Formando lideres para la Construcción de un Nuevo País en Paz

IMPLEMENTACIÓN DE SERVIDOR DE PRUEBAS A PARTIR DE CONTENEDORES DOCKER Y VIRTUALIZACIÓN ANIDADA EN LA EMPRESA DIGITALWARE

Autor

WILMAR DANIEL BUSTOS MORALES

Director

JOSE DANIEL RAMÍREZ CORZO

INGENIERÍA ELECTRÓNICA
DEPARTAMENTO DE ELÉCTRICA, ELECTRÓNICA SISTEMAS Y TELECOMUNICACIONES
FACULTAD DE INGENIERÍAS Y ARQUITECTURA
UNIVERSIDAD DE PAMPLONA
PAMPLONA
28 DE NOVIEMBRE DE 2020

UNIVERSIDAD DE PAMPLONA
FACULTAD DE INGENIERÍAS Y ARQUITECTURA
DEPARTAMENTO DE INGENIERÍAS ELÉCTRICA, ELECTRÓNICA, SISTEMAS Y TELECOMUNICACIONES
PROGRAMA DE INGENIERÍA ELECTRÓNICA
TRABAJO PRESENTADO PARA OPTAR POR EL TITULO DE INGENIERO ELECTRÓNICO

TEMA:

IMPLEMENTACIÓN DE SERVIDOR DE PRUEBAS A PARTIR DE CONTENEDORES DOCKER Y VIRTUALIZACIÓN ANIDADA EN LA EMPRESA DIGITALWARE.

FECHA DE INICIO DEL TRABAJO: 06 de Julio 2020
FECHA DE TERMINACIÓN DEL TRABAJO: 19 de diciembre 2020

NOMBRES Y FIRMAS DE AUTORIZACIÓN PARA LA SUSTENTACIÓN:

_____
Wilmar Daniel Bustos Morales
AUTOR

_____
Jose Daniel Ramírez Corzo
DIRECTOR

_____
Jose Daniel Ramírez Corzo
DIRECTOR DE PROGRAMA

JURADO CALIFICADOR:

_____
Jose Daniel Ramírez Corzo

_____
Luis Alberto Muñoz Bedoya

_____
Ronald de Jesús Torres Flórez

PAMPLONA NORTE DE SANTANDER
COLOMBIA
15 de Diciembre de 2020

RESUMEN

DigitalWare cuenta con más de 700 programas no web renderizados en Internet Explorer. Para asegurarse de que el software desarrollado cumpla con las expectativas de los clientes, las pruebas constantes son necesarias. El presente proyecto tiene como objetivo la implementación de un nuevo servidor de pruebas que dé solución a dos de las principales problemáticas que se presentan en el área de testeo de la empresa; El tiempo de ejecución de las pruebas y su comportamiento secuencial al momento de ejecutarse. Para su desarrollo se planteó la implementación de un servidor que permita el despliegue de varios contenedores Docker que posibiliten la ejecución de pruebas de forma paralela tan solo con un equipo físico, cada contenedor llevará a cabo en su interior un proceso de virtualización anidada, en donde el contenedor ejecutará una máquina virtual de Windows y esto dará acceso al Driver de Internet Explorer que permitirá la ejecución de pruebas de manera remota. Al Finalizar el proyecto se logró ejecutar pruebas dentro del contenedor Docker permitiendo así, la ejecución sin la renderización del navegador y directamente en el Host, permitiendo la ejecución de pruebas automatizadas dentro del contenedor y abriendo la posibilidad de ejecutar las pruebas en contenedores paralelos dentro de un solo Host físico reduciendo recursos y tiempo en futuras implementaciones.

Tabla de contenido

INTRODUCCIÓN

La acción de probar aplicaciones en diferentes escenarios, para detectar errores que puedan representar un problema para los usuarios que usan algún tipo de software, es de suma importancia en la actualidad. Con el auge de Start ups especializadas en el desarrollo de software, esta parte es relevante, debido a que un error o mal funcionamiento no detectado antes de que un servicio o software entre en producción, puede perjudicar gravemente a la empresa y representa un inconveniente para el usuario. En Digitalware, específicamente en el área de pruebas, se realizan este tipo de ensayos y procesos para garantizar el funcionamiento de los programas que se ofrecen a los usuarios, siguiendo un proceso que finaliza con la ejecución de pruebas automatizadas en un servidor. En este servidor se generan reportes que sirven como prueba de que un test se ejecutó correctamente en un programa específico. Sin embargo, una de las limitantes más grandes que se presentan en esta área, es la cantidad de pruebas que se deben realizar y, además, por su naturaleza, los programas se ejecutan en Internet Explorer, esto limita que se puedan aplicar soluciones relativamente sencillas relacionadas con procesos paralelos y virtualización con contenedores Docker simples.

Para poder ejecutar las pruebas dentro de contenedores Docker que permitan una ejecución dinámica, se aplicaron conceptos relacionados con visualización anidada y la posibilidad de ejecución de procesos en segundo plano a través de herramientas descritas en esta tesis.

PROBLEMA Y JUSTIFICACIÓN

Para cualquier empresa de software es de suma importancia el testeo de sus aplicaciones y productos, para así poder ofrecer a sus clientes un software empresarial de alta calidad y eficiencia. Por tal motivo, el testeo constante e identificación de problemas es una de las prioridades del área de pruebas de cualquier compañía y DigitalWare no es la excepción, ya que, cuenta con más de 700 programas empresariales y la identificación de errores es una prioridad que se desarrolla a través de la implementación de pruebas automatizadas.

Estas pruebas son enviadas a un servidor en grandes cantidades y se ejecutan una a una. Este modo de funcionamiento; aunque cumple su función, no es óptimo para el volumen de pruebas que se envían constantemente. Debido a esto, se ocupa más espacio e incrementa el consumo energético en la empresa. Actualmente si quisieran ejecutar más pruebas de manera paralela, tendrían que instalar un nuevo servidor físico. Las consecuencias de no implementar una solución a lo expuesto anteriormente son el atraso en los tiempos de entrega de las pruebas y por ende un aumento en el tiempo que le toma a una aplicación nueva o actualizada llegar a producción.

El proyecto propuesto, se desarrollará con la finalidad de incrementar la eficiencia en la ejecución de pruebas automatizadas a nivel de servidor, a través de la implementación de un nuevo servidor basado en contenedores con interfaz gráfica virtualizada. Cada contenedor construido se ejecutará de forma paralela, permitiendo así, la ejecución de un mayor número de pruebas automatizadas en una misma máquina, lo cual permitirá la reducción del tiempo total de ejecución de la pruebas enviadas y permitirá optimizar el modo en que se trabaja en el área de pruebas de DigitalWare, de manera que cada persona relacionada con el área de pruebas pueda desplegar un contenedor y enviar un paquete de pruebas independiente de los procesos que estén llevando a cabo otros ingenieros en el servidor, reduciendo así, interrupciones.

OBJETIVO GENERAL

- Implementar un servidor de pruebas a través de virtualización anidada y contenedores Docker

OBJETIVOS ESPECÍFICOS

- Definir las tecnologías a usar con base en las necesidades del área de pruebas de DigitalWare
- Implementar el ambiente de pruebas usando contenedores Docker y Microsoft Azure
- Validar el sistema de funcionamiento a través de la ejecución de pruebas automatizadas

ACOTACIONES

El proyecto propuesto se desarrolló haciendo uso de contenedores Docker basados en Ubuntu 20.04 que en su interior; en una capa virtualizada, ejecutarán un Snapshot de una máquina virtual del sistema operativo Windows 10 ejecutada con la herramienta de virtualización Qemu; esto permite el acceso a el navegador de Internet Explorer de manera externa, haciendo posible su ejecución en memoria. Para la ejecución en memoria se usará la herramienta XVFB para crear un display virtual dentro del contenedor Docker.

GLOSARIO

- **QEMU:** QEMU es un emulador y virtualizador genérico de la CPU, lo que le diferencia de VMware y similares, que sólo permiten virtualizar.[11]
- **KVM:** KVM (para Máquina Virtual basada en el Kernel) es una solución de virtualización completa para Linux en hardware x86 que contiene extensiones de virtualización (Intel VT o AMD-V).[12]
- **Hipervisor:** Un hipervisor, también conocido como monitor de máquina virtual o VMM, es un software que crea y ejecuta máquinas virtuales (VM). Un hipervisor permite que una computadora anfitriona sea compatible con varias VM invitadas al compartir virtualmente sus recursos [13]
- **Virtualización anidada:** La virtualización anidada se refiere a la ejecución de un hipervisor "dentro" de otro hipervisor en una instancia virtual. En otras palabras, un hipervisor está efectivamente anidado dentro de otro hipervisor. [14]
- **Testing:** Es un proceso para evaluar la funcionalidad de una aplicación de software con la intención de determinar si el software desarrollado cumple o no los requisitos especificados e identificar los defectos para garantizar que el producto esté libre de defectos a fin de producir un producto de calidad. [15]
- **WinAppDriver:** Windows Application Driver (WinAppDriver) es un servicio de apoyo a la automatización de pruebas de interfaz de usuario en aplicaciones de Windows. Este servicio permite probar aplicaciones de Plataforma Universal de Windows (UWP), Formularios de Windows (WinForms), Windows Presentation Foundation (WPF) y Windows Clásico (Win32) en PCs con Windows 10. [16]
- **Host:** Un host es un ordenador al que se puede acceder a través de una red. Puede ser un cliente, un servidor o cualquier otro tipo de computadora. Cada host tiene un identificador único llamado nombre de host que permite a otras computadoras acceder a él. [17]

- **DockerFile:** Un Dockerfile es un documento de texto que contiene todos los comandos que un usuario puede llamar en la línea de comandos para ensamblar una imagen. Usando Dockerfile, los usuarios pueden crear una construcción automática que ejecuta varias instrucciones de la línea de comandos en sucesión. [18]

CAPÍTULO 1
MARCO TEÓRICO

Autor: Wilmar Daniel Bustos Morales
Director: Jose Daniel Ramírez Corzo 12

1.1 Marco Teórico

En medio del proceso de desarrollo y de despliegue de un software existe un área que permite la detección de problemas y garantiza que los programas desarrollados cumplan con su función. El área de testing se centra en dos aspectos relevantes, por un lado el funcionamiento del programa (Debugging) y por otro la verificación de que este cumpla su función (Testing) como explica 1957 C.Baker en su artículo *Review of D.D. McCracken's Digital Computer Programming.*

Debido a que el objetivo principal de las pruebas de software es comprobar la calidad de los programas, su uso se ha convertido en una prioridad y necesidad en el mundo del desarrollo y a su vez se han encontrado formas más eficientes de realizarlas. Un ejemplo de esto es la incorporación de las pruebas automatizadas para reducir tiempos de las pruebas realizadas y disminuir el error humano en este proceso.

Dependiendo de la empresa, los modelos y métodos para llevar a cabo la ejecución de las pruebas automatizadas anteriormente mencionadas, pueden variar debido a características propias de los programas a testear, los métodos o herramientas usadas pueden no ser los más eficientes, por lo tanto, incurre en un incremento en los tiempos de ejecución; lo cual puede traducirse como un aumento en los costos de la empresa.

Una herramienta ampliamente ocupada en la actualidad por Ingenieros en el área, tanto de pruebas como de desarrollo son los contenedores como Docker y máquinas virtuales tales como Qemu en sistemas Linux. Un contenedor, como se define [1] "*Es un entorno de virtualización a nivel de sistema operativo que utiliza contenedores de software para proporcionar aislamiento entre las aplicaciones*" [1] Esta herramienta abre la posibilidad de evitar incompatibilidades al momento de desarrollar código y ejecutar aplicaciones estandarizando un entorno de ejecución, debido a que aísla el entorno de trabajo del sistema operativo del host.[2] Desde un cierto punto de vista un contenedor no se aleja de lo que es una máquina virtual, sin embargo, el primero solo requiere la configuración mínima necesaria del software de un sistema operativo para ejecutar una aplicación. En general la virtualización que proveen estas dos herramientas son consideradas una innovación ampliamente usada en compañías de tecnología, incluyendo al área de pruebas de cada una de ellas, debido a que abren la posibilidad de tener procesos paralelos en un mismo equipo físico;[3] que en situaciones normales no soportan la ejecución de tareas paralelas como las que se llevan a cabo en DigitalWare.

Qemu por su parte es un hipervisor que permite el despliegue de máquinas virtuales. Una de las características mas importantes de esta herramienta, la

cual es usada en el desarrollo de la tesis, es la gestión de estas máquinas virtuales a través de una la consola, es decir, que la herramienta no cuenta con na interfaz gráfica. Esta característica facilita la realización de uno de los procesos más importantes de la tesis desarrollada la cual es la renderización de la maquina virtual desplegada en un display virtual.

Un contenedor se define como un objeto de aislamiento virtualizado para ejecutar aplicaciones sin un impacto en el sistema operativo en donde se ejecuta. Debido a esto, se asegura que el contenedor no tenga conocimiento sobre las características o procesos de la maquina anfitriona.[4]

Basado en los resultados logrados en el artículo titulado: “Performance Evaluation of Docker Container and Virtual Machine”[5] se observa que los contenedores Docker tienen un mejor rendimiento que las máquinas virtuales en pruebas de rendimiento de CPU, test de carga y en la medición de velocidad de operación: sumando a esto el hecho de que los contenedores Docker no requieren una interfaz gráfica y pueden ejecutarse paralelamente de manera sencilla, hacen que sea la opción más viable para ser el ambiente de ejecución de las pruebas automatizadas objetivo.

Un estudio llevado a cabo por la compañía IBM Research demostró que el rendimiento de un contenedor Docker es igual o superior al rendimiento de KVM en cuanto a rendimiento de CPU, memoria y red. Ambas tecnologías no demuestran ser una carga para la CPU y la memoria sin embargo KVM abstrae la topología del procesador y del hardware en el que se ejecuta, por lo tanto, no permite que la optimización tome lugar.[6]

Debido a la naturaleza de las pruebas y características de Internet Explorer en las que se implementará la solución planteada, el uso de Docker o máquinas virtuales por sí mismas, no satisfacen las condiciones necesarias para que estas se ejecuten en un servidor sin interfaz gráfica. Por lo cual, uno de los métodos investigados que pueden proveer una solución a esta problemática es la virtualización anidada junto a las dos herramientas anteriormente mencionadas.

Internet Explorer es un navegador desarrollado por Microsoft, el cual lanzó la primera versión de este en el año 1995 a partir de entonces y en el año 2013 se liberó la versión actualmente usada. La versión 11 de Internet Explorer. Si bien se incluyeron diferentes actualizaciones de compatibilidad propias de la época en que salió a producción. Este no cuenta con características como el modo de ejecución headless o modo sin interfaz grafica como los navegadores mas recientes como Google Chrome o Firefox.

La virtualización anidada como lo define el artículo Virtually timed ambients: A calculus of nested virtualization “permite a una máquina virtual; que es una capa de software que representa un entorno de ejecución, ser colocada dentro de otra máquina virtual”.[7] Este método permite la implementación de herramientas propias de un sistema operativo (Ubuntu 20.04) sobre programas que solo están disponibles en otro sistema operativo (*Internet Explorer en Windows 10*) y un comportamiento similar al descrito en el artículo titulado: “Nosv: A lightweight nested-virtualization VMM for hosting high performance computing on cloud”[8]

CAPÍTULO 2

Metodología

Autor: Wilmar Daniel Bustos Morales
Director: Jose Daniel Ramírez Corzo 16

2.1 Metodología

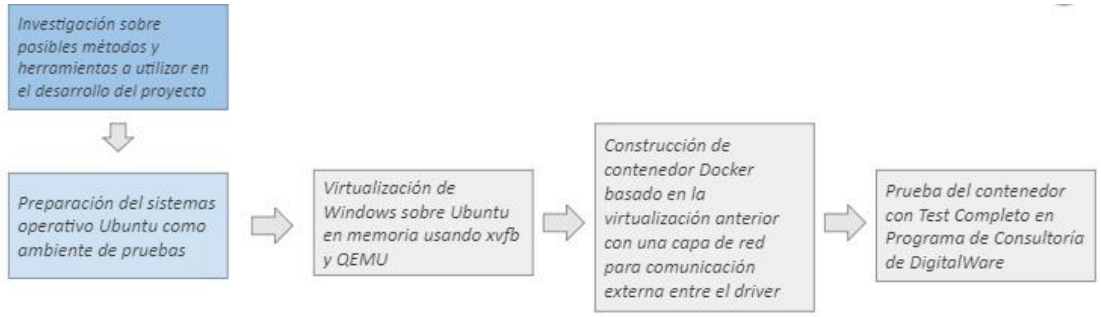


Figura 1. Ruta de desarrollo de la Metodología. Elaboración propia.

El desarrollo de este proyecto se baso en la ruta mostrada en la figura 1 en donde inicialmente se realizó una investigación con respecto a las tecnologías a usar y posteriormente se preparo el entorno donde se desarrollaron los pasos relacionados a la solución de la problemática principal.

2.1.1 Investigación sobre posibles métodos y herramientas a utilizar en el desarrollo del proyecto

Inicialmente, se llevó a cabo un estudio de las diferentes soluciones que pudieran conceder un buen rendimiento y permitieran el cumplimiento de lo que se esperaba lograr. Primeramente, debido a su capacidad de ejecutar distintos contenedores al mismo tiempo, se decidió aplicar un concepto puramente de Docker en Windows.

Primero se exploró está posibilidad con navegadores más populares, como Chrome y Firefox. La ejecución de estos dos navegadores se efectuaba, sin embargo, al realizar el mismo proceso con nuestro navegador objetivo (Internet Explorer); el resultado no fue el mismo.

Google Chrome y Firefox son navegadores recientes, que reciben soporte de manera constante y por lo tanto poseen una característica nueva llamada “headless mode”. Este modo permite la ejecución de cualquier proceso que realice Chrome en segundo plano, Es decir que los procesos que lleve a cabo Chrome no se renderizan en pantalla. Esta característica es bastante útil al realizar procesos automatizados que necesiten ejecutarse al mismo tiempo en un mismo entorno. Sin embargo, este comportamiento no está disponible para Internet Explorer, por lo tanto, un enfoque directo usando Docker, no fue una opción viable para solucionar el problema, debido a que, aunque se quiera ejecutar un contenedor Docker teniendo Internet Explorer este no se ejecutará ya que requerirá que se renderice la interfaz de dicho navegador.

En vista de la problemática presentada anteriormente, se decidió explorar otras herramientas que pudieran proveer una solución para el problema de la renderización de Internet Explorer. Existe una herramienta llamada xvfb que permite la ejecución de procesos en una salida de video virtual. Es decir, su renderización se lleva a cabo en memoria. Sin embargo, para poder usar este recurso era necesario de algún modo ejecutar Internet Explorer en un sistema Linux, una de las herramientas usadas para poder ejecutar un .exe desde un sistema Linux fue Wine, sin embargo, esta herramienta no proporcionó un resultado favorable con respecto a la ejecución del navegador por lo que se tuvo que descartar.

Posteriormente a investigaciones sobre proyectos similares se encontró un enfoque prometedor que combinaba las tecnologías exploradas e indagadas anteriormente. Una extensión del proyecto Aerokube, permitía la ejecución de navegadores Windows en Linux de manera virtualizada y no renderizada en pantalla, usando el gestor de máquinas virtuales QEMU. Empleando el enfoque descrito en su cuenta oficial de Github, se pudo solucionar la problemática principal con respecto a las limitaciones del navegador

2.1.2 Preparación del sistema operativo Ubuntu como ambiente de pruebas

Con el fin de decidir el entorno en donde se desarrolló el proyecto, se tuvo en cuenta las diferentes limitaciones y herramientas necesarias para llevar a cabo la creación de los contenedores. En Primer lugar, la tecnología de virtualización y contenedores Docker está disponible en distintos sistemas, por lo que a primera vista esto permitirá más flexibilidad en cuanto al sistema operativo a utilizar, sin embargo, la necesidad de usar xvfb (virtual frame buffer), para ocultar en una capa virtualizada el proceso de ejecución de una máquina virtual, es algo que se puede ejecutar en sistemas Linux. Debido a esto, se procedió a elegir Ubuntu 20.04 como entorno de desarrollo de los procesos y contenedores Docker.

La instalación para iniciar las pruebas se llevó a cabo como una partición de Ubuntu 20.04 junto a Windows 10 en una máquina Lenovo G40 sin ningún cambio o configuración adicional.

Inicialmente, se comprobó que la virtualización estaba activada en el sistema Ubuntu. Para verificar esta funcionalidad, se debe abrir en la terminal de Ubuntu el siguiente comando mostrado en la Figura 1.

Figura 1. Verificación de Soporte para virtualización. Elaboración propia.

El comando anteriormente mencionado, comprueba la existencia del archivo kvm en Ubuntu, el cual nos permite ejecutar máquinas virtuales en este sistema. Después de comprobar la existencia de este archivo, se procedió a instalar Qemu (emulador y virtualizador a nivel de hardware) y git. Una vez instalado debe mostrar la descripción de la versión a partir del comando mostrado en a Figura 2.

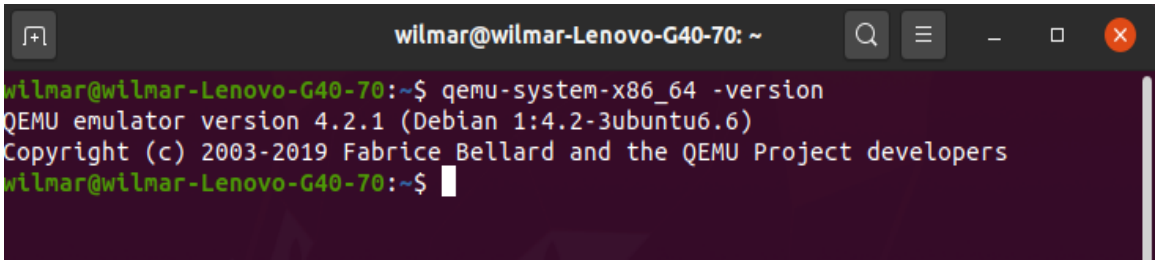


Figura 2. Versión instalada de qemu. Elaboración propia

A continuación, se procedió a crear una partición en el disco en donde se instaló la máquina virtual de Windows. usando el comando qemu-image; como se muestra en la *Figura 3.* Esta instrucción permite asignar una cantidad determinada de memoria a la máquina virtual a usar; En este caso se usó una capacidad de 40 GB para tener espacio suficiente al instalar la dependencia y herramientas necesarias dentro de Windows.

```
wilmar@wilmar-Lenovo-G40-70:~/windows-images$ qemu-img create -f qcow2 hdd.img 40G
```

Figura 3. Creación de imagen de disco duro en donde se instaló el sistema que se ejecutara con el gestor de máquinas virtuales qemu . Elaboración propia.

2.1.3 Virtualización de Windows sobre Ubuntu en memoria usando xvfb y QEMU

Uno de los requisitos de las pruebas es que deben ejecutarse en Internet Explorer, por lo tanto, el uso de Windows es necesario. Para que las pruebas se puedan ejecutar dentro de un contenedor Docker, se necesita renderizar en memoria la interfaz gráfica de la máquina virtual con Windows y ejecutando

Internet Explorer. Esto se llevará a cabo haciendo uso del Virtual Framebuffer disponible en sistemas Linux para evitar la necesidad de una salida gráfica.

Ya con una imagen del disco de almacenamiento disponible para la máquina virtual, se procedió a ejecutar el comando mostrado en la Figura 4. Este comando permite la ejecución de la máquina virtual a partir del archivo ISO descargado anteriormente. También habilita la comunicación TCP en el puerto 4444, el cual será usado para llevar a cabo pruebas junto a la especificación del disco de almacenamiento a utilizar.

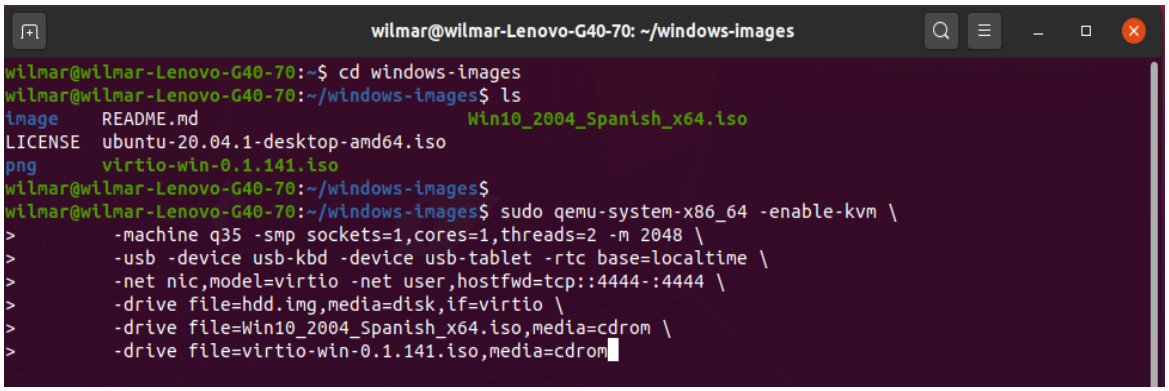


Figura 4. Instrucción para ejecución inicial de la máquina virtual de Windows en Ubuntu. Elaboración propia.

QEMU se inicia de forma visible, de este modo se llevó a cabo la instalación del sistema Windows de manera corriente, en donde se eligió Windows 10 Pro como el sistema de preferencia y se eligió el controlador virtio x84 cuando nos pregunte por un driver.

Cuando Windows se inició, se procedió a configurar el sistema actualizando manualmente el driver virtio descargado anteriormente, luego se procedió a deshabilitar el Firewall para que la máquina virtual permitiera la comunicación con el puerto 4444(*Figura 5*).

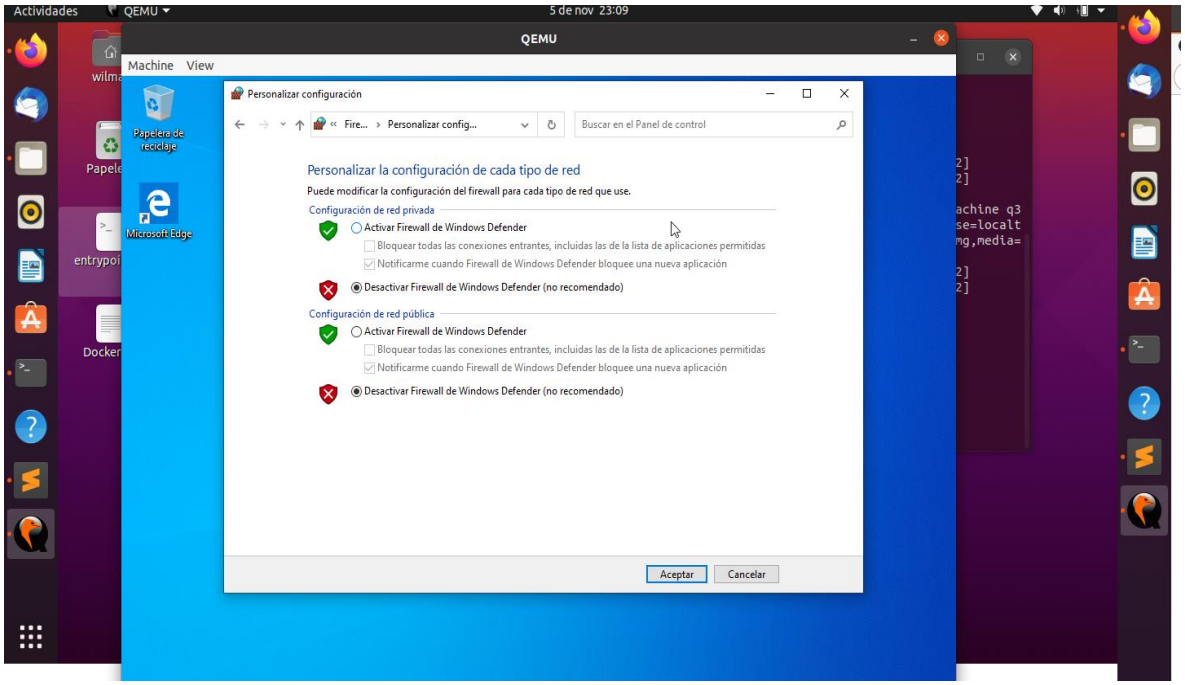


Figura 5. Proceso para Deshabilitar Firewall y permitir comunicación por el puerto 4444. Elaboración propia

Para preparar el entorno para una prueba de comunicación externa, Inicialmente se usó Selenium y eso hizo necesario el uso del driver de Internet Explorer de Selenium, sin embargo, debido a un cambio en el framework de pruebas, se hizo necesaria la instalación del Windriver para realizar las pruebas con Appium y no con Selenium.

Para ejecutar la máquina virtual de manera dinámica y más rápida, se tomó un snapshot de todas las configuraciones llevadas a cabo hasta ahora. Un snapshot permite guardar todos los datos, configuraciones, y procesos que se llevan a cabo en una máquina virtual en un tiempo determinado. Esto incluye el proceso de arranque e inicio del sistema emulado como una máquina virtual.

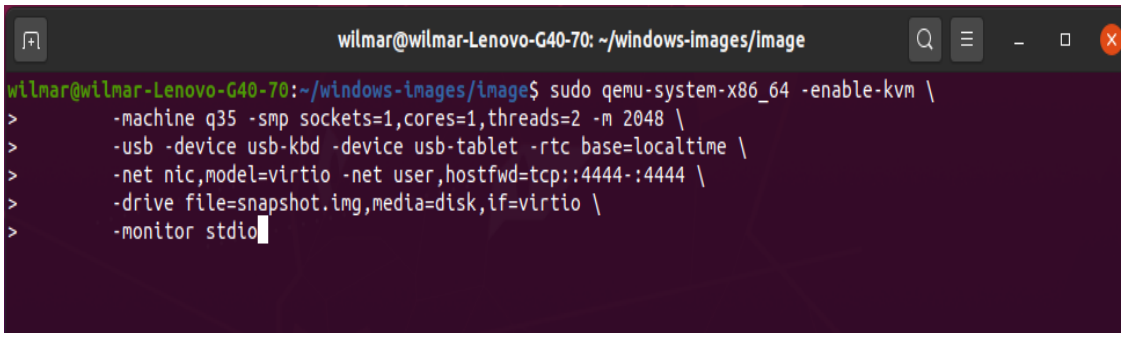


Figura 6. Comandos para la ejecución de Snapshot de la máquina virtual con cambios previamente hechos en el Firewall. Elaboración propia

Luego se ejecutó el snapshot guardado con QEMU y este ejecuto la máquina virtual con las últimas modificaciones hechas y en la última pantalla usada. Para controlar Internet Explorer desde un driver, es necesario deshabilitar las opciones de seguridad que este posee. Luego se ejecutó el WinAppDriver para permitir la comunicación con el código, ya que es un intermediario entre Internet Explorer y el código que contiene todas las instrucciones y comandos que este debe ejecutar.

Al final el proceso anterior, se guardó un Snapshot de nuevo y este último se movió a una carpeta a parte con el archivo hdd.img *(imagen del disco de almacenamiento)* Llegados a este punto fue posible ejecutar la máquina virtual desde el Snapshot (*Figura 7*) y enviar un código de prueba desde el PC host hacia la máquina virtual.

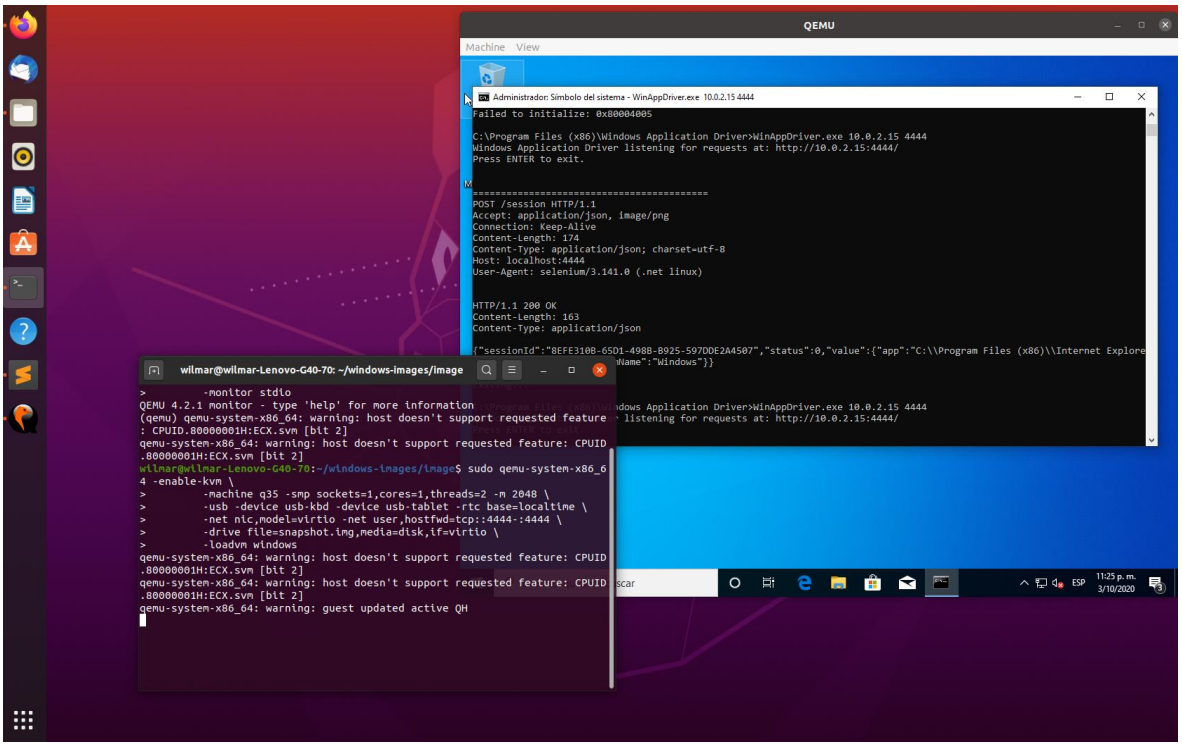


Figura 7. Ejecución de Snapshot tomado de la máquina virtual con Windriver activo y configurado para comunicarse con el puerto 4444. Elaboración propia

Una vez se verificó el correcto funcionamiento de la máquina virtual se llevaron a cabo pruebas de comunicación entre un código externo y el WinAppdriver. Para realizar las pruebas anteriormente mencionadas se usó el editor de texto Visual Studio Code y luego de ser configurado con las librerías y dependencias necesarias .NET para ejecutar código en C# se procedió a escribir un programa que abriera el navegador Internet Explorer y tomará un Screenshot cuando terminará el proceso como se muestra en la *Figura 8.* Esta prueba garantizaría que la máquina virtual ejecute el proceso correctamente cuando esta no se renderice en pantalla.

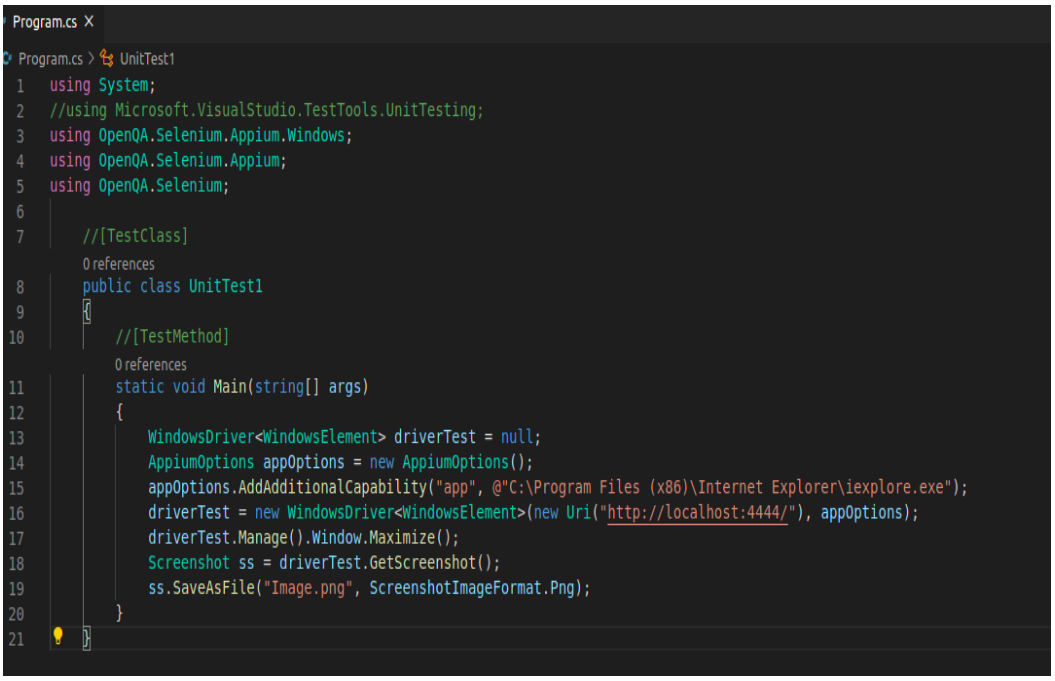


Figura 8. Código para Ejecución de Internet Explorer en la máquina virtual para tomar evidencia de la ejecución del proceso. Elaboración propia

2.1.4 Construcción de contenedor Docker basado en la virtualización anterior con una capa de red para comunicación externa entre el driver

Para hacer posible la ejecución de diferentes máquinas, se hace necesario ejecutar lo logrado en el paso anterior dentro de un contenedor Docker. Esto permitiría el envío de pruebas simultáneas hasta los contenedores ejecutándose en paralelo.

Ya teniendo el archivo hdd.img que contiene la máquina virtual de Windows y el Snapshot de los ajustes realizados se insertaron en un Folder como se muestra en la Figura 9 y posteriormente se procedió a crear el archivo Dockerfile para la creación del contenedor. El archivo Dockerfile permite enviar instrucciones para configurar un contenedor nuevo e instalar dependencias necesarias. En este archivo se especificaron, la versión de Ubuntu que se usó, las dependencias necesarias que se instalaron y el código que se ejecuta cada vez que el contenedor se inicia Figura 10.

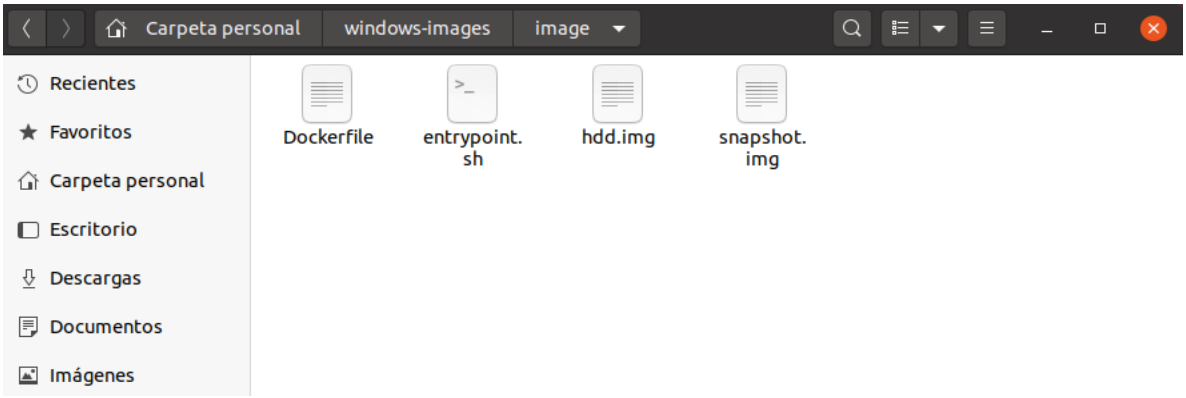


Figura 9. Archivos necesarios ubicados para la construcción del contenedor Docker. Elaboración propia.

Previamente a la creación del archivo Docker, se debe descargar el contenedor de Ubuntu. Es de suma importancia que la versión de Ubuntu en donde se creó el Snapshot coincida con el contenedor Docker de Ubuntu descargado, de lo contrario al ejecutar el contenedor en la máquina se generará un conflicto en las versiones.

Figura 10. DockerFile con instrucciones de instalación de dependencias necesarias y ejecución de código bash Elaboración propia.

En el archivo Dockerfile aparte del contenedor a usar como base, se especificaron las dependencias necesarias que se instalaron en el sistema en donde se realizó el proceso, sin embargo, esta vez se instaló la herramienta **XVFB (Virtual Frame Buffer),** la cual permite ejecutar cualquier programa en una capa oculta o virtualizada. Esto posibilita la ejecución de la máquina virtual, de manera que no sea necesario el uso de una pantalla o display. Lo antes mencionado fue especialmente útil, debido a la necesidad de Internet Explorer

de usar una salida de video. Con esto se hace posible la ejecución de la máquina virtual dentro de un contenedor Docker.

Cada vez que el contenedor Docker se iniciaba, era necesario que se ejecutara el comando para inicializar la máquina virtual de Windows ya configurada, por esta razón fue necesario agregar un script bash que se llamó “entrypoint” y se especificó en el archivo DockerFile mostrado en la *Figura 10*. Este comando describe el puerto que se usará para ejecutar la máquina virtual en segundo plano y su resolución, además contiene los comandos para inicializar la máquina virtual especificando el snapshot anteriormente creado, el archivo hdd.img y el puerto de red a usar, en este caso el puerto 4444 como se observa en la *Figura 11*.

El proceso de creación consiste en abrir la consola e ir al directorio que contiene los archivos necesarios mencionados anteriormente, específicamente, la dirección en donde se encuentra el archivo Dockerfile y todos los elementos que este requiera. Al ubicarse en este directorio se procedió a ejecutar el comando de construcción del contenedor que tomará el contenedor de Ubuntu 20.04; Este instalará y ejecutará los Scripts y dependencias descritas en la creación de archivo Docker *Figura 12.*

El proceso de creación del contenedor comienza preparando a Docker para la construcción del contenedor, al terminar esta preparación se ejecutaron automáticamente los comandos descritos en el DockerFile. En la *Figura 12* se puede observar que el proceso de instalación de dependencias no se lleva a cabo normalmente debido a que previamente se había construido un contenedor piloto que usaba las mismas dependencias a las descritas en el DockerFile, por esta razón, Docker usa las dependencias almacenadas en su cache en vez de descargarlas nuevamente.

Una vez se ejecutó el comando y finalizó el proceso se pudo observar la existencia de este al ejecutar el comando “Docker Images”, el cual muestra todos los contenedores existentes.

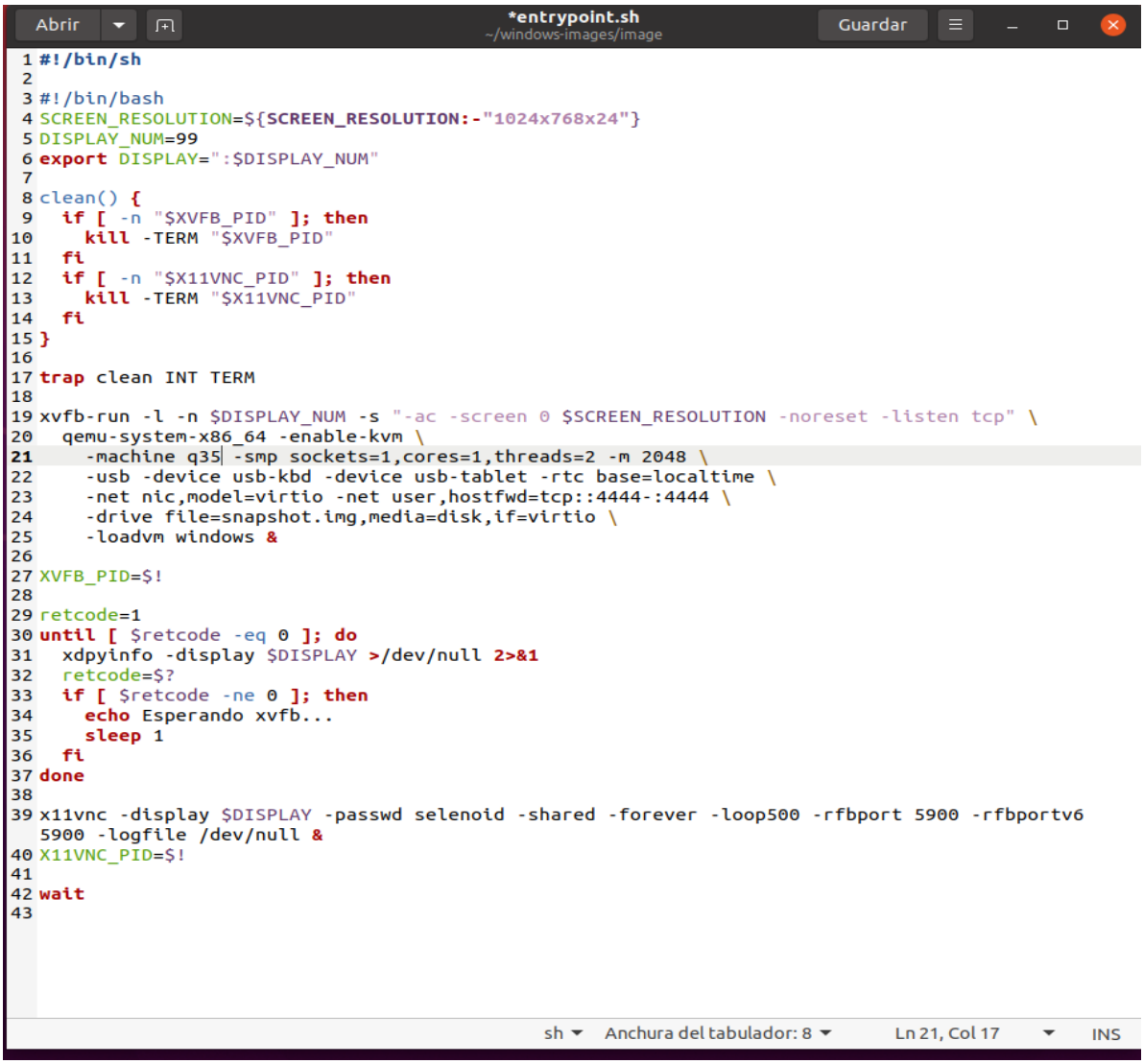


Figura 11. Contenido del script de activación de la máquina virtual. En él se especifican las características del display virtual a usar y el comando a ejecutar. Elaboración propia.

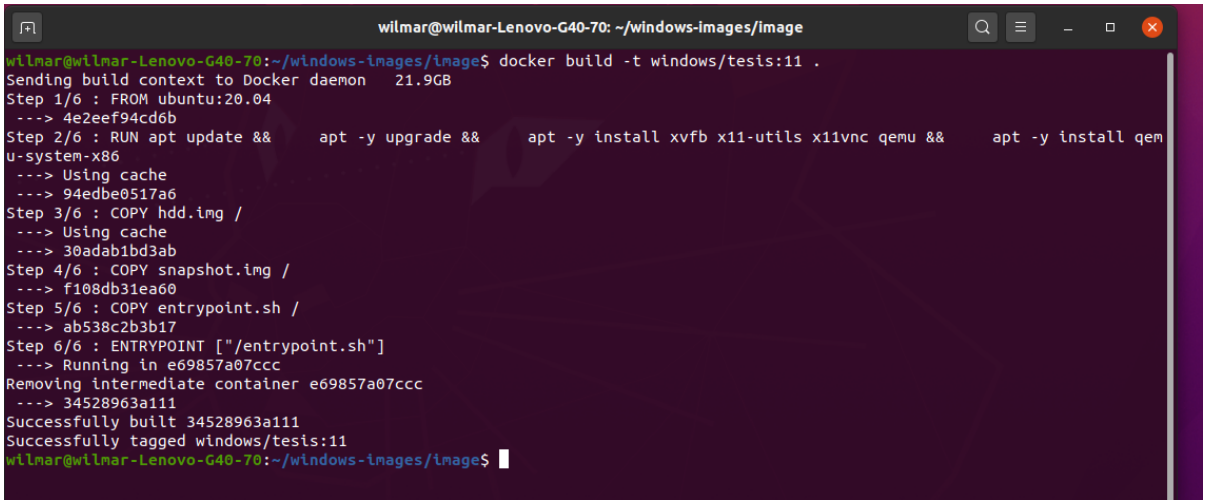


Figura 12. Comando y proceso de construcción del contenedor Docker. Elaboración propia.

Para ejecutar el contenedor se escribe el comando mostrado en la *Figura 13* en la terminal. Al ejecutarse, lo primero que muestra es la salida del Bash Script descrito anteriormente, el cual da inicio a la secuencia de ejecución de la máquina virtual. Visualmente no se podrá apreciar su ejecución, sin embargo, en una capa oculta virtualizada en Docker la máquina virtual ya estará ejecutada.

Para probar la conexión, se desarrolló un script pequeño en Visual Studio Code, el cual inicia Internet Explorer y ejecuta algunas tareas básicas usando Appium. Este programa se conecta a través de puerto 4444 al WinAppDriver, que se ejecutó en la máquina virtual de Windows, lo que da como resultado que dentro de la máquina virtual se inicie Internet Explorer y se ejecuten las tareas automatizadas por medio del Script. Como visualmente no tenemos acceso a la ejecución, una forma de poder evidenciar sus resultados es tomando Screenshot desde el código de Appium o acceder a la máquina virtual a través de VNC.

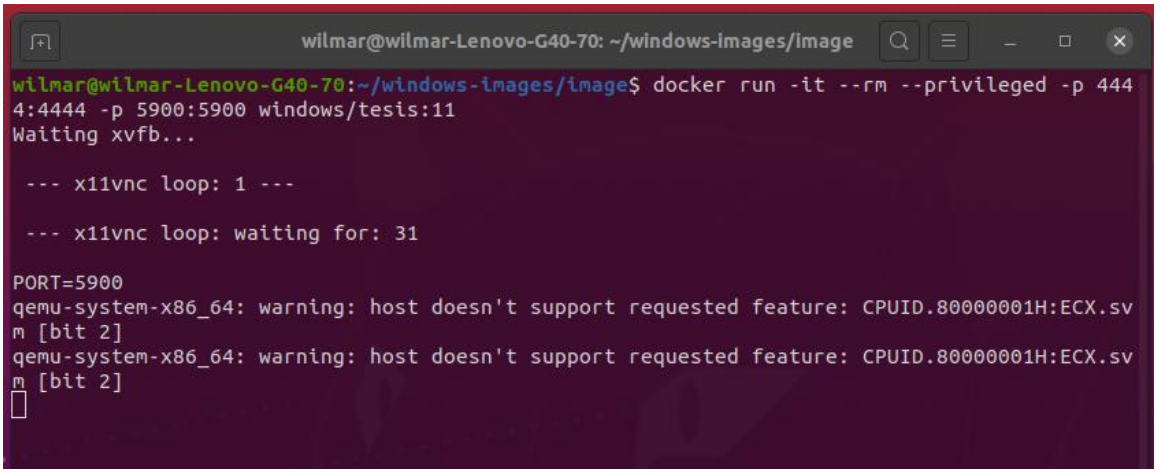


Figura 13. Comando y Ejecución de contenedor Docker especificando acceso al puerto 4444. Elaboración propia.

2.1.5 Prueba del contenedor con Test Completo en Programa de Consultoría de DigitalWare

Los Contenedores desarrollados en los pasos anteriormente descritos representan una gran ventaja con respecto al modo en que se ejecutan las pruebas debido a que no ocupan todo el equipo cuando estas se ejecutan, de modo que facilita la generación de reportes y productividad. Para demostrar esto, se realizó una prueba directamente con un programa de DigitalWare. El cual permite la búsqueda de registros, su edición y aplicación de cambios.

Para llevar a cabo dicha prueba se extendió el Test mostrado en la *Figura .8* en donde se extendieron los procedimientos llevados a cabo incluyendo diferentes pasos que engloban las acciones de Log In, selección de registro, edición de registro y Aplicación de Cambios en el registro. Estos cambios se muestran en la *Figura 14.*

```
1 reference | 1/1 passing
public static void LogIn(WindowsDriver<WindowsElement> desktopSession, string user, string password)
{

    string xpathFields = "//Window[contains(@ClassName,TSAcAccesFrm)]" +
        "[contains(@Name,Aseguramiento de Calidad)]/Pane[contains(@ClassName,TPanel)]" +
        "/Edit[contains(@ClassName,TEdit)]";
    var wait = new DefaultWait<WindowsDriver<WindowsElement>>(desktopSession)
    {
        Timeout = TimeSpan.FromSeconds(120),
        PollingInterval = TimeSpan.FromSeconds(1)
    };
    wait.IgnoreExceptionTypes(typeof(InvalidOperationException));

    ////////////////////////////////////////////
    WindowsDriver<WindowsElement> rootSession = null;
    rootSession = RootSession();
    //rootSession = ReloadXSession(rootSession, "TPanel");
    Thread.Sleep(3000);

    ReadOnlyCollection<WindowsElement> editFields = rootSession.FindElementsByClassName("TEdit");
    editFields[0].Clear();
    editFields[0].SendKeys(user);
    editFields[1].Clear();
    editFields[1].SendKeys(password);
    Thread.Sleep(2000);
    ReadOnlyCollection<WindowsElement> aceptar = rootSession.FindElementsByName("Aceptar");
    aceptar[0].Click();

}
```

Figura 14. Estructura de la función de Log In usada en el programa de consultoría. Elaboración propia.

```
1 reference | 1/1 passing
public static void QbeSeeker(WindowsDriver<WindowsElement> desktopSession) {

    string toolbar = "//Pane[contains(@ClassName,TPanel)]/ToolBar[contains(@ClassName,TToolBar)]";
    WebDriverWait bar;
    System.Collections.ObjectModel.ReadOnlyCollection<WindowsElement> bars;
    var wait = new DefaultWait<WindowsDriver<WindowsElement>>(desktopSession)
    {
        Timeout = TimeSpan.FromSeconds(120),
        PollingInterval = TimeSpan.FromSeconds(1)
    };
    wait.IgnoreExceptionTypes(typeof(InvalidOperationException));

    WindowsDriver<WindowsElement> rootSession = null;
    rootSession = RootSession();


    bar = new WebDriverWait(rootSession, TimeSpan.FromSeconds(10));
    bars = rootSession.FindElementsByName("tbrOpciones");
    //Debugger.Launch();
    bar.Until(res => bars[0].Displayed);

    rootSession.Mouse.MouseMove(bars[0].Coordinates, 50, bars[0].Size.Height/2);
    rootSession.Mouse.Click(null);
    Thread.Sleep(5000);
}
```

Figura 15. Estructura de la función de Qbe usada en el programa de consultoría. Elaboración propia.

Al ejecutar esta prueba apuntando al contenedor Docker esta inició de manera satisfactoria generando las capturas de pantalla de los pasos desarrollados. De este modo se confirma que los procesos y Test pueden ejecutarse dentro del entorno creado aún si el equipo en el que se ejecuta está siendo usado para el cumplimiento de otra tarea adicional ya que cuando la prueba se ejecuta en un entorno normal sin hacer uso de los contenedores, la prueba toma control total sobre el puntero del equipo, haciendo que este no esté disponible mientras la prueba se ejecute.

CAPÍTULO 3 *EXPERIENCIA DURANTE LA PASANTÍA*

Autor: Wilmar Daniel Bustos Morales
Director: Jose Daniel Ramírez Corzo 31

3.1 Pruebas automatizadas con coded UI Tests en programas del área Ophelia

En cuanto al desarrollo de la pasantía, esta dio inicio el 6 de julio de 2020 como practicante en la empresa de tecnología Digitalware, esta empresa se dedica a la creación de software empresarial personalizado, por lo tanto, es fácil suponer que cuenta con una gran extensión de programas desarrollados. Como practicante de Digitalware se desarrollaron diferentes actividades relacionadas con pruebas y test de los programas desarrollados del área de Ophelia de Digitalware. Original y, primeramente, las actividades consistieron en el desarrollo de pruebas automatizadas usando Coded UI test y C#. Estas pruebas consisten en la grabación de los controles de los programas siguiendo un checklist previamente definido por el área de pruebas, Este checklist asegura que se prueben las distintas funcionalidades que ofrecen los programas, desde agregar registros, generar reportes en Excel y PDF, entre otros.

Después de generar las grabaciones de cada control manualmente, se programaba un test en C# haciendo uso de estas grabaciones. Durante la programación de las pruebas era de vital importancia tener en cuenta qué podía salir mal en la prueba, por ende, se programaban respuestas para posibles errores que pudieran generarse al momento de ejecutar el test automatizado, de modo que su resultado fuera satisfactorio.

Para generar registros, es necesario insertar campos obligatorios con parámetros que en muchos campos dependen de cada programa. Estos parámetros se graban al momento de grabar la prueba con Coded UI Test sin embargo cuando es necesario crear registros con parámetros específicos esta no es una opción, por lo tanto, para poder tener flexibilidad con respecto a los datos ingresados en el programa, se usa una herramienta llamada Microsoft Team Manager(MTM), la cual permite crear casos de prueba con diferentes datos, estos casos se pueden asignar a los

test programados de modo que las pruebas se ejecuten haciendo uso de los parámetros especificados en el MTM

Una vez se terminaba de programar una prueba automatizada, esta se lanzaba localmente y se evaluaba su funcionamiento. Al finalizar, esta genera una evidencia en forma de documento Word, el cual contiene una serie de screenshots de todos los pasos ejecutados por las pruebas. Este proceso se repetía varias veces a la semana, y al final de semana se enviaban las pruebas al servidor usando los servicios de Microsoft Azure, los cuales permiten ejecutar las pruebas en grandes cantidades en un servidor.

3.2 Pruebas web automatizadas NTC

A finales de noviembre mis actividades como pasante cambiaron a la automatización de pruebas en el área web. Los programas en esta área era completamente web por lo cual el framework usado era distinto. En este sector se usó el framework de automatización Selenium implementado en C#.

Las pruebas NTC se basaban en comprobar los campos de entrada existentes en las páginas web, su funcionamiento y el nombre que cada uno llevaba, para llevar a cabo esta tarea se usaba un marco de pruebas, que consistía en comparar los nombres de los campos de programa con nombres creados en diferentes casos usando el programa MTM y si estos nombres no coincidían, se generara un mensaje de error que indicara en qué campo había un error. Cada campo se grababa usando el xpath respectivo y extrayendo la información de este haciendo uso de Selenium

En esta área fue donde pude hacer mi primera contribución notable con respecto a la refactorización del código usado. La estructura presentaba bastante repeticiones debido a que el mismo proceso debía ser repetido para cada campo, junto a esto, originalmente el xpath grabado debía ser modificado para que las funciones de Selenium pudieran reconocerlas, por estas razones el desarrollo de estas pruebas podía resultar repetitivo y tedioso, además, editar el xpath manualmente podía generar errores humanos, por lo tanto se decidió refactorizar el código editando los xpath automáticamente usando edición de strings y guardando estos xpath editados en listas. Una vez en listas, fue sencillo poder editar el código de manera que todo el proceso pudiera realizarse de manera iterativa usando ciclos.

De esta manera esta refactorización contribuyó a la reducción de líneas de código y a la disminución de la interacción del programador con los elementos necesarios al momento de escribir el código.

3.*3* Desarrollo del nuevo framework de pruebas usando Appium

Para mejorar el desempeño de las pruebas automatizadas para los programas Ophelia, el área de pruebas propuso un cambio en la herramienta de pruebas que se estaba usando hasta el momento, El área decidió que la creación de una nueva herramienta personalizada para la construcción de las pruebas personalizadas sería la forma más efectiva de mejorar su rendimiento y buen desempeño. por lo tanto, a finales de octubre se decidió dar comienzo al desarrollo de la herramienta de pruebas basada en Appium, cada miembro de área de pruebas asumió responsabilidad sobre un pasó del checklist. En mi caso me correspondió el desarrollo de la función para detectar las notas y realizar el proceso correspondiente a esta.

Mientras cada miembro desarrollaba su parte, se encontraron diferentes problemáticas a superar, relacionadas con la imposibilidad del grabador de reconocer exactamente algunos controles de los programas, Uno de estos controles que no se reconocía con exactitud eran las notas y esto me llevo a efectuar mi segunda contribución al área de pruebas.

Appium se estaba usando de manera similar a Selenium, a través de reconocimiento de xpath para la detección e interacción de controles, sin embargo, muchos controles no se reconocían de manera exacta y esto representaba un problema que podía resultar en un incremento en el tiempo de ejecución de las pruebas e incluso, la creación de código confuso y poco eficiente. Para solucionar esta problemática realicé mi segunda contribución basándome en el procesamiento de imágenes, debido a que realizando un procesamiento de representaciones pictóricas en el programa presentado, podía detectar ciertos controles basándome en

sus colores y esto me permitió crear una función que devolviera sus coordenadas con exactitud, permitiendo así contribuir a la solución, no solo de mi función sino de otras funciones que presentaban un problema con respecto a la detección de Appium.

CAPÍTULO 3
RESULTADOS

Autor: Wilmar Daniel Bustos Morales
Director: Jose Daniel Ramírez Corzo 36

ANÁLISIS DE RESULTADOS

Originalmente las pruebas automatizadas se ejecutan de manera secuencial en un computador que funciona como servidor. Esto es una limitante que causa que la ejecución de estas pruebas no se pueda llevar a cabo simultáneamente. De manera similar la naturaleza del único navegador en el que estas pruebas se pueden ejecutar no contribuye a la posibilidad de ejecutarlas al mismo tiempo en una misma pantalla u ocultar su ejecución de manera sencilla.

Inicialmente se contempló la posibilidad de desarrollar esta funcionalidad a través de contenedores Docker directamente desde Windows, sin embargo, Internet Explorer, aunque ejecuta dentro del contenedor, genera errores debido a que este navegador requiere que se renderice la parte grafica para su correcta ejecución. Para comprender mejor el comportamiento descrito anteriormente se realizaron pruebas de manera similar usando dos navegadores distintos, Google Chrome e Mozilla Firefox, en donde la apertura y ejecución de pruebas en estos dos navegadores fue satisfactoria. Esto es debido a que estos dos navegadores soportan una característica denominada “Headless mode”, la cual permite la ejecución del navegador y ejecución de procesos automatizados en segundo plano. Internet Explorer no posee soporte para esta característica por lo cual el enfoque originalmente planteado no solucionaría el problema presentado en la ejecución de pruebas con Internet Explorer.

después de concluir que el enfoque de un contenedor directamente ejecutado desde Windows no solucionaría la problemática planteada se decidió implementar XVFB para la renderización de procesos en segundo plano. Debido a esto, la implementación de este método se vio limitada a Ubuntu.

Debido a que la máquina virtual que ejecuta el programa de la empresa se separa de los procesos realizados en la maquina Host a través de la contenerización con Docker, La prueba se ejecuta en un entorno totalmente aislado, lo cual permite una mayor flexibilidad en cuanto a la ejecución de las pruebas. Desde que todo el proceso se encuentra gestionado por Docker es posible la ejecución de diversos contenedores que puedan ejecutar pruebas paralelas teniendo en cuenta las limitaciones de la maquina Host junto a una definición de puertos específica para cada contenedor.

Aunque el desempeño de los contenedores usados no difiere del presentado en las pruebas ejecutadas directamente en la maquina host, el entorno creado requiere de un tiempo corto para la inicialización de los

contenedores antes de ser utilizables por la prueba. El tiempo de carga de esta ronda entre los 10 minutos y 15 minutos, sin embargo, este es un proceso que se lleva a cabo solo una vez, después de inicializado puede usarse para la ejecución de pruebas consecutivas sin volver a pasar por esta carga inicial.

Para la ejecución de los contenedores es necesario que el entorno Host en donde se ejecute sea una distribución de Linux Física, debido a que fácilmente permiten la implementación de la virtualización anidada, sin embargo, no se descarta la posibilidad de la implementación de los contenedores en un sistema Windows o haciendo uso de una virtualización anidada más profunda.

En cuanto a la eficiencia del método descrito anteriormente con respecto al modo en que se ejecutaban las pruebas automatizadas en DigitalWare la mejora depende directamente de la cantidad de contenedores que ejecute el computador Host. Debido a que esta mejora está relacionada con la capacidad de los contenedores de ejecutarse en Paralelo de modo que solo sea necesario el cambio de los puertos de cada uno de los contenedores ejecutados en el mismo Host. Cabe aclarar que la capacidad de un Host de levantar diversos contenedores depende del Hardware que este posea haciendo que no todas las computadoras puedan ejecutar más de un contenedor a la vez y tener un resultado óptimo. El diagrama de la Figura 16 ilustra el modo en que varios contenedores se comunican con las pruebas automatizadas. Al programar una prueba se especifica el puerto con el cual se va a comunicar con el WinAppDriver. Cada una de los contenedores habilita un puerto distinto para evitar conflicto entre las pruebas y a su vez; dentro de cada contenedor, la máquina virtual se encuentra activa ejecutando el WinAppDriver al puerto indicado en la Figura.

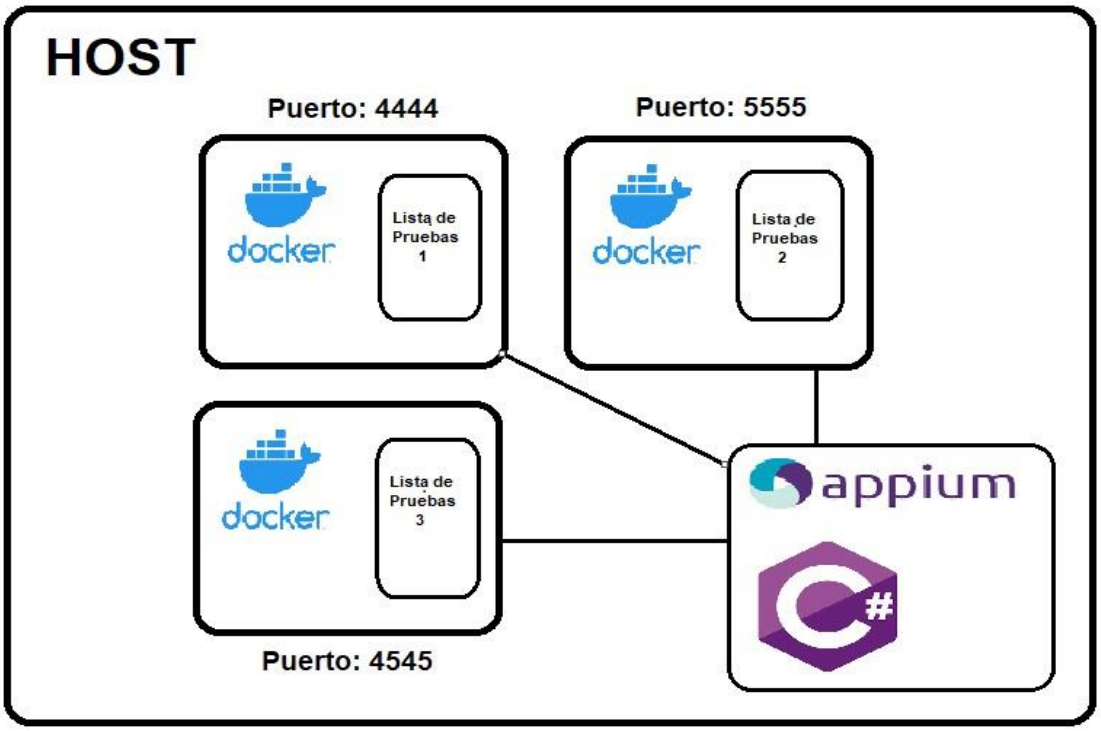
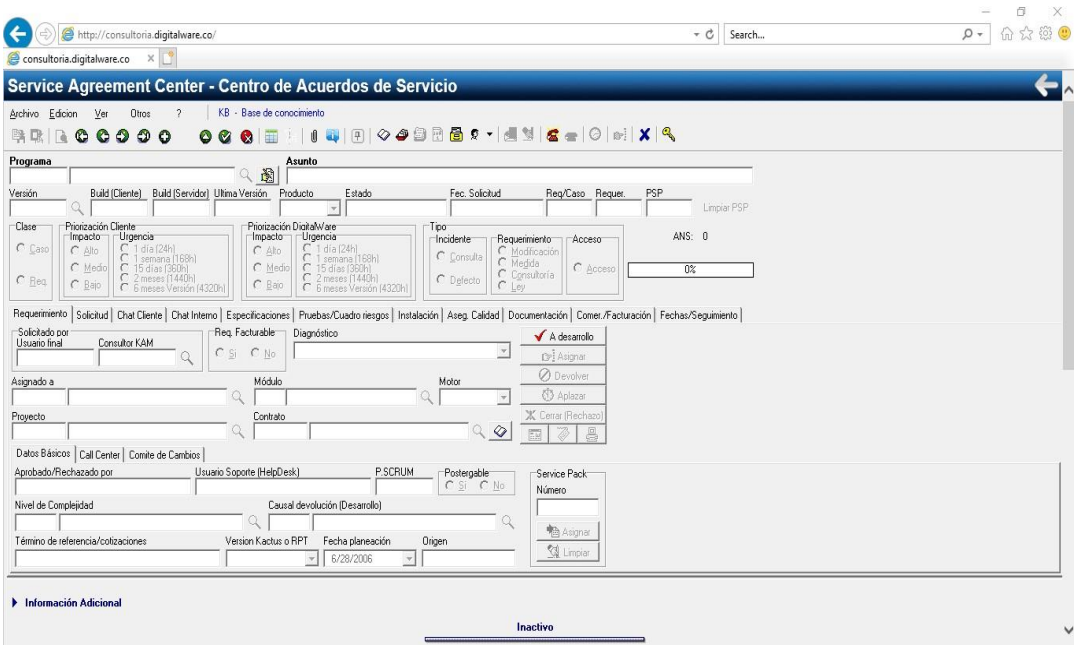


Figura 16. Diagrama de comunicación entre la prueba automatizada y los contenedores docker
Elaboración propia.

Anteriormente, al ejecutar una prueba automatizada en el servidor, esta tomaba control completo sobre el equipo permitiendo que estas se ejecutarán solo de manera secuencial, sin embargo, con la implementación del procedimiento descrito esta situación cambia permitiendo la ejecución de grupos de pruebas secuenciales de forma simultánea lo cual es posible gracias a la renderización que se lleva a cabo dentro del contenedor y no directamente en el Host. La comparación de la ejecución de una prueba se aprecia en la Figura 17, en donde en la f*igura 17 a)* la prueba toma la pantalla completa del host, mientras que en la *Figura 17 b)* la única salida visible mientras se ejecuta la prueba son los mensajes de la terminal indicando que la máquina virtual dentro del contenedor ya se encuentra activa.

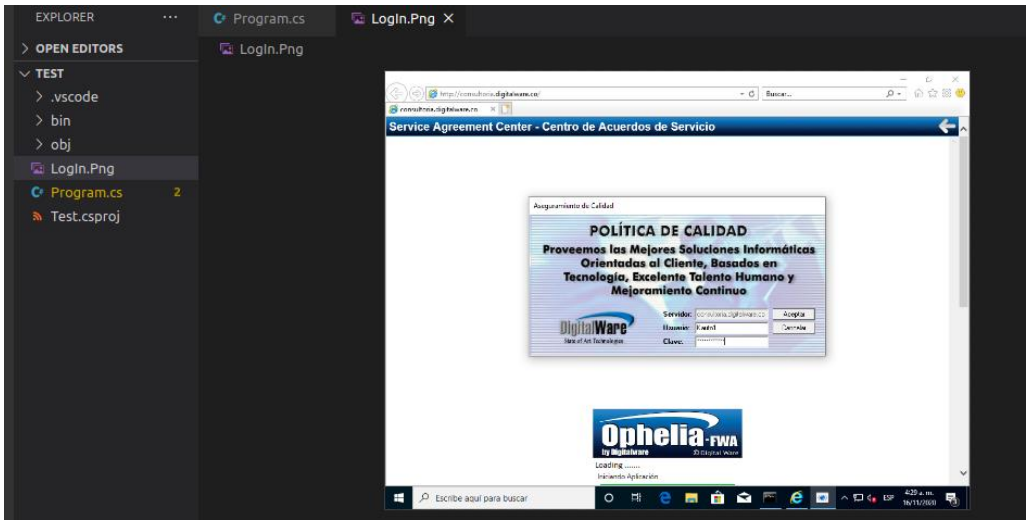


a).

b).

Figura 17). a) Ejecución de prueba convencional en el Host. *b)* Despliegue de contenedor y ejecución de prueba. Elaboración propia.

A medida que la prueba se ejecutó en el contenedor, los Screenshots programados en esta se guardan; en el directorio en donde se desarrolló la prueba automatizada, en este caso, en el Host como evidencia de su ejecución. En la F*igura 18 a)* se muestra el explorador de archivos con la evidencia de inicio de sesión ya ejecutado dentro del contenedor y en la Figura 18 b) se muestra el código para generar el Screenshot en dicha ruta.



a).

```
Screenshot image = ((ITakesScreenshot)rootSession).GetScreenshot();
image.SaveAsFile(string.Format("LogIn.Png", ScreenshotImageFormat.Png));

Thread.Sleep(2000);
System.Collections.ObjectModel.ReadOnlyCollection<WindowsElement> aceptar =
aceptar[0].Click();
Thread.Sleep(10000);
```

b).

Figura 18). a) Evidencia del proceso de inicio de sesión ejecutado dentro del contenedor. *b)* Código para generar la Evidencia antes de completar el inició de sesión. Elaboración propia.

En el caso de la computadora en la que se desarrollaron las pruebas se limitó a ejecutarse un contenedor a la vez, cuando se intentó ejecutar otro contenedor los procesos se volvieron lentos y casi inejecutables. Por lo tanto, a mayor cantidad de contenedores ejecutándose en paralelo, mayor capacidad de procesamiento debe tener la computadora Host. En base a esto se puede concluir que el costo para mejorar la velocidad de ejecución de las listas de pruebas Ophelia en DigitalWare se reduce a la inversión en un solo Host con suficiente potencia de procesamiento para ejecutar diferentes Host de manera paralela en la figura 19 se observan las ventajas de la implementación de el servidor desarrollado con respecto a servidor empleado.

Servidor basado en virtualización y contenedores Docker	Servidor basado en ejecución secuencial y renderizado en el host
• Permite la paralelización de ejecución de pruebas	• No permite paralelización debido a la renderización de Internet Explorer en el Host directamente
• El tiempo de ejecución de los paquetes depende directamente de numero de contenedores que se estén usando.	• El tiempo de ejecución de los paquetes depende directamente de la duración de cada prueba individual.
• Permite el manejo del Host mientras se ejecutan las pruebas dentro de los contenedores.	• El Host no puede manipularse mientras las pruebas se están ejecutando ya que afecta directamente al desempeño de la prueba.
• El costo de aumentar la velocidad de producción de reportes se reduce a el mantenimiento y mejora de un solo Host	• El costo de aumentar la velocidad de producción de reportes depende del número de Host que DigitalWare posea.

Tabla1). Tabla comparativa entre el servidor desarrollado y el servidor usado en DigitalWare. Elaboración propia.

CONCLUSIONES

A través de la tesis anteriormente expuesta en donde se presentó el desarrollo e implementación de un servidor de pruebas usando herramientas de virtualización y contenedores basados en sistemas Linux se puede concluir que haciendo uso concepto de virtualización anidada es posible la construcción de contenedores Docker que permitan la ejecución de las pruebas automatizadas desarrolladas en la empresa Digitalware, partiendo de la necesidad de que Internet Explorer requiere de la renderización de su interfaz al contrario que otros navegadores más recientes como lo son Google Chrome y Firefox.

La aplicación de este método para la ejecución de las pruebas realizadas en el área de pruebas de los programas Ophelia no solo permitiría su uso como servidor de pruebas, su implementación también se puede expandir a los computadores en donde se ejecutan las pruebas locales. De este modo, la ejecución de pruebas locales obtendría un espacio aislado para su ejecución reduciendo interrupciones accidentales o por el bloqueo de pantalla mientras se ejecutan las pruebas.

El funcionamiento de los contenedores implementados en este proyecto se limita a ser ejecutados en distribuciones Linux, sin embargo, no se descarta la posibilidad de construirse y ejecutarse directamente en Windows. Como se menciona en este trabajo, la condición más importante para el funcionamiento de los contenedores Docker creados es que el Host en donde se ejecute permita la habilitación de virtualización anidada.

En base a lo descrito al finalizar los resultados, el costo por incluir capacidades de mejora con respecto a velocidad y escalamiento de las pruebas. Se reduce a la inversión de DigitalWare en un Host con las características necesarias para permitir la ejecución de más de dos contenedores en paralelo, necesitando así, un solo servidor.

REFERENCIAS

[1] D.Morris, S Voutsnas, N.C Hambly, R.G.Mann. Use of Docker for deployment and testing of astronomy software. En: Astronomy and Computing. Vol.; 20 No(July, 2017); p. 105-119

[2] CHAE M., LEE H., KIYEOL K., A performance comparison of linux containers and virtual machines using Docker and KVM En: Cluster Comput Vol.;22 No(2019); p.1765-1775.

[3] FINK. John. Docker: a software as a service, operating system-level virtualization framework. En: Code4Lib J. 25, 31 (2014)

[4] Syrewicze A., Siddaway R. Containers. In: Pro Microsoft Hyper-V 2019. Apress, Berkeley, CA. (2018)

[5] A. M. Potdar, D. Narayan, S. Kengond, M. M. Mulla. Performance Evaluation of Docker Container and Virtual Machine. En: Procedia Computer Science. Vol.; 171 No(2020); p. 1419-1428

[6] HIGGINGS J. Holmes V. Venters C. Orchestraring Docker Containers in the HPC Enviroment. En: High Performance Computing. Lecture Notes in Computer Science, vol.; 9137 .p 506-513

[7] BROCH Johnsen, STEFFEN Martin, Stumpf Johanna. Virtually timed ambients: A calculus of nested virtualization. En: Journal of Logical and Algebraic Methods in Programming. Vol.;94 (Jan-2018); p. 109-127

[8]Jinbao Ren, Yong Qi, Yuehua Dai, Yu Xuan, Yi Shi. Nosv: A lightweight nested-virtualization VMM for hosting high performance computing on cloud. En: The Journal of Systems and Software. Vol.;124(Feb, 2017); p.137-152

[9] X. Wan, X. Guan, T. Wang, G. Bai, B. Choi. Application deployment using Microservice and Docker containers: Framework and optimization. En: Journal of Network and Computer Applications Vol.; 119(2018); p.97-109.

[10] A. Martin, S. Raponi, T. Combe, R. Di Pietro. Docker ecosystem – Vulnerability Analysis. En: Computer Communications. Vol.; 122(2018); p.30-43.

[11]. Qemu, "*GUÍA DOCUMENTADA PARA UBUNTU*" Lugar de publicación: https://www.guia-ubuntu.com/index.php/Qemu

[12]. "*¿Que es KVM Kernel Virtual Machine?*, "HostDime" Lugar de publicación: https://www.hostdime.com.pe/blog/que-es-kvm-kernel-virtual-machine/

[13] Hypervisor, "*VMware*" Lugar de publicación: https://www.vmware.com/topics/glossary/content/hypervisor#:~:text=A%20hypervisor%2C%20also%20known%20as,such%20as%20memory%20and%20processing.

[14]. What Is Hyper-V Nested Virtualization?, "*datto*" Lugar de publicación: https://www.datto.com/library/what-is-hyper-v-nested-virtualization#:~:text=Nested%20virtualization%20refers%20to%20running,machines%20as%20resources%20are%20available.

[15]. What is Software Testing? Definition, Basics & Types, "*guru99*" Lugar de publicación:https://www.guru99.com/software-testing-introduction-importance.html

[16] Windows Application Driver, "*Github*" Lugar de publication: https://github.com/microsoft/WinAppDriver

[17] Host, "*TechTerms*" Lugar de publicación: https://techterms.com/definition/host#:~:text=A%20host%20is%20a%20computer,other%20computers%20to%20access%20it.&text=A%20host%20can%20access%20its,using%20the%20hostname%20%22localhost.%22

[18] Dockerfile reference, "*docker docs*" Lugar de publication: https://docs.docker.com/engine/reference/builder/