido el archivo (2), se establecen las s tuberías por utilizar para el proceso de IDAMCodificación (una tubería por segmento) (3), al codificar cada segmento se generan cinco piezas codificadas (4), las piezas son enviadas a los VCs de Meta-Filtro almacén codificación (5), los almacenes colocan las piezas en su sistema de archivos (6). De las cinco piezas generadas por IDAMCodificación, una de las piezas es almacenada en el sistema de archivos local y las otras cuatro son enviadas a través de la red a los VCs de Meta-Filtro almacén de codificación.

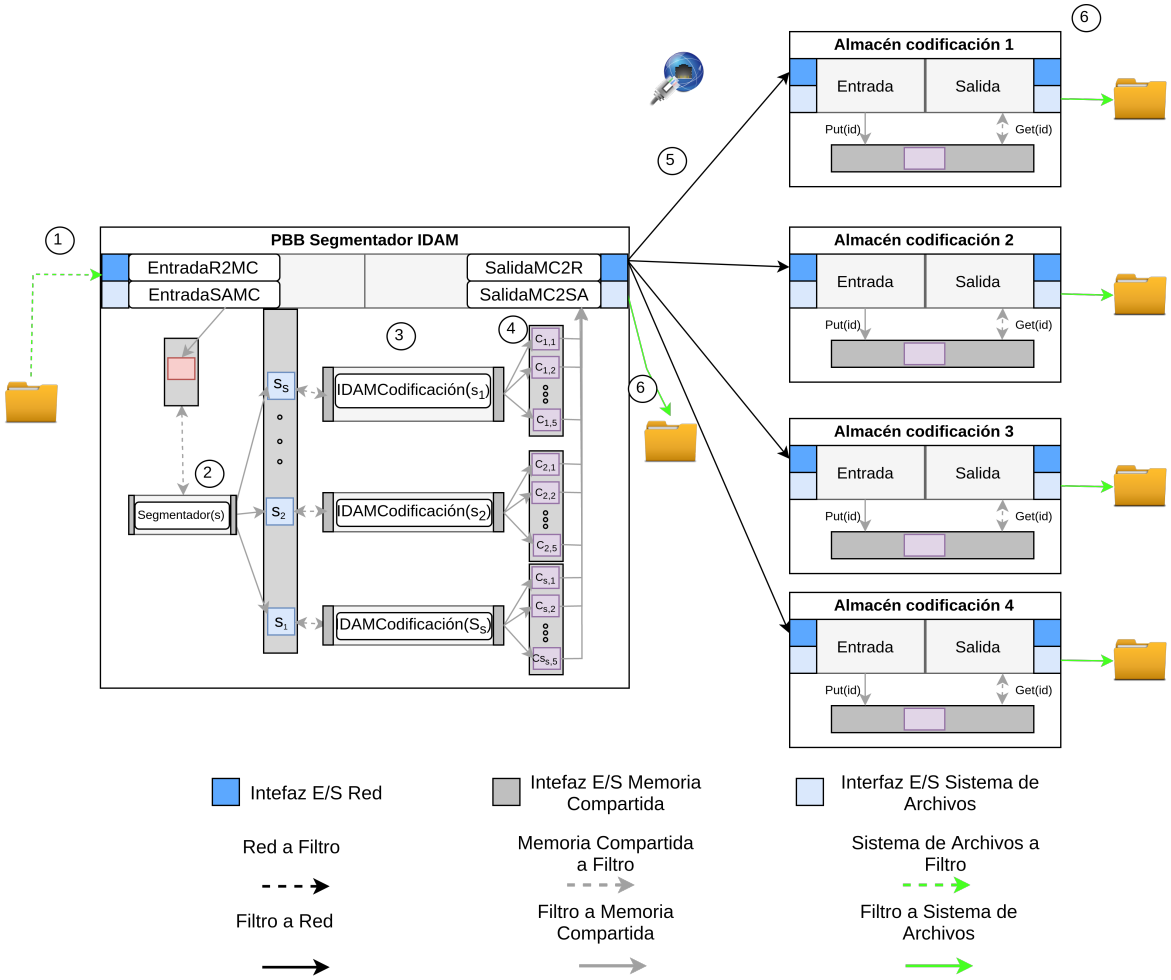


Figura 4.11: Proceso de codificación utilizando los Meta-Filtros PBB Segmentador IDAM y almacén codificación.

Para el proceso de decodificación se generó un PBB que encapsula en su interior el filtro integrador y IDAMDecodificación produciendo el Meta-Filtro PBB Integrador IDAM. El proceso realizado PBB Integrador IDAM es mostrado en la Figura 4.12 y descrito a continuación: se le indica al integrador el archivo por decodificar y el número de segmentos que fueron usados en su proceso de codificación (1), al conocer estos datos se ejecutan las s tuberías por utilizar (una por hilo) e iniciar la petición de las piezas de cada segmento mediante IDAMDecodificación (2), enseguida, IDAMDecodificación solicita a su sistema de archivos una pieza y a los VC de Meta-Filtros de almacén decodificación dos

más (3), los almacenes buscan en su sistema de archivos la pieza solicitada (4), si esta se encuentra es enviada al proceso IDAMDecodificación que lo solicito (5), IDAMDecodificación obtiene las tres piezas y aplica el proceso de decodificación (6), los s segmentos decodificados son devueltos al integrador quien realiza la integración de los s segmentos en uno (7), el archivo integrado es leído por la salida del sistema de archivos (8) y es almacenado en el sistema de archivos (9).

Las características de esta solución son: el uso de SMS para la comunicación entre filtros de procesamiento que disminuye el tiempo de Lectura/Escritura de datos incrementando la velocidad de procesamiento de la Meta-tubería, la utilización de un VC que encapsula el PBB que genera múltiples tuberías utilizando cores distintos para cada una y la utilización de la red para el transporte de los datos.

Las dependencias de esta solución son el uso de un sistema operativo Linux que permita el manejo de SMS además del uso de la biblioteca *pthreads* para la generación de hilos y el compilador GCC para realizar la compilación de las aplicaciones que componen las BBs y PBBs utilizadas.

4.4.2 Multicontenedor

Es la solución que implementa el algoritmo *divide y encapsula* para conseguir *velocidad* de procesamiento al utilizar múltiples Meta-Tuberías distribuidas para el procesamiento de los datos. Las Meta-Tuberías generadas permiten la codificación y decodificación de datos usando los Meta-Filtros de IDAMCodificación y IDAMDecodificación en distintas infraestructuras utilizando los PBBs como origen (PBB Segmentador) y consumidor (PBB Integrador).

Las CIs de Meta-Filtros utilizadas son seis, tres para la codificación: PBB segmentador, IDAMCodificación y almacén codificación; y tres para la decodificación: PBB Integrador,

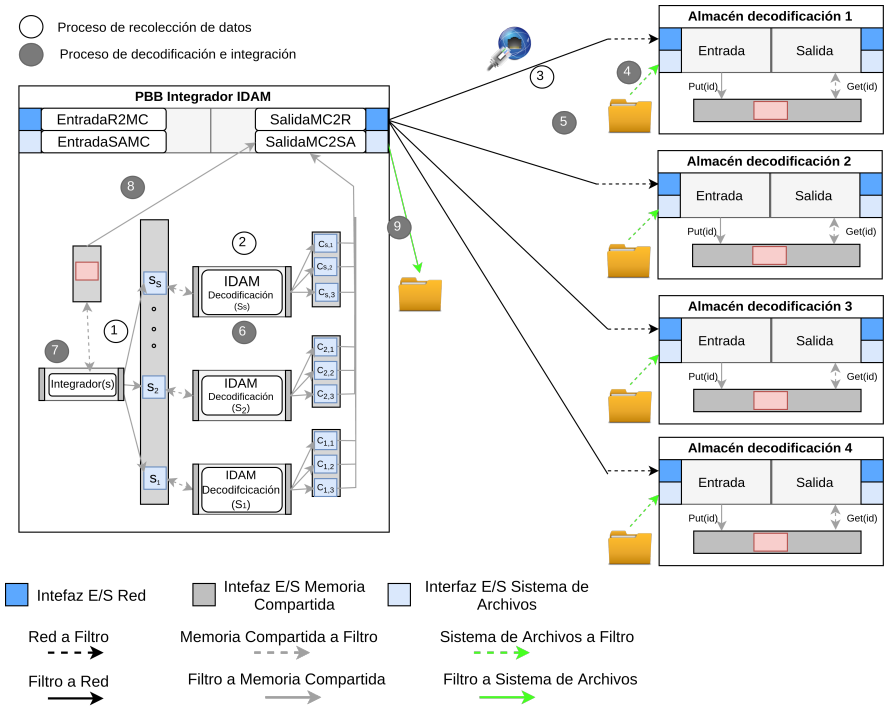


Figura 4.12: Proceso de decodificación utilizando los Meta-Filtros PBB Integrador IDAM y almacén decodificación.

IDAMDecodificación y almacén decodificación.

Para la prueba de codificación se deben generar los siguientes VCs de Meta-Filtro: un PBB de origen que realiza la segmentación de los datos de origen (hasta cinco segmentos), cinco IDAMCodificación y cinco almacenamiento codificación. Estos VCs de Meta-Filtro pueden ser distribuidos y configurados en diferentes infraestructuras.

El proceso de codificación que utiliza el algoritmo *divide y encapsula* es mostrado en la Figura 4.13. Este proceso consiste en que el VC de Meta-Filtro PBB Segmentador realiza la lectura de datos a través del sistema de archivos (1), segmenta los datos con un valor de s entre 1 y 5, cada segmento es colocado en en SMS para poder realizar el envió de los datos a través de la red al VC de Meta-Filtro IDAMCodificación (2), los datos de cada segmento son recibidos y codificados por

los VCs de Meta-Filtro IDAMcodificación, este proceso genera 5 piezas codificadas, dichas piezas son transferidas a las 5 VCs de Meta-Filtros de almacenamiento codificación a través de la interfaz de red (3) y son almacenados en el sistema de archivos del VC enlazado al sistema de archivos del sistema operativo anfitrión (4).

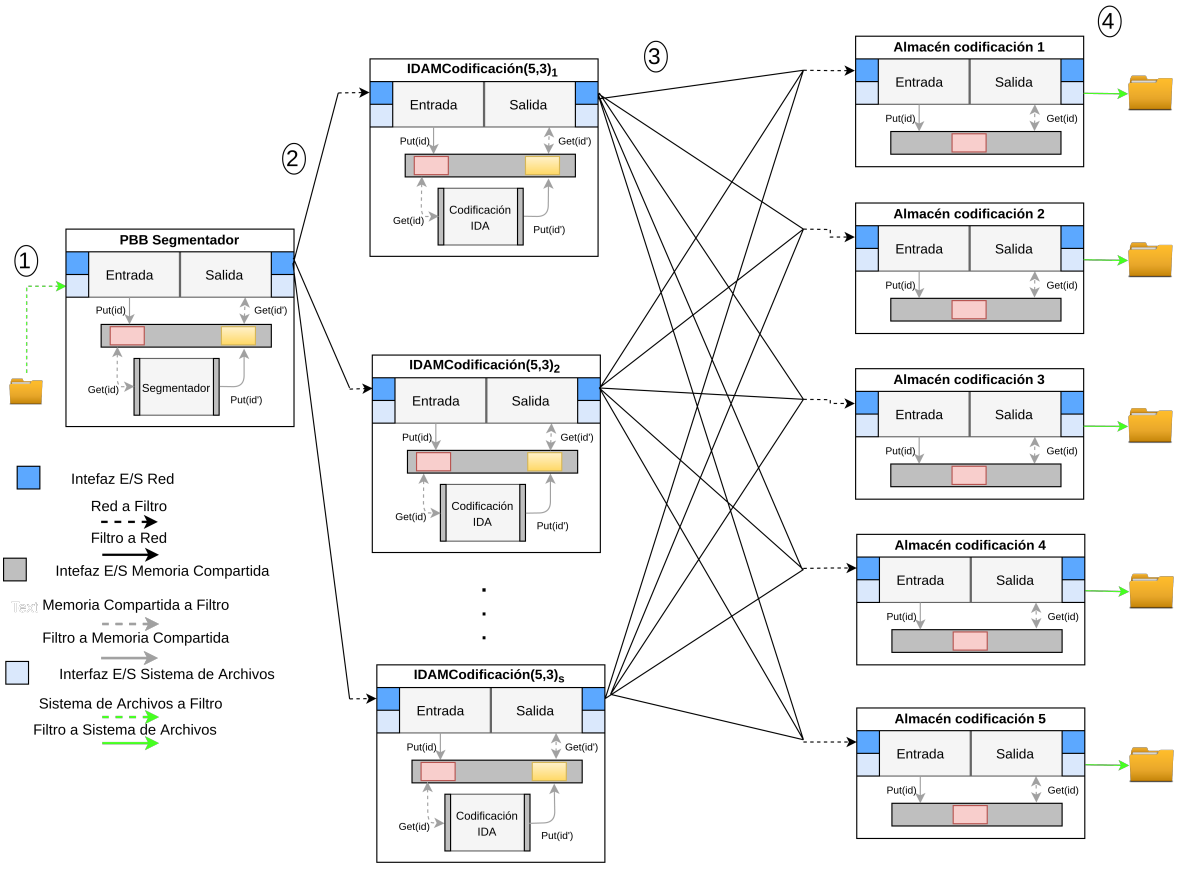


Figura 4.13: Proceso de codificación utilizando la solución *multicontenedor*.

Para la prueba de decodificación se requiere el despliegue de los siguientes VCs de Meta-Filtros: un PBB de integración que permite realizar la integración de un dato que fue segmentado hasta con un nivel de segmentación de cinco, cinco IDAMDecodificación y cinco almacenes de decodificación distribuidos y configurados en diferentes o en la misma infraestructura. El proceso realizado para la decodificación inicia en el consumidor donde el integrador realiza la solicitud de los s segmentos

decodificados a los VCs de Meta-Filtro IDAMDecodificación (1), IDAMDecodificación obtiene tres de las cinco piezas desde los VCs de Meta-Filtros de almacenamiento decodificación (2), al obtener las piezas requeridas se inicia el proceso de IDAMDecodificación y se obtienen los datos del segmento solicitado(3), estos datos son devueltos al consumidor(4) en su interior realiza la concatenación de segmentos en uno solo y este es almacenado en su sistema de archivos.

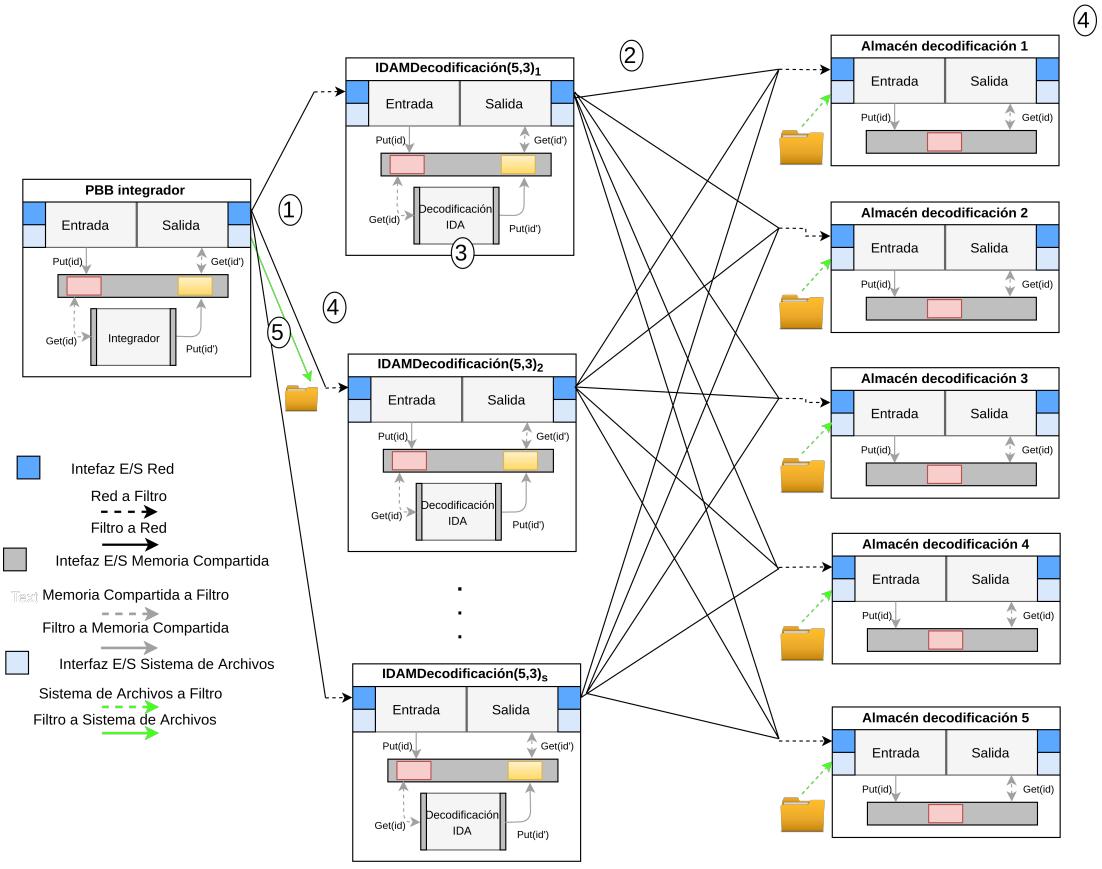


Figura 4.14: Decodificación *Multicontenedor*.

Las características de esta solución son: el uso de SMS para la comunicación entre filtros de procesamiento que disminuye el tiempo de Lectura/Escritura de datos incrementando la velocidad de procesamiento de la Meta-tubería, la utilización de múltiples VCs con la misma BB utilizando cores distintos para cada una, uso de PBB para la división de la carga de trabajo en los contenedores

previamente lanzados y la utilización de la red para el transporte de los datos.

Las dependencias de esta solución son el uso de un sistema operativo Linux que permita el manejo de SMS además del uso de la biblioteca *pthreads* para la generación de hilos y el compilador GCC para realizar la compilación de las aplicaciones que componen las BBs y PBBs utilizadas.

4.4.3 Secuencial

Solución que implementa los procesos de codificación y decodificación utilizando la versión secuencial del algoritmo IDAM que será utilizado como comparativa con las demás soluciones usadas para la evaluación. Para esta solución existen dos versiones: multicontenedor local o en red y PBB. La distribución de las versiones presentadas a continuación corresponde a la distribución de las soluciones *Multicontenedor* y *PBB* que serán utilizadas como comparación. A continuación son descritas las dos versiones de secuencial desarrolladas.

Secuencial para multicontenedor local y red

Esta versión se conforma por seis Meta-Filtros encapsulados en seis VCs de Meta-Filtros, tres para el proceso de codificación: Origen, IDAMCodificación y almacén; y tres para el proceso de decodificación: Consumidor, IDAMDecodificación y almacén.

Para el proceso de codificación se desplegaron siete VCs de Meta-Filtro: un origen, cinco IDAMCodificación y cinco almacenes codificación. En la Figura 4.15 se muestra la Meta-Tubería de codificación, en la cual el *Origen* realiza la lectura de datos de origen a través del sistema de archivos y los coloca en SMS (1) para poder realizar el envío de los datos a través de la red a *IDAMCodificación* (2), los datos son recibidos por *IDAMCodificación* son codificados generando 5 piezas, las cinco piezas son transferidas a 5 almacenes de codificación usando la interfaz de red (3)

y son almacenados en el sistema de archivos (4).

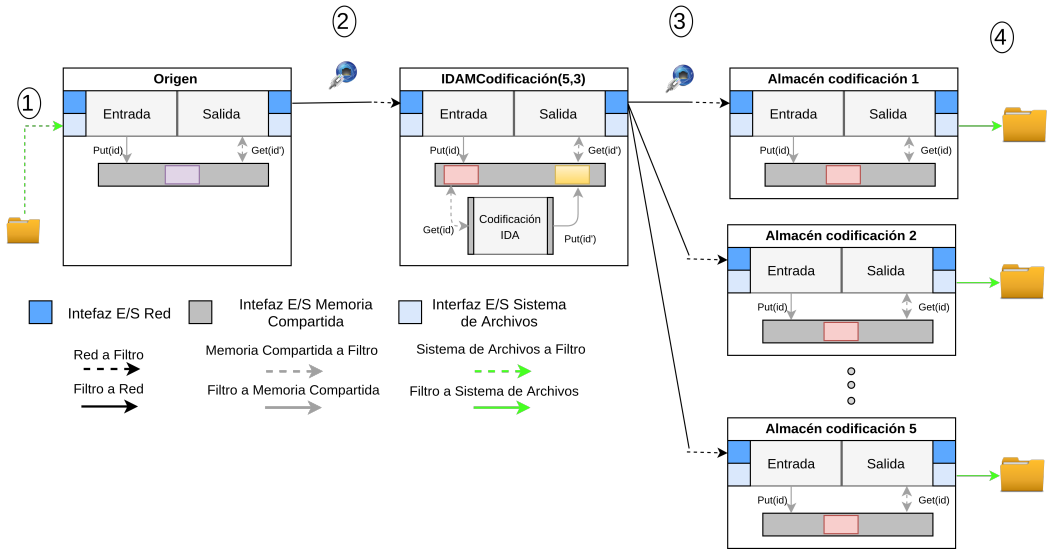


Figura 4.15: Proceso de codificación secuencial para la solución munticontenedor.

Para el proceso de decodificación se desplegaron siete VCs de Meta-Filtro: un consumidor, cinco IDAMDecodificación y cinco almacenes decodificación. En la Figura 4.16 se muestra la el proceso de decodificación el cual es descrito a continuación: El proceso inicia en el *consumidor* solicitando los datos decodificados a *IDAMDecodificación* (1), *IDAMDecodificación* obtiene tres de las cinco piezas generadas por la codificación desde los almacenes de decodificación (2), al obtener las piezas requeridas se inicia el proceso de decodificación y se obtienen los datos originales (3), estos datos son devueltos al consumidor(4) y son almacenados en su sistema de archivos.

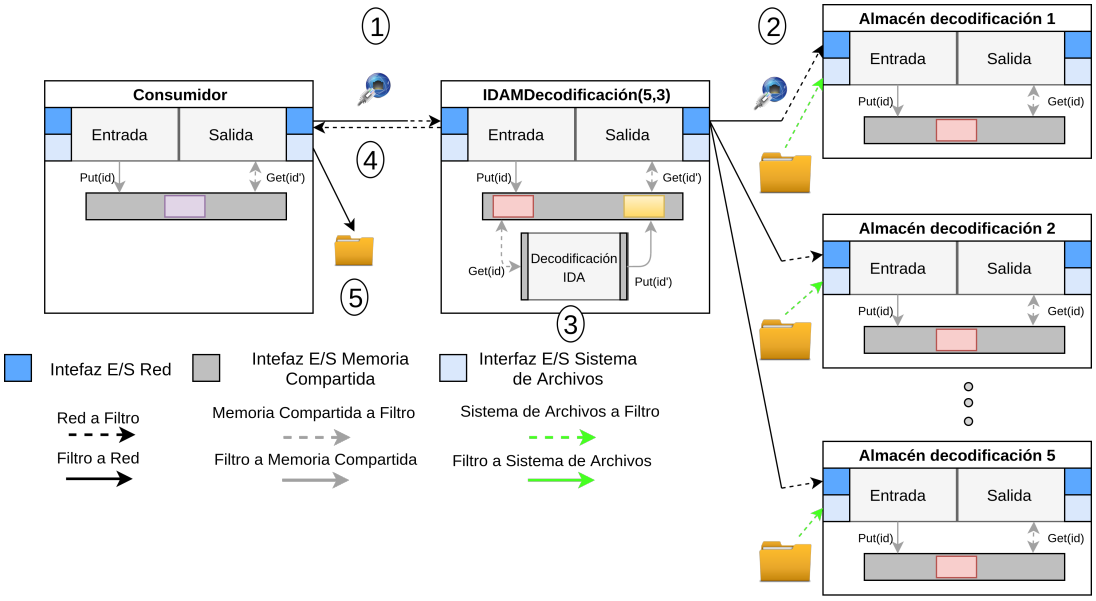


Figura 4.16: Proceso de decodificación secuencial para la solución munticontenedor.

La diferencia entre la versión local y en red es el despliegue de los VC de Meta-Filtros en un solo equipo o múltiples equipos distribuidos respectivamente. Además, el uso del VC de Meta-Filtro *origen* es agregar el costo de comunicación por red que genera *multicontenedor* con quien será comparado.

Secuencial para caja negra paralela

Es la versión del PBB que realiza el procesamiento de los datos de entrada sin segmentar, y solo genera una tubería y las piezas generadas son enviadas/obtenidas desde los almacenes. Cuenta con cuatro VC de Meta-Filtros, dos para la codificación: origen y almacén; y dos para decodificación: consumido y almacén.

En la Figura 4.17 se muestra el proceso de codificación de esta versión de *secuencial* que será usado para la comparación con PBB. Se puede observar que el dato de entrada es leído desde el sistema de archivos y enviado al filtro de decodificación donde se producen las cinco piezas, una de ellas es almacenada en el sistema de archivos del VC de Meta-Filtro y las demás son enviadas a los VCs de Meta-Filtros de almacenes.

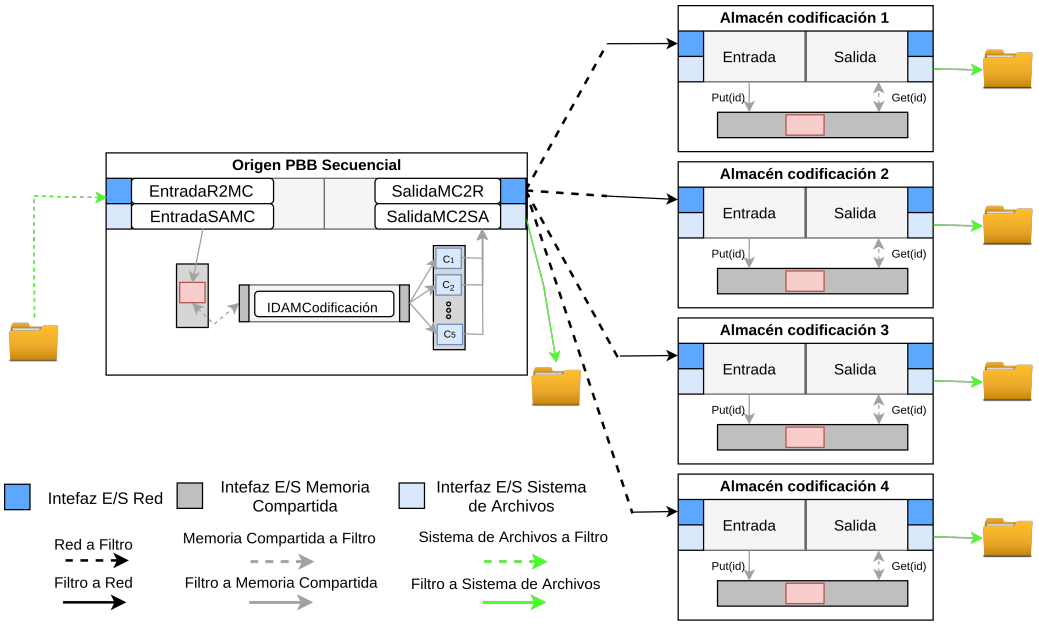


Figura 4.17: Proceso de codificación secuencial para la solución PBB.

En la Figura 4.18 se muestra el proceso de decodificación de esta versión de *secuencial* que será usado para la comparación con PBB. Se puede observar que las piezas para la decodificación son obtenidas desde el sistema de archivos (una pieza) y desde la red (las dos restantes). Estas piezas son usadas por el filtro IDAMDecodificación y la salida producida es almacenada en el sistema de archivos.

La característica de *Secuencial* es el uso de SMS para la comunicación entre filtros de procesamiento dentro de los VCs de Meta-Filtros haciendo que disminuya el tiempo de Lectura/Escritura de datos incrementando la velocidad de procesamiento de la Meta-tubería. Las dependencias de esta solución son el uso de un sistema operativo Linux que permita el manejo de SMS y el compilador GCC para realizar la compilación de las aplicaciones que componen las BBs.

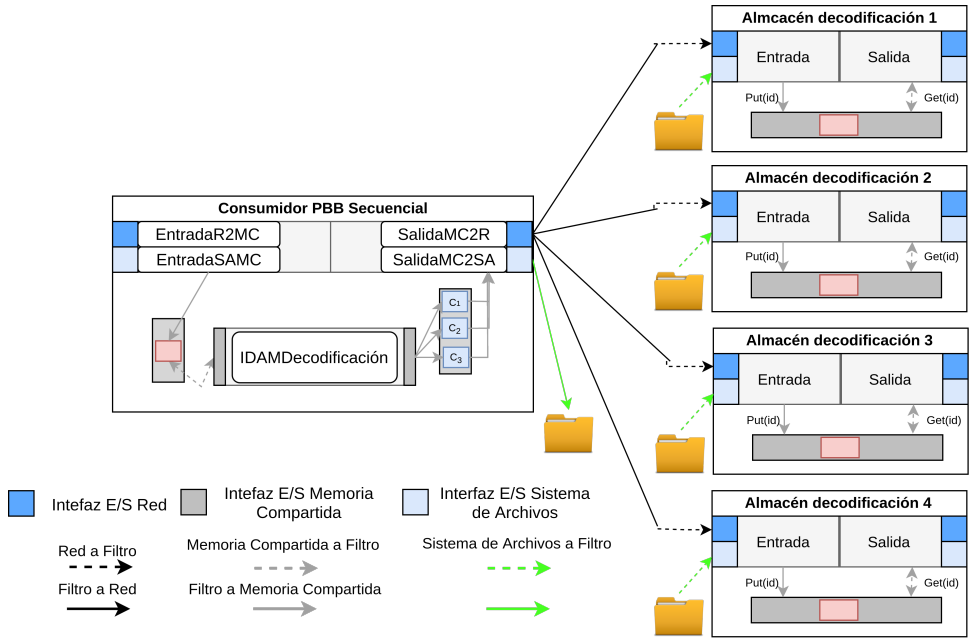


Figura 4.18: Proceso de decodificación secuencial para la solución PBB.

4.4.4 Multipipeline

Consiste en la encapsulación de dos aplicaciones dentro de VCs para realizar la codificación y decodificación de datos. Estas aplicaciones se encuentran paralelizadas aplicando técnicas de paralelización sobre las funciones que las componen utilizando la biblioteca *threading building blocks* de Intel (Intel TBB, por sus siglas en inglés). Es una solución que funciona con sus propias interfaces de comunicación, la interfaz de comunicación utilizada es el sistema de archivos. Las CIs generadas para esta solución son dos.

- *Origen multipipeline*: Realiza la lectura de datos usando el sistema de archivos, codifica los datos leídos y almacena las piezas resultantes en el sistema de archivos tal como se muestra en la Figura 4.19.
- *Consumidor multipipeline*: Realiza la lectura las tres piezas requeridas para la codificación desde el sistema de archivos y almacena los datos decodificados en sistema de archivos tal como se

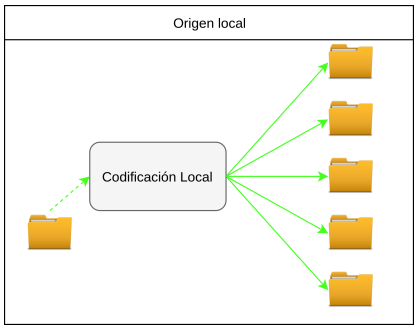


Figura 4.19: Contenedor Multipipeline Origen multipipeline.

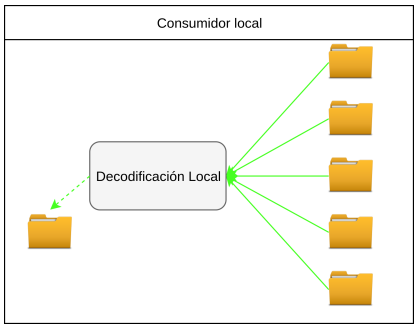


Figura 4.20: Contenedor Multipipeline Consumidor multipipeline.

muestra en la Figura 4.20.

Las características de esta solución son: el uso de técnicas de paralelización dentro de las aplicaciones, tiene dos interfaces de comunicación habilitadas (sistema de archivos y red), la utilización del sistema de archivos como medio de comunicación entre procesos cuando la entrada o salida de los datos es por sistema de archivos, el uso de la memoria como mecanismo de comunicación entre procesos cuando se utiliza la interfaz de red como entrada o salida.

Las dependencias de esta aplicación son: el uso de un sistema operativo Linux, el compilador GPP utilizado para compilar los códigos de las aplicaciones utilizadas así como su ejecución y la biblioteca Intel TBB. También tiene dependencias de *hardware* como el uso de una infraestructura con procesador Intel para la utilización de sus aplicaciones.

5

Evaluación experimental y resultados

En este capítulo se describen las pruebas y escenarios de experimentación utilizados para la evaluación del prototipo de Meta-Tuberías de procesamiento. Cada escenario incluye las configuraciones, variaciones y métricas utilizadas en la experimentación así como la evaluación de los resultados obtenidos.

5.1 Metodología de evaluación

Se definieron una prueba de funcionalidad y dos escenarios de experimentación para comprobar los requerimientos de *variedad* de procesos (utilizar múltiples VC de Meta-Filtros), manejo de grandes *volúmenes* de datos (procesando múltiples archivos con tamaños de hasta un GB) y *velocidad* de procesamiento (procesando los datos usando el esquema *divide y encapsula*).

En la prueba de funcionalidad se desplegaron Meta-Filtro tales como: *Meta-Filtro Envío* y *Meta-Filtro Recepción* los cuales fueron desplegados en *Equipo1* y *Equipo3* (ver características de los

equipos en la tabla 5.1). Se realizó un análisis de integridad, usando el algoritmo de resumen MD5 [54], de los datos al entrar y salir de los filtros de la tubería.

En el primer escenario de experimentación, se realizó un análisis del impacto de la variedad de filtros sobre la utilización del almacenamiento. En este escenario también se analizó el impacto de la utilización de las interfaces de E/S de los filtros en el rendimiento de la tubería así como el uso de memoria compartida entre filtros.

En el segundo escenario se realizó un análisis de rendimiento del esquema de procesamiento *divide y encapsula*. Este escenario se diseñó para comprobar las características de manejo de grandes volúmenes de datos y velocidad de procesamiento del esquema propuesto. El esquema fue comparado con las soluciones *secuencial* y *multipipeline* de forma local. En este escenario, también se analizó el despliegue de una fábrica de procesamiento y se generaron múltiples tuberías desplegadas en un cluster en la nube, los cuales fueron utilizados para mostrar la flexibilidad de despliegue del método propuesto en diferentes infraestructuras así como para evaluar el comportamiento de los tiempos de respuesta obtenidos.

5.1.1 Infraestructura, métricas utilizadas y experimentos realizados

En esta sección se describe la infraestructura utilizada para desplegar las soluciones estudiadas en los escenarios de evaluación previamente definidos. También se describen los experimentos realizados así como las métricas capturadas durante la ejecución de los experimentos.

Infraestructura

La infraestructura utilizada para el despliegue de las soluciones implementadas es mostrada en la Tabla 5.1.

Tabla 5.1: Infraestructura utilizada

Id Infraestructura	PCs	Sistema Operativo	Cores	RAM	HD
Equipo1	1	Elementary OS	8	6	240GB
Equipo2	5	CentOs	6	12 GB	500 GB
Equipo3	1	Ubuntu 14.04	4	4 GB	240GB

Variaciones de parámetros de entrada en experimentos realizados

Los experimentos realizados por los escenarios de experimentación son:

- Tamaño del archivo: Representa el tamaño del archivo por codificar con valores de 1, 10, 100 y 1000 MB.
- Número de segmentos: Indica el número de segmentos s que serán generados (en el proceso de codificación) o recuperados (en el proceso de decodificación) por la propuesta *divide y encapsula*. Este valor también indica el número de tuberías ejecutadas en paralelo que serán utilizadas para el procesamiento de los datos (codificación y decodificación).
- Soluciones implementadas: Es la solución que será utilizada en los escenarios de experimentación, las cuales son: *secuencial, multicontenedor, PBB y multipipeline* (ver descripción en la sección 4.4).

Métricas

Las métricas utilizadas para la evaluación de los tres escenarios se describen a continuación:

- *Resumen MD5*. Es el resultado de aplicar el algoritmo *hash* MD5 [54] al contenido $|F|$ y obtener su resumen que será usado para verificar la integridad de los datos al aplicar el mismo hash a los datos que se quieren validar y comparar los resúmenes. Si los resúmenes son iguales entonces el archivo está íntegro, de lo contrario algún byte fue modificado.

- *Tiempo de respuesta*. El tiempo de respuesta (Tr) se compone de tres elementos correspondientes a una etapa de realización del procesamiento de los datos. Estos elementos son mostrados en la Figura 5.1 y descritos a continuación.

  - El tiempo de recepción/entrada de parámetros ($T_{entrada}$). Es el tiempo requerido por el filtro de procesamiento para la recepción de sus parámetros de entrada.
  - El tiempo de servicio ($T_{servicio}$). Representa el tiempo que consume la ejecución del filtro de procesamiento.
  - El tiempo de entrega de la respuesta al cliente ($T_{entrega}$). Muestra el tiempo que se demoró el filtro de procesamiento en entregar los resultados al usuario.

  El Tr se obtiene al realizar la suma de los tres elementos descritos anteriormente y es representado por la ecuación 5.1.

$$Tr = T_{entrada} + T_{servicio} + T_{entrega} \tag{5.1}$$

Figura 5.1: Tiempos que conforman el Tr.

- *Diferencia entre tiempos de respuesta*. $D_{Tr}(Tr_{solucion1}, Tr_{solucion2})$ es la diferencia del Tr de dos soluciones ($Tr_{solucion1}$ y $Tr_{solucion2}$). Es utilizado para saber el tiempo ahorrado/incrementado por $Tr_{solucion2}$ con respecto a $Tr_{solucion1}$, si es positivo significa que $Tr_{solucion2}$ es más rápido, de los contrario $Tr_{solucion1}$ es más rápido.

$$D_{Tr}(Tr_{solucion1}, Tr_{solucion2}) = Tr_{solucion1} - Tr_{solucion2} \tag{5.2}$$

- *Ganancia en tiempo de respuesta*. $G_{Tr}(Tr_{solucion1}, Tr_{solucion2})$ representa el porcentaje de ganancia obtenida por una solución con respecto a otra ($Tr_{solucion2}$ con respecto a $Tr_{solucion1}$). Si es positiva, la segunda solución es mejor, de lo contrario la primera es mejor.

$$G_{Tr}(Tr_{solucion1}, Tr_{solucion2}) = (D_{Tr}(Tr_{solucion1}, Tr_{solucion2})/Tr_{solucion1}) * 100 \quad (5.3)$$

- *Tiempo de respuesta de un Meta-Filtro*. El Tr de una Meta-Filtro (Tr_{MF}) es el resultado de sumar el tiempo de respuesta de los f filtros que se encuentran en su capa de procesamiento y su capa de entrada. El Tr_{MF} se obtiene utilizando la ecuación 5.4.

$$Tr_{BB} = \sum_{i=1}^{f} Tr_i \quad (5.4)$$

- *Tiempo de respuesta de una Meta-Tubería*. El Tr de una Meta-Tubería (Tr_{MT}) es el resultado de sumar los Tr_{MF} de los F Meta-Filtros que la componen. El Tr_{MT} se obtiene utilizando la ecuación 5.5.

$$Tr_{MT} = \sum_{j=1}^{F} Tr_{MFj} \quad (5.5)$$

- *Tiempo de respuesta de Meta-Tubería generada por los PBBs*. El tiempo de respuesta de una Meta-Tubería generada por las PBBs $TrPBB_{MT}$ corresponde al Tr_{MT} más lenta. Esto ocurre ya que la segmentación implementada genera s salidas en forma paralela y cada una procesa un segmento en una Meta-Tubería distinta, por lo que su Tr equivale al Tr de la última en terminar. El $TrPBB_{MT}$ se obtiene utilizando la ecuación 5.6.

$$TrPBB_{MT} = \max_{j=2}^{s} Tr_{MTj} \quad (5.6)$$

- *Tasa de procesamiento o Throughput*. T, es la cantidad máxima de datos procesados en una

unidad de tiempo. Para estas pruebas se utilizaron MB procesados en el Tr (s).

$$T = MBsProcesados/Tr \tag{5.7}$$

- *Aceleración*. Ac, es la aceleración obtenida por una segunda versión de una aplicación con respecto al Tr obtenido. Es obtenida al dividir el Tr de la versión de la aplicación por el Tr de la segunda versión.

$$Ac = Tr_{version1}/Tr_{version2} \tag{5.8}$$

- *Tamaño del archivo resultante*: Es la cantidad de bytes requeridos para el almacenamiento del contenido procesado por toda la tubería.

En las siguientes secciones se detallan los escenarios, pruebas realizadas y resultados obtenidos durante la experimentación.

5.2 Prueba de funcionalidad: mediante el análisis de integridad en tuberías de procesamiento.

Para comprobar que la implementación los componentes (de la capa de entrada y capa de procesamiento) de la arquitectura diseñada funcionen correctamente, se realizó una prueba de funcionalidad donde se utilizaron los VCs de Meta-Filtros llamados: *Meta-Filtro Envío* y *Meta-Filtro Recepción*.

Para esta etapa de evaluación se realizó el experimento descrito en Figura 5.2. En éste experimento se utilizó el *Meta-Filtro Envío* desplegado en *Equipo1*, el cual lee un archivo desde el sistema de archivos (E1), calcula un MD5 del contenido del mismo (E2) y envía ambos resultados a la capa de entrada (E3). Los datos son enviados al *Meta-Filtro Recepción* (E4) el cual los recibe y coloca

en la memoria compartida (R1) y calcula el MD5 de los datos recibidos (R2), compara el MD5 calculado con el MD5 enviado por el Meta-Filtro anterior y lo envía a la capa de entrada (R3). En este punto los datos resultantes son almacenados en el sistema de archivos (R4). La tubería entrega como resultado la cantidad de MD5 no coincidentes, para validar la integridad de los datos durante el flujo continuo de datos.

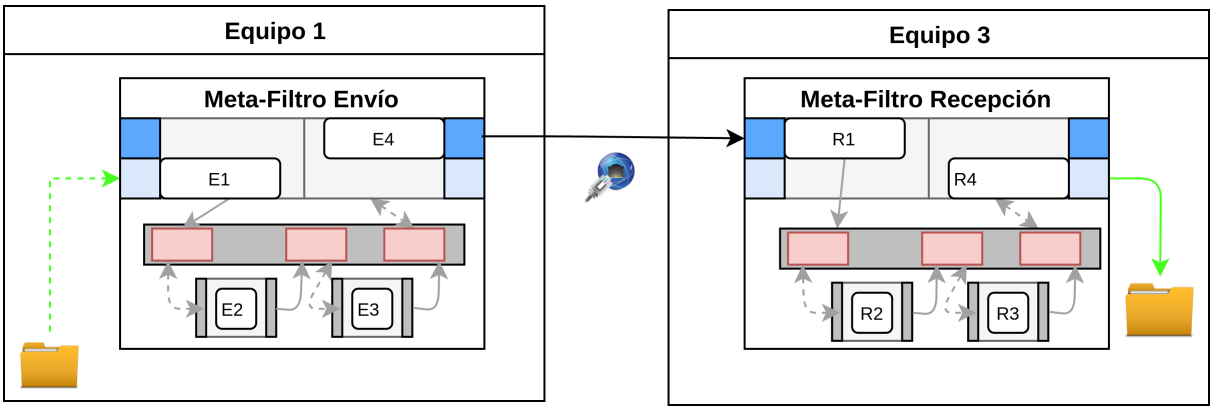


Figura 5.2: Distribución de los VCs de Meta-Filtros para prueba de funcionalidad

El total de archivos enviados entre ambos Meta-Filtros fue de 100 (25 por cada tamaño descrito en la sección 5.1.1.2), de los cuales el 100 % pasó la prueba de integridad. Estos resultados muestran que no se detectó ninguna alteración en la transferencia de datos de la tubería desplegada usando las interfaces de comunicación entre filtros de procesamiento y Meta-filtros validando las implementaciones realizadas.

5.3 Escenario 1: Análisis del impacto de la variedad de filtros y almacenamiento en memoria sobre la solución de procesamiento

En esta sección se describen las pruebas realizadas con la solución de procesamiento IDA(5,3), la cual permite validar que el método propuesto cumple con las características de *variedad* de filtros de procesamiento y mostrar un incremento en la *velocidad* de procesamiento.

5.3.1 Análisis de la variedad de filtros en tuberías

El poder realizar *variedad* de procesos a lo largo de las Meta-Tubería es uno de nuestros objetivos específicos. En este apartado se describen las pruebas realizadas para mostrar uno de los beneficios que pueden obtenerse al realizar la adición de diversos filtros de procesamiento a una tubería. La variación de la secuencia de ejecución de filtros en una tubería impacta en la utilización de almacenamiento de una tubería.

Los filtros de procesamiento utilizados para los experimentos en este escenario son: *IDASA*, *confidencialidad* y *compresión* descritos en la sección 4.3. Con los tres filtros antes mencionados se desplegaron cuatro tuberías de procesamiento: La primera solo incluía el filtro *IDASA*, la segunda solo incluía el filtro *compresión*, la tercera incluía los filtros de *compresión+ IDASA* y la última incluía los filtros de *compresión + confidencialidad + IDASA*. Para el despliegue de las tuberías de procesamiento creadas fue utilizado el *Equipo1* descrito en la Tabla 5.1.

Las cuatro tuberías fueron evaluadas variando el tamaño del archivo de entrada (ver la sección 5.1.1.2).

Las variaciones utilizadas para esta configuración son: la combinación de diferentes filtros de procesamiento utilizados para la generación de las cuatro tuberías evaluadas y el tamaño del archivo de entrada definido en la sección 5.1.1.2. El tamaño del archivo resultante generado por las tuberías de procesamiento utilizadas fue la métrica considerada en estos experimentos.

Discusión de resultados

En la Figura 5.3 se muestra, en el eje de abscisas, el tamaño del archivo original y en el eje de ordenadas, el tamaño del archivo resultante producido por las cuatro tuberías evaluadas.

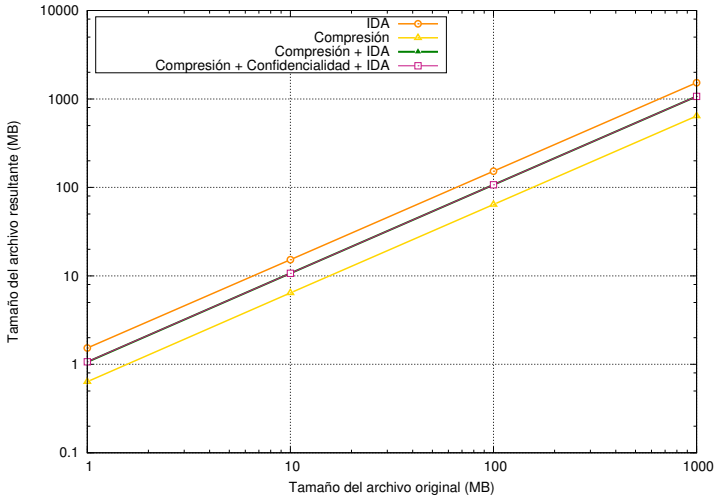


Figura 5.3: Tamaños de los archivos resultantes utilizando diferentes tuberías de procesamiento

Como se puede ver en la Figura 5.3, como era de esperarse el uso de la tubería de *compresión* reduce la cantidad de datos que deben ser almacenados en comparación al resto de tuberías analizadas. La tubería *IDA* incrementa el total de datos por la redundancia agregada por el esquema de tolerancia a fallos, lo cual es deseable en entornos de nube. Un efecto interesante de reducción en la utilización de almacenamiento se observa en la Figura 5.3 en la configuración que combina los filtros de *compresión + IDA* en una tubería, donde el tamaño resultante de esta tubería es menor al tamaño producido por la configuración desplegando la tubería *IDA*. Esto significa que aunque ambas

configuraciones ofrecen el mismo nivel de tolerancia a fallos, la configuración que no combina filtros (IDA) consume más recursos de almacenamiento.

Lo anterior se debe a dos aspectos: i) El filtro de *compresión* reduce el área de procesamiento sobre la cual trabajará el siguiente filtro (IDA), lo cual también reduce la redundancia agregada por dicho filtro. 2) Al reducir los datos producidos por el filtro IDA, también se reducen las operaciones que se deben realizar para almacenar los datos resultantes.

La Figura 5.3 también muestra este efecto cuando se encadenan los filtros de *compresión + codificación + IDA* en una tubería, lo cual permite a esa tubería ofrecer los servicios de reducción de datos, confidencialidad y tolerancia a fallos por casi el mismo costo de almacenamiento que *compresión + IDA* (en la Figura 5.3 las líneas de ambas configuraciones aparecen sobrepuestas).

Esta prueba nos permite constatar la posibilidad de construir tuberías de procesamiento a través de la conexión de filtros (procesos) con tareas distintas intercambiando datos a través de interfaces de E/S estandarizadas (capa de acceso de los filtros). La prueba ofrece un ejemplo de cómo puede construirse un servicio (tubería) eficiente, combinando filtros (variedad) que proporcionan servicios de tolerancia a fallos y confidencialidad, que demandan recursos de almacenamiento, con otros servicios que ayudan a compensar esta demanda (comprensión), ofreciendo un servicio integral y transparente para los usuario y/o aplicaciones clientes que invoquen a la tubería.

5.3.2 Impacto del almacenamiento en memoria sobre la solución de procesamiento

En la sección anterior se mostraron los beneficios de las tuberías de procesamiento en términos de reducción de área de procesamiento y espacio de almacenamiento. Sin embargo, se intuye que el transporte de datos entre la fuente, los filtros y el almacén final impacta en la eficiencia de la tubería

en términos de *velocidad* de procesamiento. Con el fin de hacer frente a este desafió, la arquitectura propuesta ofrece el uso de almacenamiento en memoria como una opción para mejorar el flujo continuo de datos a través de las tuberías de procesamiento. De hecho, el conseguir incrementar la *velocidad* de procesamiento es un objetivo particular del presente trabajo de tesis.

La velocidad de procesamiento de datos, en una tubería como las propuestas en el presente trabajo de tesis, se puede incrementar evitando el uso de las interfaces de red y el sistema de archivos para la comunicación entre los filtros intermedios de una tubería, lo cual resulta en una flujo de datos a través de las interfaces de memoria. Esta configuración crea un esquema de almacenamiento en memoria. Para probar ésta característica de la arquitectura propuesta, se realizaron experimentos con las tuberías donde los filtros trabajan con el sistema de archivos (*IDASA*)y memoria *IDAM*. Los filtros utilizados cuentan con las versiones de codificación y decodificación del algoritmo IDA (ver implementación en la sección 4.3). En la Tabla 5.2 se muestran las interfaces utilizadas por las versiones de los filtros desplegados en las tuberías.

Tabla 5.2: Configuraciones de comunicación para los tuberías evaluadas.

Aplicación	Comunicación por memoria
IDASACodificación	No
IDAMCodificación	Si
IDASACodificación	No
IDAMDecodificación	Si

Para el despliegue de estos filtros de procesamiento se utilizó la infraestructura *Equipo1* descrita en la Tabla 5.1. Las variaciones utilizadas para esta configuración es el tamaño del archivo de entrada (ver sección 5.1.1.2). Las métricas evaluadas son el Tr, T, D_{Tr}, G_{Tr} y Ac descritas en la sección 5.1.1.3.

Discusión de resultados de codificación

Los resultados de las métricas evaluadas para los filtros utilizados en el proceso de codificación se encuentran en la Tabla 5.3. Estos resultados muestran que el uso de la memoria como medio de comunicación entre procesos permite disminuir los tiempos de respuesta al incrementar la cantidad de datos procesados por segundo para archivos de diferentes tamaños.

Tabla 5.3: Resultados codificación

Tamaño(MB)	Filtro	$Tr(s)$	$T(MB/s)$	$D_{Tr}(s)$	$G_{Tr}(\%)$	$Ac(X)$
1	IDASACodificación	0.201	4.97	0.072	35.82	1.55814
	IDAMCodificación	0.129	7.75			
10	IDASACodificación	1.61	6.18	0.5	31.18	1.45045
	IDAMCodificación	1.11	8.99			
100	IDASACodificación	15.812	6.32	4.506	28.49	1.39855
	IDAMCodificación	11.306	8.84			
1000	IDASACodificación	155.81	6.41	38.177	24.50	1.32446
	IDAMCodificación	117.64	8.50			

En la Figura 5.4 se muestra en el eje de abscisas, el tamaño del archivo por codificar y en el eje de ordenadas el tiempo de respuesta de los dos filtros evaluados en el proceso de codificación.

En la Figura 5.4 se observa que el comportamiento del tiempo de respuesta del filtro IDAMCodificación siempre es menor que el obtenido por IDASACodificación. Esto se debe a la reducción de los tiempos de procesamiento, lectura y escritura conseguidos por el uso de la memoria. La solución *IDAMCodificación* consume el 94.47 % (promedio) del Tr mientras que las operaciones de entrada el 3.15 % (promedio) y las operaciones de salida el 2.37 % (promedio). Para el caso del filtro *IDASACodificación* utiliza un 68.78 % (promedio) del *Tr* para el procesamiento, el 9.79 % (promedio) para la entrada de datos y el 22.41 % (promedio) para la salida de los datos.

Este comportamiento confirma que el uso de la memoria como medio de comunicación entre

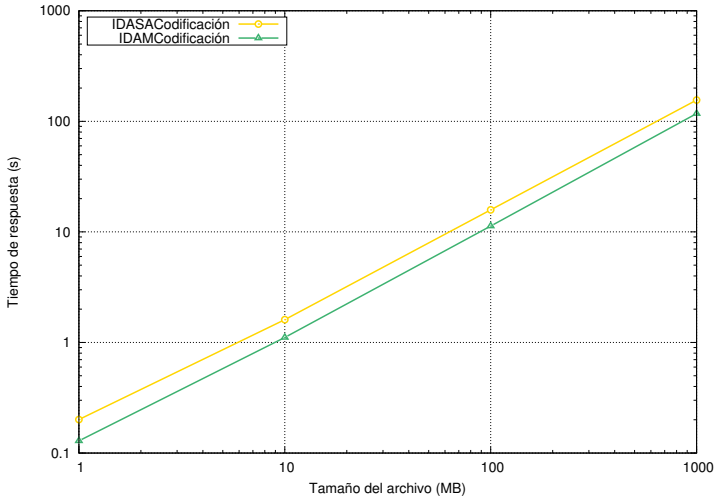


Figura 5.4: Comportamiento del *Tr* de los filtros de procesamiento utilizados

procesos de un filtro disminuye el tiempo de respuesta en el proceso de codificación de datos, debido al incremento en la cantidad de datos procesados por segundo T conseguido por la velocidad E/S de la memoria en comparación con las velocidades del sistema de archivos.

Discusión de resultados de decodificación

Los resultados de las métricas evaluadas para los filtros utilizados en el proceso de decodificación se encuentran en la Tabla 5.4. Estos resultados, al igual que los obtenidos por el proceso de codificación, muestran que el uso de la memoria como medio de comunicación entre procesos permite disminuir los tiempos de respuesta Tr al incrementar la cantidad de datos procesados por segundo T al decodificar diferentes tamaños de archivos.

En la Figura 5.5 se muestra en el eje de abscisas, el tamaño del archivo y en el eje de ordenadas, el tiempo de respuesta Tr obtenido por los filtros de procesamiento evaluados para el proceso de decodificación.

Tabla 5.4: Resultados decodificación

Tamaño(MB)	Filtro	$Tr(s)$	$T(MB/s)$	$D_{Tr}(s)$	$G(\%)$	$Ac(X)$
1	IDASADecodificación	0.116	8.62	0.04	34.482	1.52632
	IDAMDecodificación	0.076	13.15			
10	IDASADecodificación	0.947	10.55	0.383	40.443	1.67908
	IDAMDecodificación	0.564	17.73			
100	IDASADecodificación	9.343	10.70	3.773	40.383	1.67738
	IDAMDecodificación	5.57	17.95			
1000	IDASADecodificación	91.726	10.90	36.282	39.554	1.65439
	IDAMDecodificación	55.444	18.03			

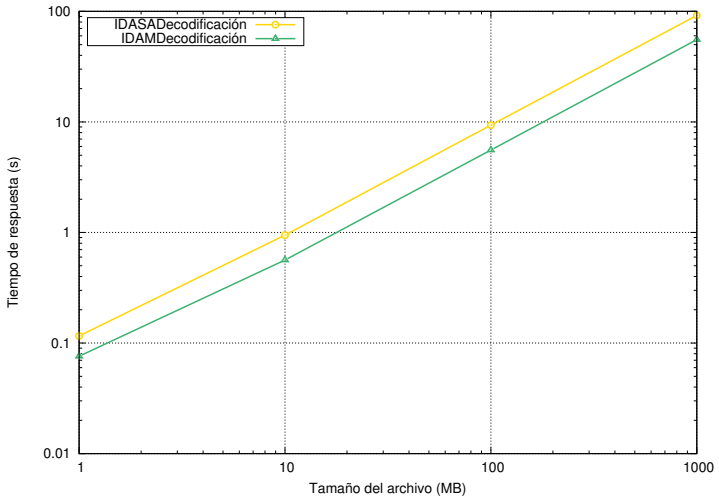


Figura 5.5: Comportamiento del Tr de los filtros de procesamiento utilizados

En la Figura 5.5 se observa que el *Tr* del filtro *IDAMDecodificación* siempre es menor que el *Tr* de *IDASADecodificación* al igual que el comportamiento observado en la prueba de codificación. La solución *IDAMDecodificación* consume el 95.05 % (promedio) del Tr para el procesamiento de los datos, las operaciones de entrada el 1.67 % (promedio) y las operaciones de salida el 3.26 % (promedio). En el caso del filtro *IDASADecodificación* se utiliza un 68.78 % (promedio) del *Tr* para el procesamiento, el 9.79 % (promedio) para la entrada de datos y el 22.41 % (promedio) para la salida de los datos. Estos porcentajes muestran que se obtuvo una reducción en los tiempos requeridos por las operaciones de entrada, procesamiento y salida de datos por el uso de la memoria como interfaz de comunicación.

Los resultados mostrados en las Tablas 5.3 y 5.4 muestran que el uso de la memoria como interfaz de comunicación entre los procesos de las aplicaciones (filtros) evaluadas permite obtener una ganancia promedio del 29.83 % en el proceso de codificación y del 39.96 % en el proceso de decodificación los cuales producen una aceleración promedio de 1.42X al codificar y un 1.66X al decodificar. Esta aceleración es obtenida debido a que la velocidad de acceso a memoria es más rápida que la del sistema de archivos. Por lo tanto, el uso de una interfaz de comunicación con mayor velocidad nos permitirá acelerar las aplicaciones desarrolladas reduciendo los tiempos de comunicación entre proceso de los filtros de una tubería.

El utilizar las interfaces de memoria hace que las operaciones de E/S no sean significativas en comparación al tiempo de procesamiento incrementando la *velocidad* de procesamiento, lo cual es una característica inherente de la solución propuesta en el presente trabajo de tesis.

La posibilidad de cambiar las interfaces de comunicación dependiendo de las circunstancias es una característica inherente de la solución propuesta en el presente trabajo de tesis. Esta característica no solo ofrece flexibilidad en el despliegue de la tubería de procesamiento si no también, como se puede ver en los resultados presentados, permite mejorar la eficiencia de la tubería en términos de tiempos de respuesta.

Un factor crítico para que esta solución consiga las ventajas del almacenamiento en memoria, es el manejo controlado de este recurso. En este sentido, evitar el desbordamiento de memoria, la liberación oportuna de segmentos de memoria una vez utilizados y la comparación de memoria entre filtros son aspectos críticos contemplados en el esquema de almacenamiento en memoria que de no ser atendidos o controlados podrían potencialmente representar una limitación.

5.4 Escenario 2: Análisis de rendimiento del esquema divide y encapsula

En secciones previas se ha mostrado dos aspectos que permiten hacer frente a la problemática establecida en el presente trabajo de tesis. El primero tiene que ver con la variedad de filtros de procesamiento, la cual reduce el área procesamiento y almacenamiento de datos (si se añade un filtro de compresión de datos), mientras que el segundo tiene que ver con los beneficios del almacenamiento en memoria y los flujos continuos de datos, los cuales mejoran la eficiencia de las tuberías en términos de *velocidad* de procesamiento. Sin embargo, para hacer frente a largos *volúmenes* de información se requiere mejorar el despliegue de las tuberías. Para lo cual se propuso el esquema de procesamiento *divide y encapsula*, el cual es evaluado en la presente sección.

Para esta evaluación se desplegaron tuberías de las soluciones: *secuencial*, *multicontenedor* y *multipipeline* descritas en la sección 4.4.3.1 para la codificación y decodificación de datos usando *IDAM* (para *multicontenedor y secuencial*) y *IDASA* en paralelo para (para *multipipeline*). Para el análisis de rendimiento, las soluciones fueron desplegadas en el *Equipo1* descrito en la Tabla 5.1.

Para la solución *multicontenedor*, se establecieron tantas versiones como número de segmentos s utilizados. El valor de s se encuentra entre 2 y 5 ($2 \leq s \leq 5$) debido a que el *Equipo 1* cuenta con 8 cores (4 físicos y 4 virtuales), se decidió usar un core adicional a los físicos para observar el comportamiento de utilizar dos Meta-Tuberías compartiendo un core. Los VCs de Meta-Filtros utilizados para esta configuración en el proceso de codificación son: un PBB segmentador, cinco IDAMCodificación y cinco almacenamiento codificación. Mientras que para la decodificación se utiliza: un PBB integrador, cinco IDAMDecodificación y cinco almacenes decodificación.

Para la evaluación de la solución *multipileline*, se utilizó el VC *origen multipipeline* configurada para acceder todos los cores disponibles por *Equipo1* ($c = 8$) para realizar la codificación. Esta solución almacena las cinco piezas generadas en 5 directorios diferentes del sistema de archivos de VC, se utilizó el VC *consumidor multipipeline* para evaluar el proceso de decodificación, esta solución accede a tres piezas cualesquiera de las cinco almacenadas en el sistema de archivos, realiza el proceso de decodificación y almacena el resultado en el directorio designado.

Las métricas utilizadas para la evaluación de las tres soluciones son: el Tr_{MT}, la $G_{Tr_{MT}}$, el T y la Ac descritas en la sección 5.1.1.3. Las pruebas fueron realizadas variando el tamaño del archivo de entrada (para las tres soluciones) descrito en la sección 5.1.1.2 y el número de segmentos por utilizar (solo en *multicontenedor*).

5.4.1 Características involucradas en las soluciones estudiadas

Antes de mostrar los resultados de codificación y decodificación se presentan en la Tabla 5.5 las soluciones implementadas para la evaluación con las características que incluyen y el efecto que estas tienen en la cantidad de datos que procesan por segundo T.

Tabla 5.5: Características de las soluciones y efecto en la cantidad de datos procesados por segundo (T)

	Soluciones estudiadas				
Característica	Secuencial	Multicontenedor (s)	PBB(s)	Multipipeline	Efecto en T
Paralelismo	No	Si	Si	Si	Aumenta
Segmentación	No	Si	Si	No	Disminuye
Distribución	Si	Si	Si	Si	Disminuye
Almacenamiento en Memoria	Si	Si	Si	No	Aumenta
Manejo	No	Si	Si	No	Disminuye

El efecto de las características mostradas en la Tabla 5.5 son causadas por:

- *Paralelismo*: Permite incrementar la cantidad de datos procesados al realizar el procesamiento de los datos en múltiples hilos en paralelo. La solución *multipipeline* realiza la paralelización a nivel de aplicación donde los procesos fueron paralelizados mediante técnicas de programación, las soluciones *multicontenedor* y *PBB* realizan la división de carga de trabajo en múltiples tuberías en paralelo al implementar el esquema de procesamiento divide y encapsula.

- *Segmentación*: Disminuye el T debido al tiempo de servicio utilizado por el proceso de segmentación.

- *Distribución*: Disminuye el T debido al costo ocasionado por el transporte de datos a través de las interfaces de comunicación (red y sistema de archivos) fuera del VC incrementando el tiempo de Tr y el Tr_{MT}.

- *Almacenamiento en memoria*: Al utilizar la interfaz de memoria se incrementa el T debido a la velocidad de acceso a los datos que ésta ofrece como se mostró en la sección 5.3.2.

- *Gestión*: En las soluciones *multicontenedor* y *PBB* que implementan el esquema de procesamiento *divide y encapsula* se añaden el tiempo de la sincronización de los hilos ejecutados por el segmentador y el tiempo de acceso a los recursos de *hardware* del contenedor a través de la plataforma de VC.

5.4.2 Discusión de resultados de codificación

En la Figura 5.6 se muestran, en el eje de abscisas, los Tr_{MT} producidos por las soluciones estudiadas realizando el proceso de codificación y sus correspondientes versiones para diferentes tamaños de archivos (eje de ordenadas).

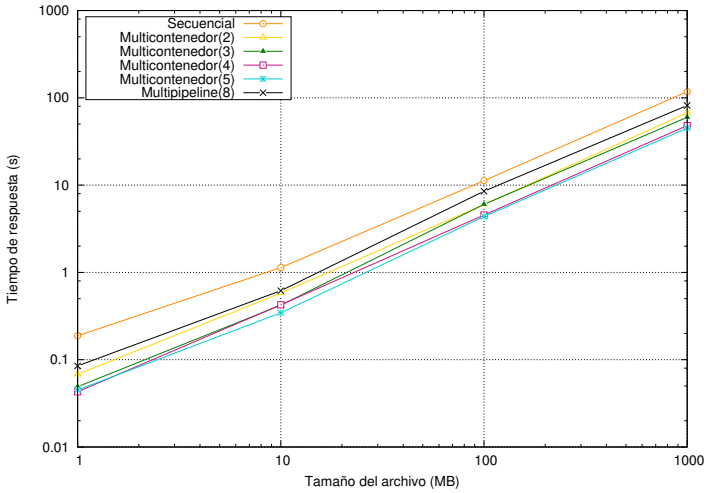


Figura 5.6: Comportamiento del tiempo de respuesta de las soluciones *secuencial*, *multipipeline* y *multicontenedor* en todas sus versiones para el proceso de codificación

En la Figura 5.6 se observa que al incrementar el valor de s que indica el número de Meta-tuberías que utiliza la solución *multicontenedor*, se reduce el Tr_{MT} con respecto al Tr obtenido por las soluciones *secuencial* y *multipipeline* para archivos de 1 MB hasta archivos de 1000 MB. Por ejemplo, en el caso de utilizar la solución *Multicontenedor(5)* se obtiene una aceleración 2.5X con respecto a *secuencial* y 1.6X con respecto a *multipipeline(8)*. La reducción del Tr_{MT} se debe a las características utilizadas por *multicontenedor* descritas en la sección 5.4.1, donde *multicontenedor* se beneficia del almacenamiento en memoria y el paralelismo ofrecido por el esquema de procesamiento *divide y encapsula* las cuales incrementan el T. La solución *secuencial* utiliza el almacenamiento en memoria que incrementa el T y *multipipeline* solo ofrece paralelismo que incrementa el T pero utiliza el sistema de archivos como medio de comunicación produciendo una reducción del T.

En la Figura 5.7 se muestran, en el eje de abscisas, el porcentaje de ganancia obtenido en el Tr_{MT} por *multicontenedor* en todas sus versiones y *multipipeline* con respecto al Tr de la solución *secuencial* para diferentes tamaños de archivos (eje de ordenadas).

En la Figura 5.7 se observa que el porcentaje de ganancia obtenido por la solución *multicontenedor*

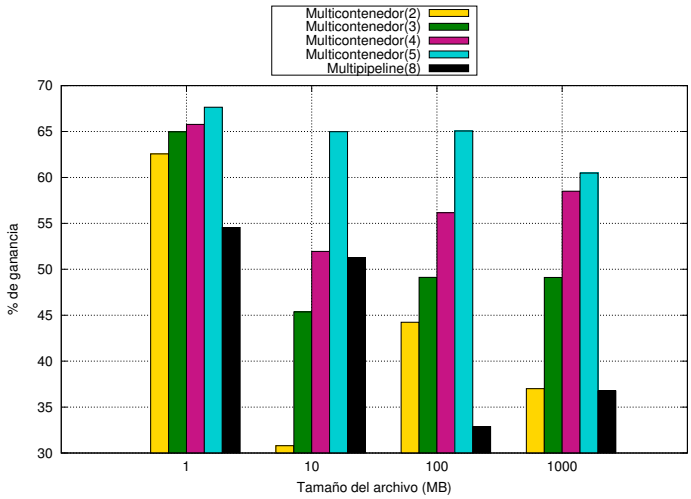


Figura 5.7: Porcentaje de ganancia obtenida por la soluciones *multicontenedor* y *multipipeline* con respecto a *secuencial* en el proceso de codificación

con respecto a la solución *secuencial* incrementa su valor mientras mayor sea el número s de Meta-tuberías utilizadas durante el proceso de codificación. Por otra parte, la ganancia obtenida por *multipipeline* tiende a disminuir mientras mayor sea el tamaño del archivo, excepto en el archivo de 1000 MB donde la solución *secuencial* consume un Tr mayor (117.682 s [mediana]), lo que permite al paralelismo reducir el Tr ya que reduce el tiempo de procesamiento en los hilos incrementando la ganancia obtenida.

El comportamiento de la ganancia observado en las soluciones *multicontenedor* para archivos de 1 y 1000 MB, reflejan que al incrementar el número de Meta-Tuberías de procesamiento s, se incrementan los costos del manejo de los hilos y VCs evitando que la ganancia incremente de forma constante, en los archivos de 10 y 100 MB no es tan notorio el costo de gestión. La diferencia del comportamiento ocurre debido a que la reducción del tiempo de procesamiento es mayor que el costo de la gestión de acceso a los recursos. Este comportamiento se observa en los archivos de 1000 MB, donde la diferencia de ganancia obtenida por *multicontenedor(4)* y *multicontenedor(5)* es del 2 % mientras que para *multicontenedor(3)* y *multicontenedor(4)* la diferencia es del 9 %.

Las ganancias obtenidas por *multipipeline* son menores para casi todas las soluciones de *multicontenedor*, esto se debe a la cantidad de llamadas al sistema de archivos que es más lento que utilizar la memoria (ver sección 5.3.2) para la lectura y escritura del contenido que procesa (la escritura de 5 bytes por cada byte procesado), lo que incrementa su Tr reduciendo la ganancia obtenida.

5.4.3 Discusión de resultados de decodificación

En las pruebas de decodificación, se realizó el proceso la decodificación a los archivos procesados por la prueba anterior, considerando el tamaño y nivel de segmentación utilizados en dichas pruebas. En la Figura 5.8 se muestran, en el eje abscisas, los Tr_{MT} producidos por las soluciones estudiadas y sus correspondientes versiones realizando el proceso de decodificación para diferentes tamaños de archivos (eje de ordenadas).

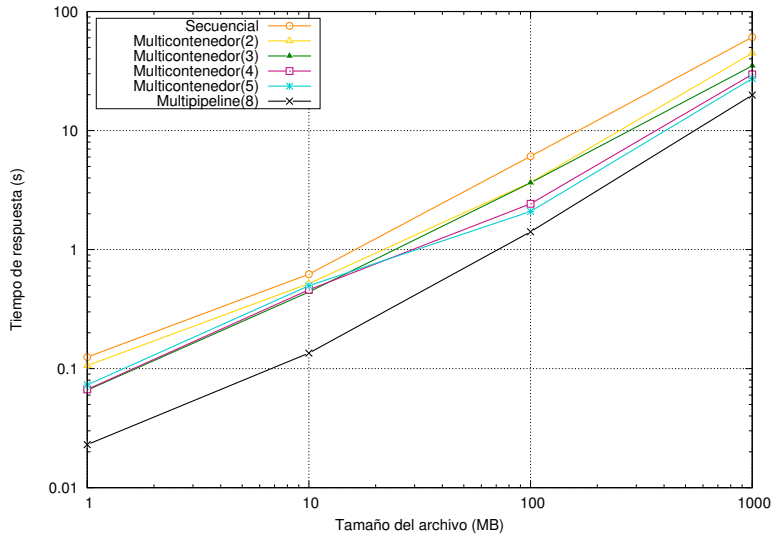


Figura 5.8: Comportamiento del tiempo de respuesta de las soluciones *secuencial*, *multipipeline* y *multicontenedor* en todas sus versiones para el proceso de decodificación

En la Figura 5.8 se observa que al incrementar el valor de s que indica el número de Meta-tuberías

que utiliza la solución *multicontenedor* mediante el esquema de procesamiento *divide y encapsula*, se reduce el Tr_{MF} obtenido con respecto al Tr obtenido por la solución *secuencial* en todas sus versiones para la decodificación de archivos de 1 MB hasta archivos de 1000 MB. Esta diferencia es obtenida gracias al procesamiento en paralelo del archivo de entrada en múltiples Meta-Tuberías ya que ambas poseen el almacenamiento en memoria. Por ejemplo, en el caso de *multicontenedor(5)* se obtiene una aceleración 3.7X con respecto a la solución *secuencial* procesando archivos de 1000 MB.

En el caso de *multipipeline*, esta solución obtiene mejores resultados que la solución *multicontenedor*, este comportamiento se atribuye a los costos de manejo descritos en la sección 5.4.1 y a que realiza 40 % menos accesos al sistema de archivos (1 escritura por cada tres lecturas) que el proceso de codificación, aprovechando su característica de procesamiento paralelo en las operaciones de la aplicación (ver la sección 5.4.1). Sin embargo, al incrementar el valor de s a 5, *multicontenedor(5)* puede reducir la diferencia en el tiempo de procesamiento a dos segundos con respecto al Tr de *multipipeline* en archivos de 1000 MB utilizando una versión de la aplicación que no contiene paralelización en su código fuente si no que es ejecutada s veces para procesar segmento del contenido original.

En la Figura 5.9 se muestran, en el eje de abscisas, el porcentaje de ganancia obtenido por *multicontenedor* en todas sus versiones y *multipipeline* con respecto a *secuencial* para diferentes tamaños de archivos (eje de ordenadas).

En la Figura 5.9 se observa que el porcentaje de ganancia obtenida por *multicontenedor* con respecto a la versión *secuencial* tiende a incrementar mientras mayor sea el número de Meta-tuberías s utilizadas. Debido a que el proceso de decodificación de IDA requiere menos peticiones de lectura/escritura al sistema de archivos incrementa su porcentaje de ganancia con respecto a la

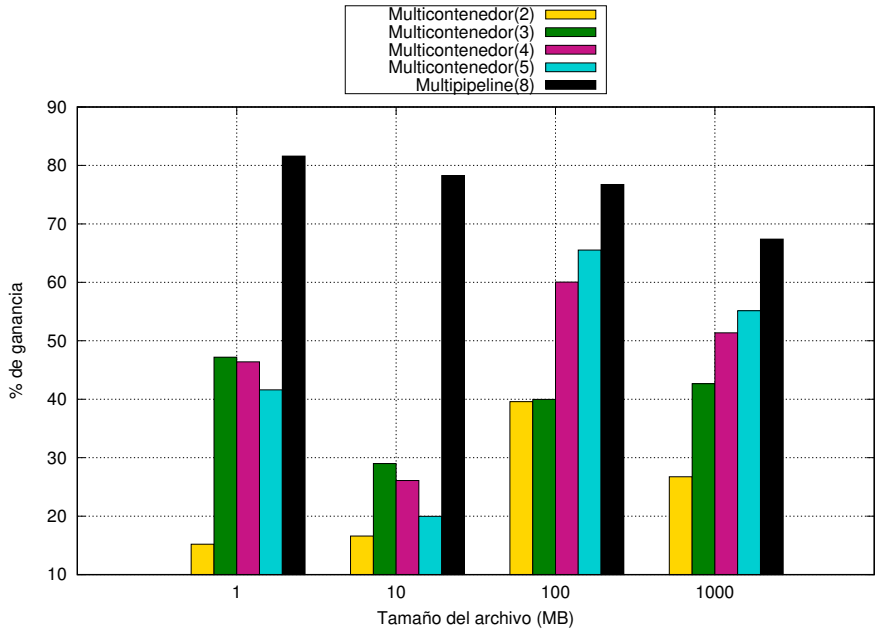


Figura 5.9: Porcentaje de ganancia obtenida por la soluciones *multicontenedor* y *multipipeline* con respecto a *secuencial* en el proceso de decodificación

solución *multicontenedor* propuesta para los tamaños de archivos evaluados.

El comportamiento de la ganancia observado en las soluciones *multicontenedor* para el proceso de decodificación es diferente que los obtenidos en el proceso de codificación. Para archivos de 1 y 10 MB la ganancia obtenida incrementa a un máximo ($s = 3$) y después es reducida, mientras que para archivos de 100 y 1000 MB se incrementa la ganancia hasta $s = 5$. Esto ocurre debido a los costos de manejo producidos por el manejo de hilos y VCs es mayor que el costo del proceso de decodificación para los archivos de 1 y 10 MB mientras que a los archivos de 100 y 1000 MB no les afecta de la misma medida.

Si bien es cierto que *multipipeline* obtiene mejores resultados para la decodificación de archivos, se logró demostrar que el esquema *divide y encapsula* usado en la solución multipipeline permite conseguir un incremento en la *velocidad* de procesamiento para grandes *volúmenes* de datos

(al procesar múltiples archivos de 1000 MB) usando Meta-Tuberías con *variedad* de filtros de procesamiento (los Meta-Filtros utilizados). Estas características cumplen los objetivos de este trabajo de tesis. Además, *multicontenedor* permite obtener resultados competentes con aplicaciones paralelizadas de los filtros utilizados sin alterar el contenido del filtro, mediante la replicación de contenedores y segmentación de procesos en Meta-Tuberías.

Una vez cumplidos los objetivos de: *variedad, volumen y velocidad*. Se realizó el despliegue de dos casos de uso, que permiten realizar la distribución de Meta-Tuberías en entornos de red. Además, estos casos de uso permiten probar la flexibilidad de la arquitectura diseñada combinando o separando filtros de procesamiento dentro de una CI de Meta-Filtro.

5.4.4 Casos de despliegue del esquema divide y encapsula

En las secciones anteriores, se mostraron los beneficios de: usar la *variedad* de filtros de procesamiento para reducir el espacio requerido para el almacenaje de datos, el uso de la memoria como interfaz de comunicación entre filtros de procesamiento para obtener una mayor *velocidad* de procesamiento y acceso a dato así como el uso del esquema *divide y encapsula* para incrementar la *velocidad* de procesamiento de grandes *volúmenes* de datos.

En esta sección se propone realizar el despliegue del esquema *divide y encapsula* para dos casos de uso: *fábrica de procesamiento en cluster* y *tuberías de procesamiento en cluster*. En estos casos de uso se observó el comportamiento del uso de múltiples Meta-Tuberías distribuidas en la red.

El caso de uso llamado, *fábrica de procesamiento en cluster*, corresponde al despliegue de los VC de Meta-Filtros de la solución *multicontenedor* para los procesos de codificación y decodificación de datos descrito en la sección 4.4. Se realizó el despliegue de los Meta-Filtros en un cluster en

la nube, distribuyendo los Meta-Filtros en tres etapas distintas: origen, fábrica de procesamiento y almacenamiento.

El caso de uso llamado, *tuberías de procesamiento en cluster*, corresponde al despliegue de los contenedores de la solución *PBB* para el proceso de codificación y decodificación mostrado en la sección 4.4 en un cluster en la nube, dividiéndolos en dos etapas: procesamiento en el origen y almacenamiento.

Estos dos casos de uso permitieron obtener los resultados sobre el comportamiento del método propuesto, en un entorno de red con diferentes configuraciones, validando la flexibilidad del despliegue de los Meta-Filtros, la *velocidad* de procesamiento proporcionada por el esquema de procesamiento *divide y encapsula*. Además, estos escenarios permiten validar el uso de *variedad* de filtros de procesamiento para procesar grandes *volúmenes* de datos.

Fábrica de procesamiento en cluster

Para este caso de uso, se utilizaron las soluciones *secuencial* (separando el origen y destino de datos) y *multicontenedor* (desplegando los Meta-Filtros en nodos diferentes) de la cual se establecieron tantas versiones como número de segmentos s requeridos. El valor de s se encuentra entre 2 y 5 ($2 \leq s \leq 5$) para usar el mismo número de cores que la evaluación del esquema de procesamiento *divide y encapsula* realizada en la sección 5.4.4.

Las CIs de Meta-Filtros utilizadas para este caso de uso en el proceso de codificación son: PBB segmentador, IDAMCodificación y almacén codificación. Estas CIs de Meta-Filtros fueron utilizadas para el despliegue de los siguientes VCs de Meta-Filtros: un PBB segmentador, cinco IDAMCodificación y cinco almacenes de codificación.

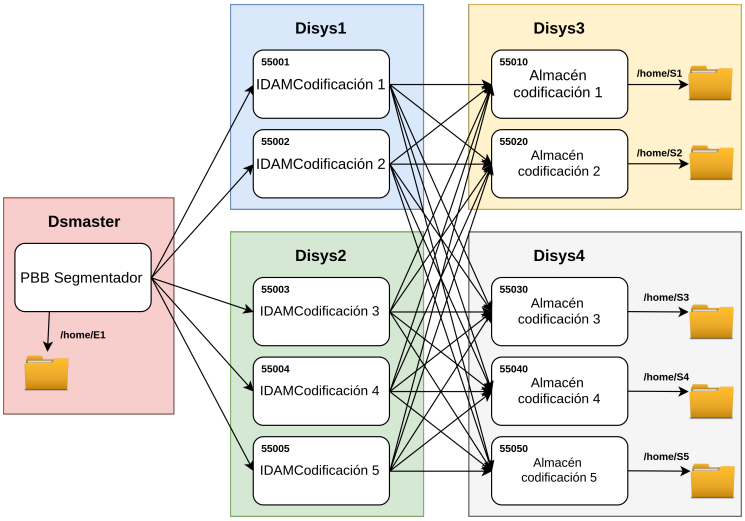


Figura 5.10: Distribución de los contenedores para el proceso de codificación en *Equipo2* formando una *fábrica de procesamiento*.

Para el proceso de decodificación se utilizaron las siguientes CIs de Meta Filtros: PBB integrador, IDAMDecodificación y almacén decodificación. Estas CIs de Meta-Filtros fueron utilizadas para el despliegue de los siguientes VCs de Meta-Filtros: un PBB integrador, cinco IDAMDecodificación y cinco almacenes de decodificación.

En este caso de uso se realizó la distribución de los VCs de Meta-Filtros en un cluster (*Equipo2* descrito en la Tabla 5.1) compuesto de cinco equipos (etiquetados como: Dsmaster, Disys1, Disys2, Disys3, Disys4). La distribución de los VCs de Meta-Filtros y sus parámetros de configuración para el proceso de codificación son mostrados en la Figura 5.10. En la Figura 5.11 se muestran la distribución de los VC de Meta-Filtros y sus configuraciones para el proceso de decodificación.

Discusión de resultados de codificación. En la Figura 5.12 se muestran, en el eje de abscisas, los Tr producidos por las soluciones estudiadas realizando el proceso de codificación y sus correspondientes versiones para diferentes tamaños de archivos (eje de ordenadas).

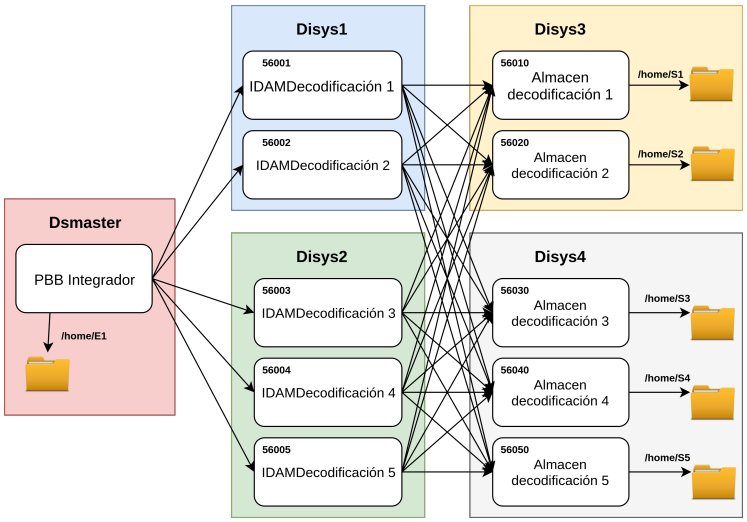


Figura 5.11: Distribución de los contenedores para el proceso de decodificación en *Equipo2* formando una *fábrica de procesamiento*.

En la Figura 5.12 se puede observar que el Tr obtenido por las soluciones de *multicontenedor* es menor que el obtenido por la versión *secuencial* en todos los casos excepto para archivos de 1 MB con *multicontenedor(2)* Meta-Tuberías. Esto ocurre por la adición del costo de distribución de los segmentos y piezas producidas por el proceso de codificación a través de la red en vez de procesarlos en memoria y almacenar los datos en el sistema de archivos. Este costo incrementa el Tr_{MT} de la solución *multicontenedor*, resultando más costoso que la versión *secuencial* en el caso antes mencionado. Este resultado concuerda con las expectativas iniciales, ya que se tenía pensado que la segmentación y transmisión de archivos pequeños (segmentos y piezas codificadas) entre Meta-Filtros a través de la red podría resultar igual o más costoso que la realización del procesamiento de los datos en un solo equipo.

En la Figura 5.13 se muestran, en el eje de abscisas, el porcentaje de ganancia obtenido por la solución *multicontenedor* en todas sus versiones con respecto a la solución *secuencial* para diferentes tamaños de archivos (eje de ordenadas).

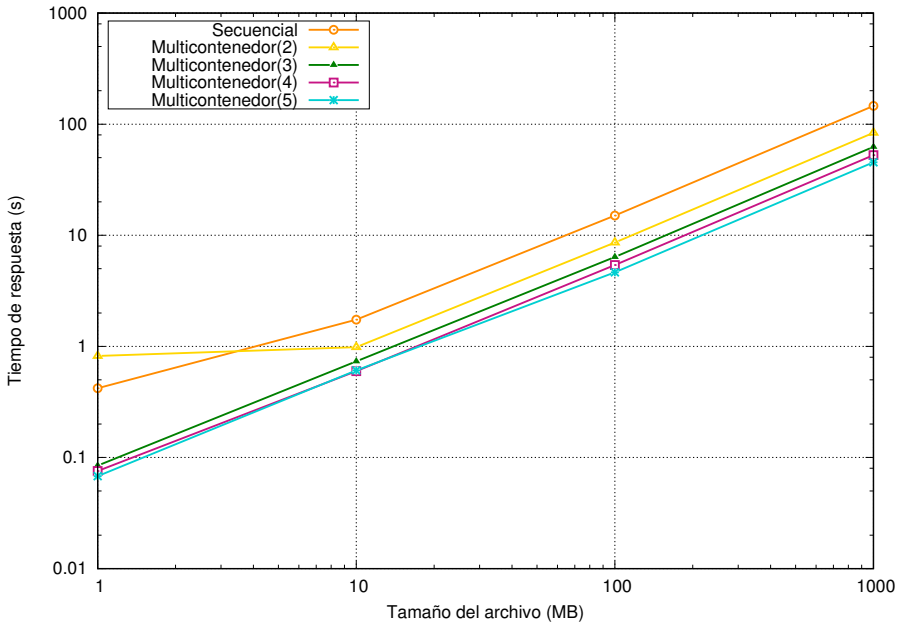


Figura 5.12: Comportamiento del tiempo de respuesta para las soluciones *secuencial* y *multicontenedor* en todas sus versiones para el proceso de codificación usando la *fábrica de procesamiento*.

En la Figura 5.13 se observa que el porcentaje de ganancia obtenida por *multicontenedor* tiende a incrementar mientras mayor sea el número de Meta-tuberías utilizadas s y decrementa al procesar archivos de mayor tamaño. Este comportamiento se asemeja a los resultados obtenidos en la evaluación del rendimiento (en archivos de 1 y 1000 MB) donde los costos de gestión que ahora incluyen el transporte. Estos costos evitan obtener una ganancia lineal, además, en la prueba con la solución *multicontenedor(2)* codificando archivos de 1 MB, el transporte de los dos segmentos y 10 piezas codificadas entre los Meta-Filtros resultó más costosa (en Tr) que la solución *secuencial* produciendo una ganancia negativa.

Discusión de resultados de decodificación Por otro lado en la prueba de decodificación de este caso de uso, se realiza la decodificación de los archivos procesados por la prueba anterior considerando el tamaño y nivel de segmentación utilizados en dichas pruebas. En la Figura 5.14 se muestran, en el

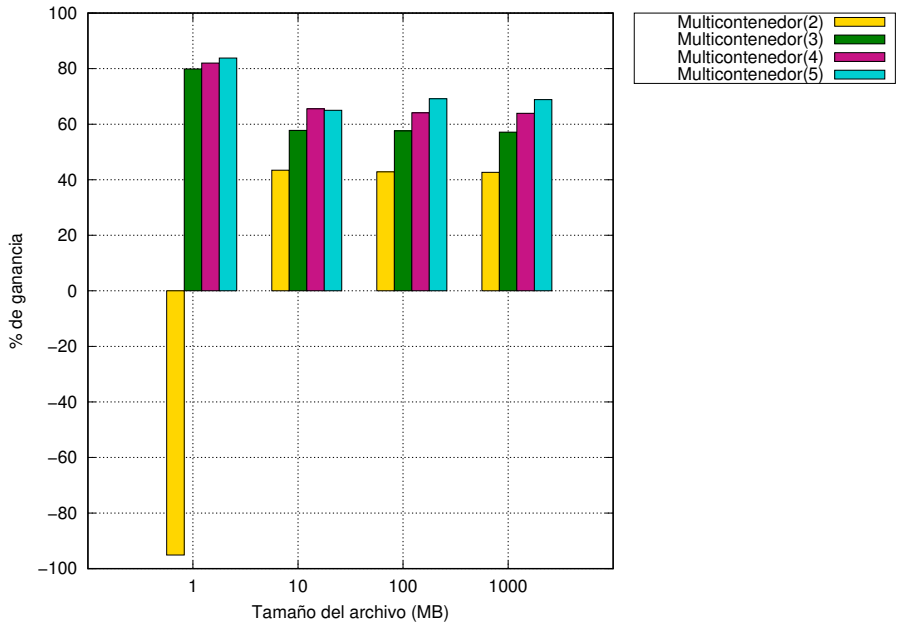


Figura 5.13: Porcentaje de ganancia obtenida por la soluciones *Multicontenedor* con respecto a *Secuencial* en el proceso de codificación usando la *fábrica de procesamiento*.

eje de abscisas, los Tr_{MF} producidos por las soluciones estudiadas y sus correspondientes versiones realizando el proceso de decodificación para diferentes tamaños de archivos (eje de ordenadas).

En la Figura 5.14 se puede observar que el Tr_{MF} obtenido por las soluciones generadas por *multicontenedor* son menores que el Tr obtenido por la versión *secuencial* en todos los casos. Esto ocurre debido a que la cantidad de piezas requeridas para la reconstrucción de cada segmento es menor (se obtienen 3 de las 5 piezas codificadas para cada segmento) y se genera menos tráfico en la red que al realizar la codificación (se envían las 5 piezas generados por segmento) lo que evita que la solución *multicontenedor* obtenga un desempeño menor (en archivos de 1 MB como en la codificación) que la solución *secuencial*.

En la Figura 5.15 se muestran, en el eje de abscisas, el porcentaje de ganancia obtenido por la solución *multicontenedor* en todas sus versiones con respecto a la solución *secuencial* para diferentes

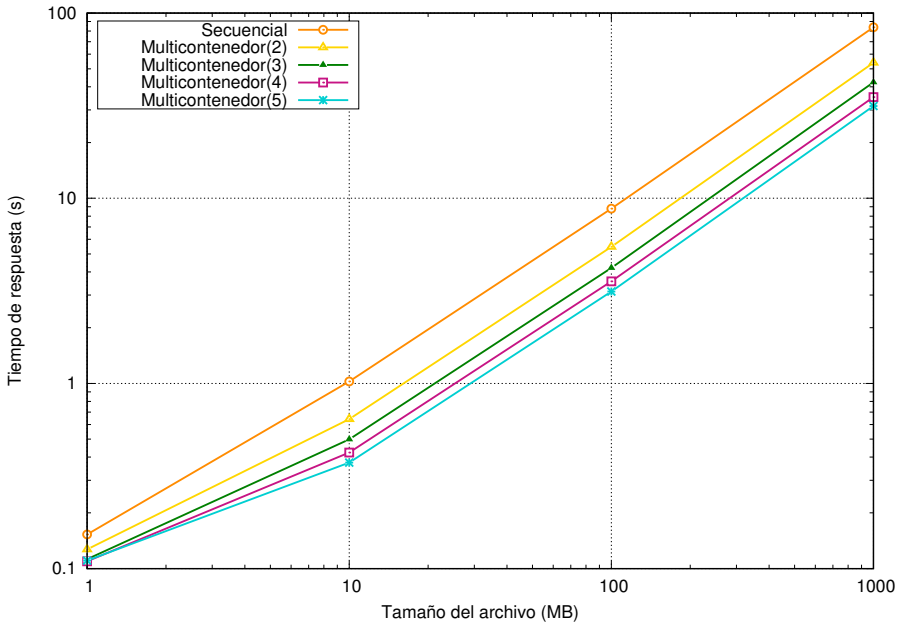


Figura 5.14: Comportamiento del tiempo de respuesta para las soluciones *secuencial* y *multicontenedor* en todas sus versiones para el proceso de codificación usando la *fábrica de procesamiento*.

tamaños de archivos (eje de ordenadas).

En la Figura 5.15 se puede observar que el porcentaje de ganancia obtenida por *multicontenedor* incrementa mientras mayor sea el número de Meta-tuberías utilizadas s y decrementa al procesar archivos de mayor tamaño. Las ganancias obtenidas por la solución *multicontenedor* como *fábrica de procesamiento* en el proceso de decodificación, muestran que las ganancias obtenidas para archivos de 1 MB se incrementan hasta un tope y después disminuyen, como se mencionó en las evaluaciones anteriores esto ocurre por los costos de gestión de los recursos y transporte. Para los demás tamaños de archivos las ganancias siguen incrementando hasta s Meta-Tuberías, pero cada vez se reduce esta ganancia por lo que si se sigue incrementando el valor de s, las ganancias se comportarán igual que los archivos de 1 MB.

En esta sección se mostraron los efectos de realizar la distribución del procesamiento y

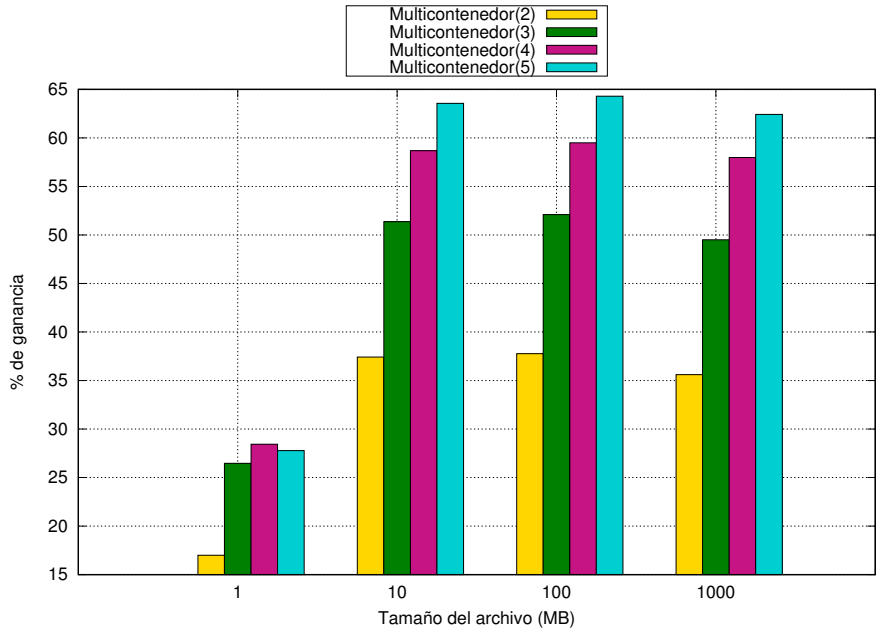


Figura 5.15: Porcentaje de ganancia obtenida por *multicontenedor* con respecto a *secuencial* en la decodificación para el caso de uso *fábrica de procesamiento*

almacenamiento en una fábrica de procesamiento. La fábrica de procesamiento se obtuvo al desplegar múltiples VCs de Meta-Filtros basados en una CI de Meta-Filtro (para cada etapa) con diferentes archivos de configuración para obtener múltiples VCs de Meta-filtros usados para la codificación y decodificación de datos. Los Meta-Filtros de cada etapa realizan el mismo proceso, pero su trayectoria para la dispersión de los datos es diferente.

Este caso de uso puede ser utilizado cuando se tienen varios nodos (equipos de cómputo) que pueden usarse para contener Meta-Filtros de procesamiento accedidos por una capa de acceso (segmentador) que les indica cuándo iniciar a trabajar y qué datos deben procesar y almacenar para reducir los Tr_{MT}.

Tuberías en Cluster

En esta sección se describe el caso de uso *tuberías en cluster* en la nube, donde se propone realizar la consolidación de las tuberías de procesamiento en el origen y realizar la distribución de los datos procesados a los demás nodos del cluster.

En este caso de uso se realiza la distribución de los VCs de Meta-Filtros en el *Equipo2* descrito en la Tabla 5.1 de tal forma que el Meta-Filtro PBB segmentador IDAM se localiza en el origen (dsMaster) y el almacenamiento es realizado por los demás nodos (Disys1-4). Este caso permite evaluar la integración de múltiples filtros (PBB Segmentador y IDAM) en un solo Meta-Filtro. Este modo de uso representa el escenario, donde se cuenta con una infraestructura con grandes características de procesamiento y almacenamiento permitiendo la división de las responsabilidades acorde a estas características.

Las soluciones por evaluar son *secuencial para caja negra paralela* y *PBB segmentador IDAM* descritas en la sección 4.4 de la cual se establecieron tantas versiones como número de segmentos s utilizados. El valor de s se encuentra entre 2 y 6 debido al número de cores del equipo por utilizar $(2 \leq s \leq 6)$.

Las CIs de Meta-Filtros utilizados para el proceso de codificación son: PBB Segmentador IDAM y almacén codificación. A partir de estas CIs se generaron los siguientes Meta-Filtros: un PBB Segmentador IDAM codificación, cuatro almacenes de codificación. Para los almacenes se utilizó la misma CIs de Meta-Filtro solo se utilizó un archivo de configuración diferente para cada uno. El despliegue de estos VCs de Meta-Filtro y los datos usados para su creación es mostrado en la Figura 5.16 donde se puede observar que el proceso de codificación se realiza dentro de el nodo *dsMaster* que almacena una salida en el sistema de archivos local y las demás son transferidas a través de la

red a los almacenes.

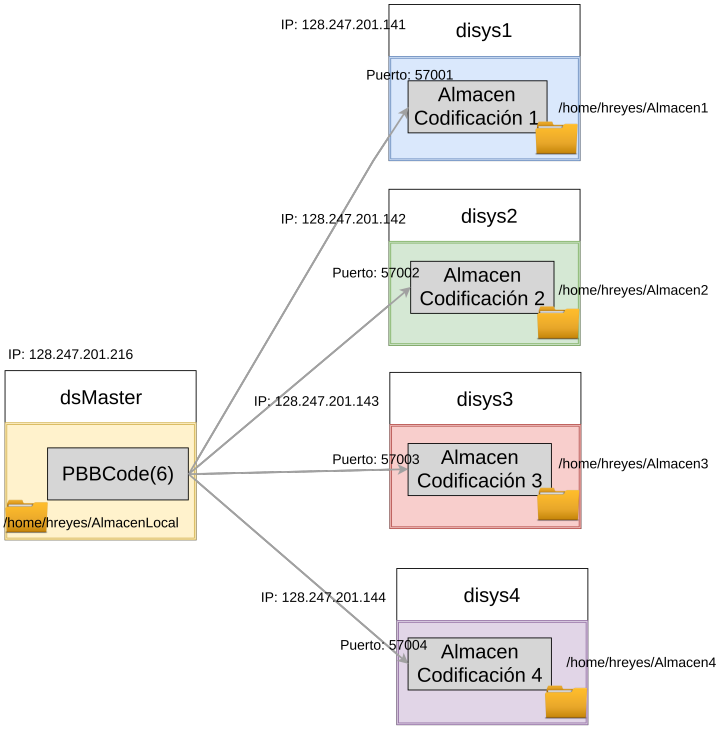


Figura 5.16: Distribución de los contenedores para el proceso de codificación en *Equipo2* formando *tuberías de procesamiento*.

Las CIs de Meta-Filtros utilizados para el proceso de decodificación son: PBB Integrador con IDAM decodificación y almacén decodificación. A partir de estas CIs se generaron los siguientes VC de Meta-Filtros: 1 PBB Integrador con IDAM decodificación, 4 almacenes de decodificación. Para los almacenes se utilizó la misma CIs de Meta-Filtro solo se utilizó un archivo de configuración diferente para cada uno. El despliegue de estos VCs y los datos usados para su creación son mostrados en la Figura 5.17 donde se puede observar que el proceso de decodificación se realiza dentro de el nodo *dsMaster*, que obtiene una pieza desde el sistema de archivos y las dos restantes de los almacenes distribuidos en la red.

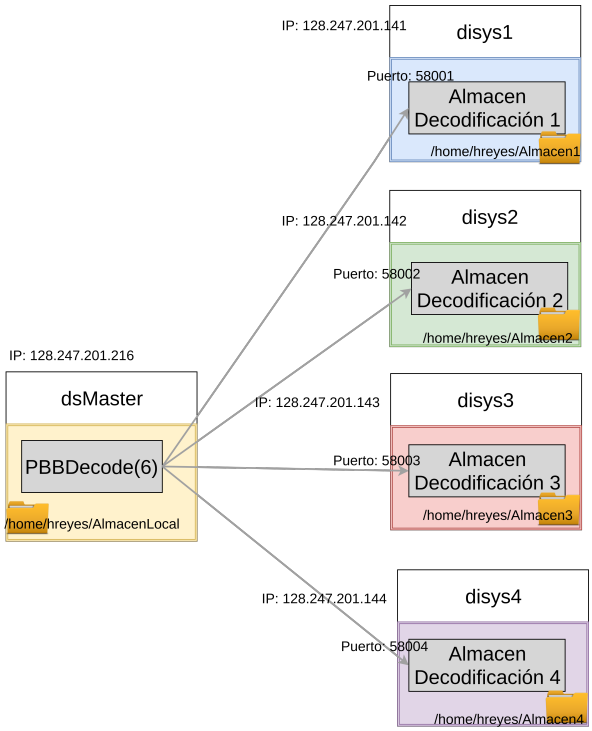


Figura 5.17: Distribución de los contenedores para el proceso de decodificación en *Equipo2* formando *tuberías de procesamiento*.

Para la ejecución de las pruebas se varió el tamaño del archivo utilizado (ver sección 5.1.1.2) y el número de segmentos utilizados. Las métricas para evaluar los resultados obtenidos son: el Tr, la G_{Tr}, la Ac descritas en la sección 5.1.1.3.

Discusión de resultados de codificación En la Figura 5.18 se muestran, en el eje de abscisas, los Tr producidos por las soluciones estudiadas realizando el proceso de codificación y sus correspondientes versiones para diferentes tamaños de archivos (eje ordenadas).

Como se puede observar en la Figura 5.18 al incrementar el número segmentos utilizados por PBB se logra disminuir el Tr al procesar los archivos de distintos tamaños debido a la utilización del esquema de procesamiento *divide y encapsula* aún y cuando el 80 % de los resultados producidos

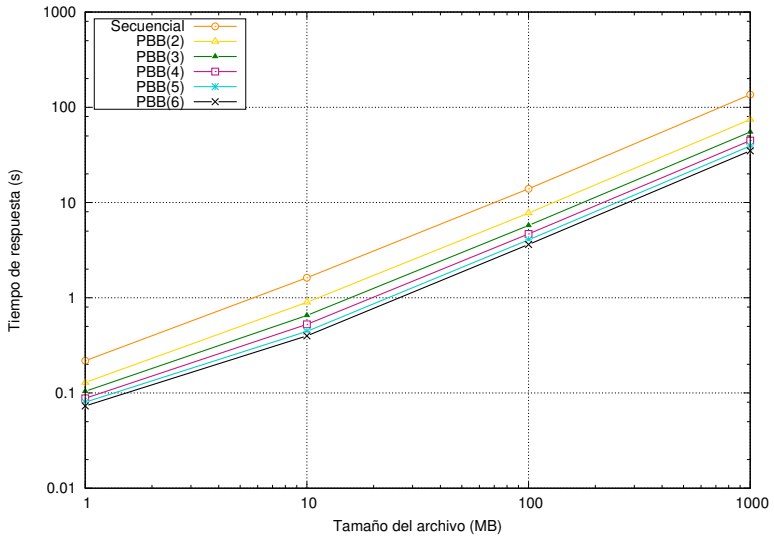


Figura 5.18: Comportamiento del tiempo de respuesta para las soluciones *secuencial* y *PBB* en todas sus versiones para el proceso de codificación usando *tuberías de procesamiento*.

son distribuidos a través de la red y el 20 % al sistema de archivos. Esta prueba muestra que la consolidación de procesos dentro de un Meta-Filtro y distribuir los resultados por las interfaces de la capa de acceso no afecta negativamente al comportamiento de la solución PBB.

En la Figura 5.19 se muestran, en el eje de abscisas, el porcentaje de ganancia obtenido por *PBB* en todas sus versiones con respecto a *secuencial* para diferentes tamaños de archivos (eje de ordenadas).

Como se puede observar en la Figura 5.19 el porcentaje de ganancia incrementa al utilizar más tuberías de procesamiento s al igual que las pruebas realizadas en las secciones anteriores. Al igual que las otras soluciones evaluadas la ganancia se va reduciendo por los costos de gestión y transporte de datos.

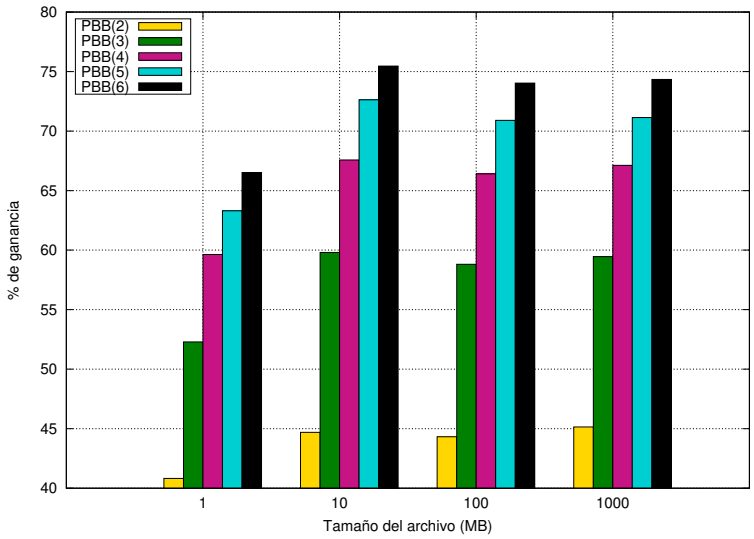


Figura 5.19: Porcentaje de ganancia obtenida por *PBB* en todas sus versiones con respecto a *secuencial* en el proceso de decodificación utilizando *tuberías de procesamiento*.

Discusión de resultados de decodificación En la Figura 5.20 se muestran, en el eje de abscisas, los Tr_{MT} producidos por las soluciones estudiadas y sus correspondientes versiones, realizando el proceso de decodificación para diferentes tamaños de archivos (eje de ordenadas).

El proceso de decodificación muestra que al incrementar el número segmentos s utilizados por PBB se logra disminuir el Tr_{MT} al procesar los archivos de distintos tamaños aún y cuando el 66.67 % de los datos requeridos son obtenidos a través de la red y el otro 33.33 % es leído desde el sistema de archivos. De igual forma al incrementar el valor de s se añaden más costos de gestión y transporte que reducen los Tr_{MT} y ganancia.

En la Figura 5.21 se muestran, en el eje de abscisas, el porcentaje de ganancia obtenido por *PBB* en todas sus versiones con respecto a *secuencial* para diferentes tamaños de archivos (eje de ordenadas).

Como se puede observar en la Figura 5.19 el porcentaje de ganancia incrementa al utilizar

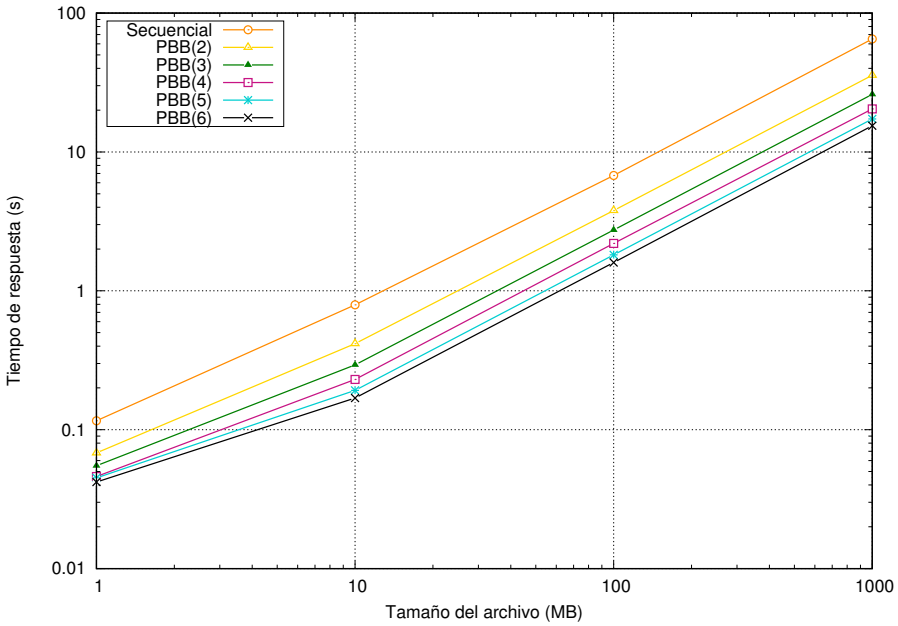


Figura 5.20: Comportamiento del tiempo de respuesta para las soluciones *secuencial* y *PBB* en todas sus versiones para el proceso de decodificación usando *tuberías de procesamiento*.

más tuberías de procesamiento al igual que las pruebas realizadas anteriormente confirmando que el comportamiento por los filtros utilizados se conserva aún y cuando son distribuidos de formas diferentes.

La flexibilidad de despliegue de contenedores proporcionados por la plataforma de contenedores *Docker* permite el lanzamiento de los VCs de Meta-Filtros en distintas infraestructuras, mientras que, el diseño de la arquitectura del método propuesto permite la generación de múltiples Meta-Filtros que contienen *variedad* de filtros de procesamiento comunicado por distintas interfaces de comunicación logrando mejorar los tiempos de comunicación entre filtros de procesamiento (memoria).

El esquema *divide y encapsula* permite la división de trabajo en múltiples Meta-Tuberías para conseguir mayor *velocidad* de procesamiento para grandes *volúmenes* de datos ya sea de forma centralizada (tuberías de procesamiento) o distribuidas (fábricas de procesamiento) que se adaptan a

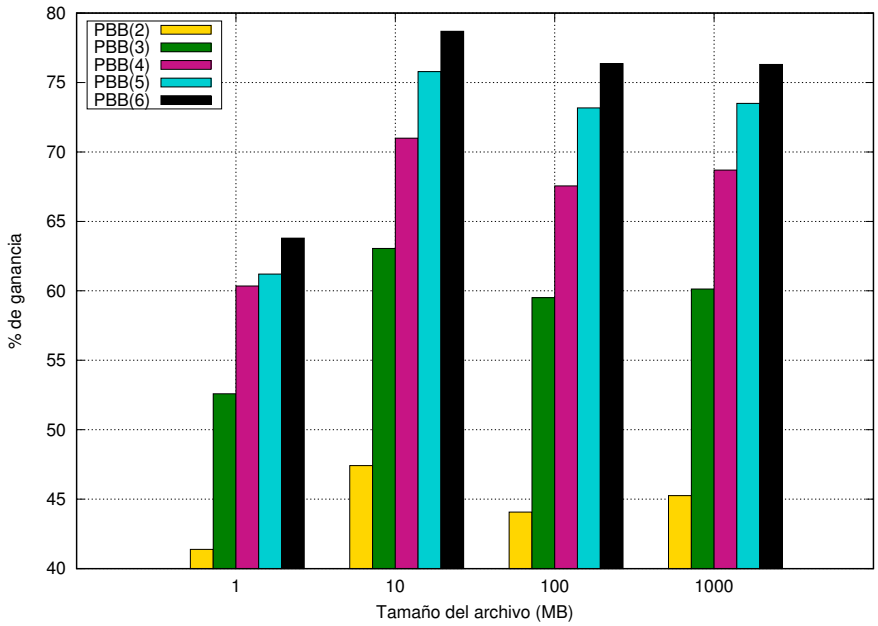


Figura 5.21: Porcentaje de ganancia obtenida por *PBB* en todas sus versiones con respecto a *secuencial* en el proceso de decodificación utilizando *tuberías de procesamiento*.

las necesidades del desarrollador, de la infraestructura utilizada y los requerimientos del procesamiento como se mostró a lo largo de este capítulo.

6

Conclusiones y trabajo futuro

A lo largo de este trabajo de tesis se presentó el diseño, implementación y evaluación de un método para la construcción de tuberías de procesamiento de datos para la nube, usando contenedores virtuales para resolver la problemática del procesamiento de grandes volúmenes de datos a través de diferentes infraestructuras en la nube.

6.1 Conclusiones generales

Desde un perspectiva cualitativa el método exhibe las siguientes características:

- *Variedad de procesamiento*:

  La unidad de construcción o Building Block (BB) propuesta, permite la encapsulación de f filtros de procesamiento en su interior formando un Meta-Filtro. Los Meta-Filtros pueden ser interconectados brindando así más variedad de procesamiento. Esta arquitectura cumple con el

primer objetivo específico, y la característica de *variedad* de procesos planteada en el objetivo general.

- *Almacenamiento en memoria*:

  El uso de la memoria como interfaz de comunicación utilizando la Interfaz de Programación de Aplicaciones (API, por sus siglas en inglés) de acceso a memoria desarrollada, permite incrementar la velocidad de procesamiento y comunicación entre filtros. El incrementar la velocidad de procesamiento es una de las características que debe de conseguir el método propuesto establecida en el objetivo general.

- *Esquema divide y encapsula*:

  Este esquema permite la división de carga de procesamiento de datos a múltiples Meta-Tuberías. Obteniendo *variedad* de procesamiento y un incremento en la *velocidad* de procesamiento de grandes *volúmenes* de datos. Con el esquema propuesto se consiguieron las tres características de variedad, volumen y velocidad establecidas en el objetivo general.

- *Flexibilidad de despliegue*:

  Característica brindada por el BB, que permite la consolidación de múltiples filtros en el interior de un solo Meta-Filtro o separarlos en múltiples Meta-Filtros. Estos Meta-Filtros pueden ser desplegados en diferentes infraestructuras en entornos que cumplan con sus dependencias. Al ser contenerizados, los Meta-Filtro fueron agregados a una imagen de contenedor (IC, por sus siglas en inglés) la cual incluye todas sus dependencias y el entorno necesario para su ejecución, generando una nueva CI llamada CI de Meta-Filtro. Las CIs de Meta-Filtro están listas para su ejecución, estas CIs de Meta-Filtro pueden ser distribuidas en diferentes infraestructuras que utilicen la misma plataforma de contenedores, para cada CI se deben crear archivos de configuración mediante planificador de conexiones y pueden ser ejecutadas por el constructor de Meta-tuberías propuesto para convertirse en contenedores virtuales (VC, por sus siglas en inglés) de Meta-Filtros que formarán parte de Meta-Tuberías. La generación de un esquema

de despliegue requerido por el segundo objetivo específico fue cumplido con el esquema *divide y encapsula*, mientras que el tercer objetivo fue cumplido con el planificador de conexiones y el constructor de Meta-Tuberías.

Desde un perspectiva cuantitativa el método exhibe las siguientes características:

- Al aplicar variedad de filtros de procesamiento, se consigue la reducción del espacio de almacenamiento requerido, utilizando un filtro de compresión. Esto permite agregar más filtros de procesamiento con un incremento en el espacio de almacenamiento menor o igual al no utilizar la compresión. Al reducir el contenido enviado entre filtros de procesamiento puede conseguir una aceleración en el tiempo de procesamiento.

- La utilización de la interfaz de memoria como medio de comunicación entre procesos dentro de un filtro de procesamiento, permite conseguir en la aplicación IDA una aceleración promedio de *1.4X* en el proceso de codificación y *1.6X* en el proceso de decodificación con respecto a utilizar el sistema de archivos. Esto ocurre debido a que la memoria tiene una mayor velocidad de acceso que la interfaz de sistema de archivos.

- El esquema *divide y encapsula* permite incrementar la *velocidad* de procesamiento de grandes *volúmenes* de datos en escenarios donde se requieren VCs de Meta-Filtros con *variedad* de procesos (filtros). Demostrado con la implementación del filtro IDA en diferentes escenarios.

El método propuesto permite encapsular *variedad* de filtros de procesamiento en un Meta-Filtro, generar CIs de Meta-Filtros para generar VC de Meta-Filtros configurables mediante archivos de configuración, manejar grandes *volúmenes* de datos e incrementar la *velocidad* de procesamiento al utilizar el esquema *divide y encapsula*. Los VCs de Meta-Filtros producidos son compatibles con infraestructuras locales, distribuidos en cluster y en la nube, cumpliendo el objetivo general de este trabajo de tesis.

6.2 Contribuciones

1. Una arquitectura de software modular que permite la creación de Meta-Filtros que incluye.
    - Un constructor de imágenes de contenedor con Meta-Filtros.
    - Un silo de imágenes de contenedor de Meta-Filtros que permite el almacenamiento y adquisición de imágenes de contenedor de Meta-Filtros creadas por los usuarios.
    - Una planificador de conexiones que permite la generación de archivos de configuración para las imágenes de contenedor de Meta-Filtros.
    - Un lanzador de contenedores virtuales de Meta-Filtros para generar Meta-Tuberías al desplegar las imágenes de contenedor de Meta-Filtros con sus respectivos archivos de configuración.
2. Un esquema para la división de carga de trabajo en Meta-Tuberías que permite utilizar múltiples Meta-Tuberías para el procesamiento de grandes *volúmenes* de datos. Este esquema permite aplicar una *variedad* de filtros de procesamiento (Meta-Filtros) y conseguir un incremento en la *velocidad* de procesamiento de los datos.

6.3 Dificultades

Conseguir las características de variedad de procesos, procesamiento de grandes volúmenes de datos e incrementar la velocidad de procesamiento implicó afrontar diferentes retos técnicos como lo son:

- La administración de los VCs de Meta-Filtros pertenecientes a una Meta-Tubería de procesamiento.
- Definir la forma de comunicación entre las distintas infraestructuras utilizadas para las pruebas.

- Establecer las configuraciones para las interfaces de comunicación de la capa de entrada entre los VCs de Meta-Filtros adyacentes.

6.4 Trabajo futuro

Las características que se pretenden integrar al método desarrollado son:

1. Realizar el análisis, selección e implementación de un balanceador de carga para permitir elasticidad bajo demanda.

2. Reducir las llamadas a la interfaz de red de los Meta-Filtros mediante la consolidación de resultados en memoria.

3. Definir un API para la construcción y despliegue de Meta-Tuberías definidas por software para el procesamiento de datos en múltiples infraestructuras.

A

Algoritmo IDA

En este apéndice, se describen los algoritmos de codificación y decodificación de IDA con la configuración $n = 5$ y $n = 3$.

A.1 Proceso de codificación

El proceso de codificación del algoritmo IDA con una configuración de $n = 5$ y $m = 3$ realiza el proceso descrito por el Algoritmo 5. En el cual se recibe el contenido por codificar F, el tamaño del archivo por codificar $|F|$ y las cinco salidas que serán usadas para su almacenamiento. Este proceso genera las n piezas de las cuales m son requeridas para realizar el proceso de decodificación.

Algoritmo 5 Proceso de codificación IDA(5,3)

Entrada: Contenido F, Tamaño del contenido $|F|$, lista de salidas $salidas = \{salida_1, salida_2, \ldots, salida_5\}$

1: $orden = 0, exp[256], log[256], b[3]$;

2: $generarCampoFinito(orden, exp, log)$;

3: $a[5][3] = \{\{'1','3','2'\},\{'1','1','1'\},\{'2','3','1'\},\{'2','2','3'\},\{'2','3','3'\}\};$

4: $tamanioPiezas = |F|/3 + 5;$

5: $residuo = |F|\,\%3$

6: $posicionpieza1 = 0, posicionpieza2 = 0, posicionpieza3 = 0, posicionpieza4 = 0, posicionpieza5 = 0;$

7: $i = 0, j = 0, t = 0;$

8: $pieza1 = crearArregloBytes(tamanioPiezas);$

9: $pieza2 = crearArregloBytes(tamanioPiezas);$

10: $pieza3 = crearArregloBytes(tamanioPiezas);$

11: $pieza4 = crearArregloBytes(tamanioPiezas);$

12: $pieza5 = crearArregloBytes(tamanioPiezas);$

13: **para** $(i = 0; i < 5; i++)$ **hacer**

14: **switch** (i)

15: **case** 0**:**

16: $pieza1[posicionpieza1] = residuo;$

17: $posicionpieza1++;$

18: **para** $(j = 0; j < 3; i++)$ **hacer**

19: $pieza1[posicionpieza1] = a[i][j];$

20: $posicionpieza1++;$

21: **fin para**

22: **case** 1**:**

23: $pieza2[posicionpieza2] = residuo;$

24: $posicionpieza2++;$

25: **para** $(j = 0; j < 3; i++)$ **hacer**

26: $pieza2[posicionpieza2] = a[i][j];$

27: $posicionpieza2++;$

```
28:     fin para
29:   case 2:
30:     pieza3[posicionpieza3] = residuo;
31:     posicionpieza3 + +;
32:     para (j = 0; j < 3; i + +) hacer
33:       pieza3[posicionpieza3] = a[i][j];
34:       posicionpieza3 + +;
35:     fin para
36:   case 3:
37:     pieza4[posicionpieza4] = residuo;
38:     posicionpieza4 + +;
39:     para (j = 0; j < 3; i + +) hacer
40:       pieza4[posicionpieza4] = a[i][j];
41:       posicionpieza4 + +;
42:     fin para
43:   case 4:
44:     pieza5[posicionpieza5] = residuo;
45:     posicionpieza5 + +;
46:     para (j = 0; j < 3; i + +) hacer
47:       pieza5[posicionpieza5] = a[i][j];
48:       posicionpieza5 + +;
49:     fin para
50:   end switch
51: fin para
52: nuevoContador = 0;
53: posicionEntrada = posicionArchivo1;
```

54: $contador = 4;$

55: **mientras** $(contador < tamanioPiezas)$ **hacer**

56: **para** $j = 0; j < 3; j++$ **hacer**

57: $b[j] = 0;$

58: **fin para**

59: $j = 0;$

60: **DO**$valor = F[nuevoContador];$

61: **si** $(valor > 255)$ **entonces**

62: $b[j++] = valor - 4294967040;$

63: **si no**

64: $b[j++] = valor;$

65: **fin si**

66: $posicionEntrada++;$

67: $nuevoContador++;$

68: **WHILE** $(((F! =' \backslash 0')\&\&(j < 3)\&\&(posicionEntrada < (|F| + 4))))$

69: **si** $(j < 0)$ **entonces**

70: **para** $(i = 0; i < 5; i++)$ **hacer**

71: $t = 0;$

72: **para** $(j = 0; j < 3; j++)$ **hacer**

73: **si** $(b[j] > 255)$ **entonces**

74: $break;$

75: **fin si**

76: $t = XorBB(t, exp[(log[a[i][j]] + log[b[j]])\,\%orden);$

77: **fin para**

78: **switch** (i)

79: **case** 0:

80: $pieza1[posicionpieza1] = t;$

81: $posicionpieza1++;$

82: **case** 1**:**

83: $pieza2[posicionpieza2] = t;$

84: $posicionpieza2++;$

85: **case** 2**:**

86: $pieza3[posicionpieza3] = t;$

87: $posicionpieza3++;$

88: **case** 3**:**

89: $pieza4[posicionpieza4] = t;$

90: $posicionpieza4++;$

91: **case** 4**:**

92: $pieza5[posicionpieza5] = t;$

93: $posicionpieza5++;$

94: **end switch**

95: **fin para**

96: **fin si**

97: $contador++;$

98: **fin mientras**

99: $enviarPieza(pieza1, salida_1);$

100: $enviarPieza(pieza2, salida_2);$

101: $enviarPieza(pieza3, salida_3);$

102: $enviarPieza(pieza4, salida_4);$

103: $enviarPieza(pieza5, salida_5);$

A.2 Proceso de decodificación

El proceso de decodificación del algoritmo IDA con una configuración de $n = 5$ y $m = 3$ realiza el proceso descrito por el Algoritmo 6. Este proceso requiere como parámetros de entrada tres piezas de las cinco producidas por el proceso de codificación, en su interior realiza las operaciones requeridas para la decodificación de los datos y devuelve como resultado el contenido original.

Algoritmo 6 Proceso de decodificación IDA(5,3)

Entrada: piezas de entrada $pieza1, pieza2, pieza3$

1: $orden, exp[255], log[255], a[3][3], g[3][3], d[3], PRIMITIVO = 369;$

2: $c, b[3], c3[12] = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\};$

3: $i, j, r, primera;$

4: $t, aux[3];$

5: $contadorArchivo1 = 0, contadorArchivo2 = 0, contadorArchivo3 = 0, contadorSalida = 0, tamanioPiezas = 0;$

6: $longtamanioFinal = 0, bytesAgregados = 15, residuoDelOriginal = 0, bytesSobrantes = 0;$

7: $tamanioPiezas = obtenerTamanioArchivo(pieza1)$

8: $generarCampoFinito()$

9: $residuoDelOriginal = contenido1[0];$

10: $bytesSobrantes = bytesAgregados - residuoDelOriginal;$

11: $tamanioFinal = tamanioPiezas * 3 - bytesSobrantes;$

12: $archivoDecodificado = crearArregloBytes(tamanioFinal)$

13: **para** $(i = 0; i < 5; i++)$ **hacer**

14: **switch** (i)

15: **case** 0**:**

16: $r = pieza1[contadorArchivo1]$

```
17:       contadorArchivo1++;
18:       para (j = 0; j < 3; i++) hacer
19:          a[i][j] = pieza1[contadorArchivo1];
20:          contadorArchivo1++;
21:       fin para
22:    case 1:
23:       r = pieza1[contadorArchivo2]
24:       contadorArchivo2++;
25:       para (j = 0; j < 3; i++) hacer
26:          a[i][j] = pieza2[contadorArchivo2];
27:          contadorArchivo2++;
28:       fin para
29:    case 2:
30:       r = pieza3[contadorArchivo3]
31:       contadorArchivo3++;
32:       para (j = 0; j < 3; i++) hacer
33:          a[i][j] = pieza3[contadorArchivo3];
34:          contadorArchivo3++;
35:       fin para
36:    end switch
37: fin para
38: invierteM();
39: primera = 1;
40: para j = 0; j < 3; j++ hacer
41:    aux[j] = &c3[4 * j];
42: fin para
```

```
43: DO
44: para (j = 0; j < 3; j++) hacer
45:    switch (j)
46:    case 0:
47:       c = pieza1[contadorArchivo1];
48:       contadorArchivo1++;
49:       si (pieza1! =' \0') entonces
50:          c3[4 * j] = c;
51:          d[j] = (aux[j]);
52:       fin si
53:    case 1:
54:       c = pieza2[contadorArchivo2];
55:       contadorArchivo2++;
56:       si (pieza2! =' \0') entonces
57:          c3[4 * j] = c;
58:          d[j] = (aux[j]);
59:       fin si
60:    case 2:
61:       c = pieza3[contadorArchivo3];
62:       contadorArchivo3++;
63:       si (pieza3! =' \0') entonces
64:          c3[4 * j] = c;
65:          d[j] = (aux[j]);
66:       fin si
67:    end switch
68: fin para
```

69: **si** $((contenido1 != '\backslash 0') \&\& (!primera))$ **entonces**

70: **para** $(i = r; i < 3; i++)$ **hacer**

71: $salidaFinal[contadorSalida] = b[i];$

72: $contadorSalida++;$

73: **fin para**

74: **fin si**

75: **si** $(contenido1 != '\backslash 0')$ **entonces**

76: **para** $(i = 0; i < 3; i++)$ **hacer**

77: $t = 0;$

78: **para** $(j = 0; j < 3; j++)$ **hacer**

79: $t = suma(t, mult(g[i][j], d[j]));$

80: $b[i] = t;$

81: **fin para**

82: **fin para**

83: **para** $(i = 0; i < r; i++)$ **hacer**

84: $archivoDecodificado[contadorSalida] = b[i];$

85: $contadorSalida++;$

86: **fin para**

87: **fin si**

88: **si** $((contenido1 == '\backslash 0') \&\& (r == 0))$ **entonces**

89: **para** $i = r; i < 3; i++)$ **hacer**

90: $archivoDecodificado[contadorSalida] = b[i];$

91: $contadorSalida++;$

92: **fin para**

93: **fin si**

94: $primera = 0$

95: **WHILE** $((contadorArchivo1 < tamanioPiezas))$

96: **devolver** $archivoDecodificado$

Bibliografía

[1] Abdelbaky, M., Diaz-Montes, J., Parashar, M., Unuvar, M., and Steinder, M. (2015). Docker containers across multiple clouds and data centers. In *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*, pages 368–371. IEEE.

[2] Abrams, S., Cruse, P., Kunze, J., and Minor, D. (2011). Curation micro-services: A pipeline metaphor for repositories. *Journal of Digital Information*, 12(2).

[3] Amazon (2017). Amazon web services: Amazon s3. Disponible en: `https://aws.amazon.com/es/s3/`.

[4] Anderson, C. (2015). Docker [software engineering]. *IEEE Software*, 32(3):102–c3.

[5] Anil Karmel, Ramaswamy Chandramouli, M. I. (2011). Draft definition of microservices, application containers and system virtual machines.

[6] (auth.), J. M. G. (2002). *Performance Modeling of Operating Systems Using Object-Oriented Simulation: A Practical Introduction*. Series in Computer Science. Springer US, 1 edition.

[7] Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

[8] Bessani, A., Correia, M., Quaresma, B., André, F., and Sousa, P. (2013). Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12.

[9] Bhudavaram, D. K., Naik, S., and Uthamanathan, S. (2016). File creation through virtual containers. US Patent 9,389,931.

[10] Bittman, T. and Leong, L. (2011). Worldwide archival storage solutions 2011-2015 forecast: Archiving needs thrive in an information-thirsty world. *IDC. Market Analysis*, pages 1–21.

[11] Boettiger, C. (2015). An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79.

[12] Brewer, E. A. (2015). Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 167–167, New York, NY, USA. ACM.

[13] Buschmann, F., Henney, K., and Schmidt, D. C. (2007). *Pattern-oriented software architecture, on patterns and pattern languages*, volume 5. John wiley & sons.

[14] Buyya, R., Pathan, M., and Vakali, A. (2008). *Content delivery networks*, volume 9. Springer Science & Business Media.

[15] Chen, M., Mao, S., and Liu, Y. (2014). Big data: a survey. *Mobile Networks and Applications*, 19(2):171–209.

[16] Chung, J. Y., Joe-Wong, C., Ha, S., Hong, J. W.-K., and Chiang, M. (2015). Cyrus: Towards client-defined cloud storage. In *Proceedings of the Tenth European Conference on Computer Systems*, page 17. ACM.

[17] de Alfonso, C., Calatrava, A., and Moltó, G. (2017). Container-based virtual elastic clusters. *Journal of Systems and Software*, 127:1–11.

[18] Docker (2016a). The docker commands. Disponible en: `https://docs.docker.com/engine/reference/commandline/`.

[19] Docker (2016b). Manage data in containers. Disponible en: `https://docs.docker.com/engine/tutorials/dockervolumes/`.

[20] Docker (2016c). Welcome to the docs. Disponible en: `https://docs.docker.com/`.

[21] Docker (2017). Docker swarm overview. Disponible en: https://docs.docker.com/swarm/overview/.

[22] Draft, N. (2014). big data interoperability framework: volume 1, definitions. *NIST special publication, Information Technology Laboratory, Gaithersburg*, 23.

[23] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2016). Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*.

[24] Dua, R., Raja, A. R., and Kakadia, D. (2014). Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE.

[25] Española, R. A. (2017). Dle: Silo. Disponible en: http://dle.rae.es/?id=XtsbOwr.

[26] Foundation, F. S. (2007). Gnu general public license. Disponible en: http://www.gnu.org/licenses/gpl.html.

[27] Fundation., T. A. S. (2017a). Spark: Lightning-fast cluster computing. Disponible en: http://spark.apache.org/.

[28] Fundation., T. A. S. (2017b). Welcome to apache™ hadoop®!: What is apache hadoop? Disponible en: http://hadoop.apache.org.

[29] Gantz, J. and Reinsel, D. (2011). Extracting value from chaos. *IDC iview*, 1142:1–12.

[30] Gantz, J. and Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007:1–16.

[31] Gay, W. W. (2000). *Linux Socket Programming: By Example*. Que Corp., Indianapolis, IN, USA.

[32] Gerlach, W., Tang, W., Keegan, K., Harrison, T., Wilke, A., Bischof, J., D'Souza, M., Devoid, S., Murphy-Olson, D., Desai, N., et al. (2014). Skyport: container-based execution environment

management for multi-cloud scientific workflows. In *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*, pages 25–32. IEEE Press.

[33] Gonzalez, J. L., Diaz-Perez, A., Sosa-Sosa, V., Carretero Perez, J., and Yanez-Sierra, J. (2016). Sacbe: A modular software architecture for secure, reliable and flexible end-to-end cloud storage (sin publicar).

[34] Gonzalez, J. L., Perez, J. C., Sosa-Sosa, V. J., Sanchez, L. M., and Bergua, B. (2015). Skycds: A resilient content delivery service based on diversified cloud storage. *Simulation Modelling Practice and Theory*, 54:64–85.

[35] Google (2017). Conoce drive. Disponible en: `https://www.google.com/intl/es-419/drive/`.

[36] Grawinkel, M., Mardaus, M., Süß, T., and Brinkmann, A. (2015). Evaluation of a hash-compress-encrypt pipeline for storage system applications. In *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pages 355–356. IEEE.

[37] Humphreys, J. and Grieser, T. (2006). Mainstreaming server virtualization: The intel approach. *IDC, White Paper*.

[38] Inc., D. (2017). Overview of docker hub. Disponible en: `https://docs.docker.com/docker-hub/`.

[39] Jaramillo, D., Nguyen, D. V., and Smart, R. (2016). Leveraging microservices architecture by using docker technology. In *SoutheastCon, 2016*, pages 1–5. IEEE.

[40] Jedidiah, Y.-S., Arturo, D.-P., Victor, S.-S., and JL, G. (2015). Towards secure and dependable cloud storage based on user-defined workflows. In *Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on*, pages 405–410. IEEE.

[41] Joy, A. M. (2015). Performance comparison between linux containers and virtual machines. In *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, pages 342–346. IEEE.

[42] Kakadia, D. (2015). *Apache Mesos Essentials*. Packt Publishing Ltd.

[43] Kang, H., Le, M., and Tao, S. (2016). Container and microservice driven design for cloud infrastructure devops. In *Cloud Engineering (IC2E), 2016 IEEE International Conference on*, pages 202–211. IEEE.

[44] Kozhirbayev, Z. and Sinnott, R. O. (2017). A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175–182.

[45] Li, Z., Kihl, M., Lu, Q., and Andersson, J. A. (2017). Performance overhead comparison between hypervisor and container based virtualization. In *Advanced Information Networking and Applications (AINA), 2017 IEEE 31st International Conference on*, pages 955–962. IEEE.

[46] Mell, P. and Grance, T. (2011). The nist definition of cloud computing.

[47] Microsoft (2017). One drive: Página principal. Disponible en: `https://onedrive.live.com/about/es-mx/`.

[48] Newman, S. (2015). *Building Microservices*. .º'Reilly Media, Inc."̈.

[49] Pathan, M., Buyya, R., and Vakali, A. (2008). Content delivery networks: State of the art, insights, and imperatives. In *Content Delivery Networks*, pages 3–32. Springer.

[50] Preeth, E., Mulerickal, F. J. P., Paul, B., and Sastri, Y. (2015). Evaluation of docker containers based on hardware utilization. In *Control Communication & Computing India (ICCC), 2015 International Conference on*, pages 697–700. IEEE.

[51] Rabin, M. O. (1989). Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348.

[52] Rad, B. B., Bhatti, H. J., and Ahmadi, M. (2017). An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228.

[53] Ritchie, D. M. (1980). The evolution of the unix time-sharing system. In *Language Design and Programming Methodology*, pages 25–35. Springer.

[54] Rivest, R. (1992). The md5 message-digest algorithm.

[55] Scarfone, K. (2011). *Guide to security for full virtualization technologies*, volume 800. DIANE Publishing.

[56] Sharma, P., Chaufournier, L., Shenoy, P., and Tay, Y. (2016). Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, page 1. ACM.

[57] Shen, L., Feng, S., Sun, J., Li, Z., Wang, G., and Liu, X. (2015). Clouds: A multi-cloud storage system with multi-level security. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 703–716. Springer.

[58] Stevens, W. R. (2012). Unix network programming: Networking apis: Sockets and xti-volume 1, 2nd.

[59] Tanenbaum, A. S. and Van Steen, M. (2007). *Distributed systems: principles and paradigms*. Prentice-Hall.

[60] Wein, J. M., Kloninger, J. J., Nottingham, M. C., Karger, D. R., and Lisiecki, P. A. (2007). Content delivery network (cdn) content server request handling mechanism with metadata framework support. US Patent 7,240,100.