UNIVERSIDAD DE GRANADA

Sistemas Críticos

Contenedores para sistemas críticos

Víctor Vázquez Rodríguez
victorvazrod@correo.ugr.es

Máster Universitario en Ingeniería Informática

Curso 2019/20

Índice

1. Introducción

Hoy en día, la Industria 4.0 es uno de los principales campos de trabajo e investigación tecnológica. Podríamos decir, de forma muy simplificada, que el objetivo es automatizar por completo los procesos productivos para aumentar el rendimiento y la calidad de los productos y servicios, así como reducir los costes. Para que estos procesos sean autosuficientes y autogestionados, es necesario aplicar técnicas de inteligencia artificial al control de las instalaciones físicas, entrenando modelos de toma de decisiones con los datos obtenidos de sensores situados en las instalaciones. Debido a que se trata de grandes cantidades de datos y que estas técnicas suelen requerir de una alta capacidad de computación, no sería extraño pensar en el *cloud* como la plataforma ideal. No obstante, el transporte de todos estos datos de forma constante a través de la red, en algunos casos incluso proveniendo de múltiples fábricas o instalaciones, pone una gran carga sobre la infraestructura de red y congestiona el tráfico, además de que estas comunicaciones suponen una latencia inevitable que en muchos casos es también inaceptable.

La solución más aceptada consiste en llevar todos estos nuevos procesos a las propias fábricas e instalaciones, surgiendo nuevos modelos de computación como *edge* y *fog computing*. Estas nuevas arquitecturas plantean que los sistemas de control críticos, con restricciones de tiempo real, convivan con los procesos de inteligencia artificial y otro software no crítico, compartiendo incluso el mismo hardware. Es esencial que estos sistemas de criticalidad mixta aseguren que los requisitos de tiempo del software crítico se cumplan.

Se han realizado algunos experimentos y demostraciones de ejecución de estos sistemas de criticalidad mixta sobre un mismo microprocesador haciendo uso de hipervisores avanzados, que permiten lanzar máquinas virtuales sobre el hardware asignando los recursos en función de los requisitos de cada una. Sin embargo, el modelo planteado por el *fog computing* supone un balanceo de la carga entre todos los nodos contectados a la red, lo que nos hace plantearnos el uso de contenedores como técnica de virtualización capaz de permitir esta replicación y distribución eficiente de procesos.

La idea principal detrás de los contenedores es empaquetar cada aplicación con todo lo que ésta necesita para funcionar correctamente en un contenedor aislado del resto de procesos. De esta forma, distintos contenedores comparten el mismo kernel para las llamadas de bajo nivel (en contraposición con las máquinas virtuales, en las que cada una tiene un sistema operativo completo y comparten solo el hardware) y no pueden afectar al funcionamiento del sistema fuera de su propio contenedor, aumentando la seguridad e integridad del mismo.

Debido a esto, con este proyecto se ha querido comprobar el determinismo de los procesos contenerizados y ejecutados sobre Docker, que es el principal motor de contenedores actualmente. Se ha prestado atención, sobre todo, a los tiempos de lanzamiento, pausa y otros mecanismos de Docker para el control de los procesos. En las siguientes secciones de este documento, se explican en detalle las pruebas que se han llevado a cabo y se presentan sus resultados.

2. Pruebas realizadas

Como ya se ha indicado en la sección anterior, las pruebas que he llevado a cabo se centran en determinar los tiempos relativos a los procesos contenerizados y ejecutados sobre Docker. Las pruebas se han realizado sobre los dos siguientes dispositivos:

- Raspberry Pi 4 con procesador ARM Cortex A72 @ 1.5 GHz y 4 GB de RAM ejecutando una versión de Raspbian Lite modificada para la ejecución de contenedores con Docker.
- Portátil con procesador Intel Atom N270 @ 1.6 GHz y 1 GB de RAM ejecutando una instalación mínima de Debian.

De esta forma, obtenemos resultados para dos plataformas diferentes (armv7 e i386), lo que nos permite comparar su rendimiento. Los sistemas operativos de ambos dispositvos son de 32 bits [1].

Para no afectar al rendimiento de los dispositivos y, por tanto, a las pruebas, la ejecución de éstas se lleva a cabo desde un ordenador distinto y conectado a los dispostivos de prueba mediante una conexión física vía Ethernet. Es en este ordenador donde se ejecutan los *scripts* de Python encargados de conectarse al dispositivo de prueba vía SSH y ejecutar todo lo necesario para las pruebas. Gracias a estos *scripts*, la ejecución de las pruebas está automatizada en su totalidad, aunque es necesario conectarse a los dispositivos previamente para construir las imágenes Docker. Se han utilizado las librerías de Python de *pandas* (creación de *data frames* y almacenamiento como CSV) y *paramiko* (ejecución de comandos vía SSH). El código completo puede encontrarse en el repositorio de GitHub creado para el proyecto [2].

Además de usar plataformas distintas, también he llevado a cabo la implementación de los procesos contenerizados que se prueban en distintos lenguajes: C, Go y Rust. Todos ellos son lenguajes compilados y orientados a la programación de sistemas. Cabe destacar que para los contenedores de Go se ha usado como imagen base Alpine Linux, que es una distribución mínima, en contraposición con la imagen base de Debian para los contenedores de C y Rust debido a la inexistencia de una imagen base de Alpine para estos lenguajes.

Las pruebas realizadas se han agrupado en dos conjuntos según los tiempos que están diseñadas para medir: lanzamiento de los contenedores y respuesta de los mismos. En las siguientes subsecciones, se va a detallar en qué consiste cada prueba y cómo se ha llevado a cabo.

2.1. Tiempo de lanzamiento

El motor de contenedores Docker incorpora una funcionalidad para inspeccionar los contenedores que se han ejecutado o están en ejecución. Esta información se obtiene con el comando `docker inspect` y devuelve, entre otras cosas,

[1] Aunque el procesador de la Raspberry Pi 4 soporta la ejecución en 64 bits, el sistema operativo Raspbian solo está disponible en 32 bits.

[2] Repositorio del proyecto: https://github.com/Varrrro/container-metrics

las marcas de tiempo del inicio del contenedor, el inicio del proceso que contiene y su final. Usando estas marcas de tiempo, se puede calcular el tiempo de lanzamiento del contenedor (desde que se lanza hasta que comienza el proceso que contiene) y el tiempo de ejecución del proceso contenido, que son los datos deseados.

Para la realización de estas pruebas, se ha implementado un proceso muy sencillo en los 3 lenguajes ya indicados que simplemente envía un paquete UDP de vuelta al ordenador desde el que se ejecutan las pruebas. El proceso de prueba definido en el *script* queda entonces así:

1. Lanzar el contenedor.
2. Esperar a recibir el paquete UDP que envía el contenedor, indicando que ha terminado su ejecución.
3. Ejecutar `docker inspect` para obtener las marcas de tiempo.
4. Calcular los tiempos de lanzamiento y ejecución.
5. Eliminar el contenedor para poder lanzarlo de nuevo.

Este proceso se realiza un número concreto de veces para cada implementación del contenedor de prueba. Este número de iteraciones se especifica como argumento en la ejecución del *script*. Al terminar la ejecución de la batería de pruebas, los resultados recopilados se guardan en un fichero CSV.

2.2. Tiempo de respuesta

En este caso, lo que se quiere comprobar es el tiempo de respuesta de un proceso contenerizado partiendo desde distintos estados en los que se pueden encontrar los contenedores Docker: en ejecución, pausados y parados. El primero de estos estados es obvio, se trata de un proceso corriendo de forma ininterrumpida. Por otra parte, al pausar y parar un contenedor, lo que hace Docker es enviar una señal SIGSTOP o SIGKILL al proceso, respectivamente. Aunque el contenedor como tal sigue existiendo, su proceso no está en ejecución y deja de consumir tantos recursos.

Para hacer estas pruebas, se ha implementado un sencillo servidor UDP que responde al remitente y que actuará como proceso cuyo tiempo de respuesta se mide. Al igual que con las otras pruebas, este proceso se ha implementado en cada uno de los lenguajes propuestos. Además, como se tiene que alterar el estado de este contenedor, es necesario un proceso "maestro"que se ejecute en el dispositivo de prueba y que se encargue de ello. Debido a la naturaleza de las pruebas, se han implementado tres versiones de este proceso maestro, que son las siguientes:

- Lanza el proceso de prueba y lo mantiene funcionando.
- Mantiene el proceso de prueba parado y lo lanza de nuevo para cada petición recibida.

- Mantiene el proceso de prueba pausado y lo reanuda para cada petición recibida.

Además de controlar el estado del contenedor de prueba, este proceso maestro también actúa como *proxy*, redirigiendo los paquetes recibidos al contenedor y las respuestas de éste, al remitente. Con todo esto, el proceso de prueba definido en el *script* queda así:

1. Enviar un paquete UDP al maestro.
2. En caso de ser la versión con pausa o parada, el maestro relanza o reanuda el contenedor de prueba.
3. El maestro reenvía el paquete UDP al contenedor.
4. El maestro espera hasta recibir la respuesta del contenedor.
5. El maestro reenvía la respuesta al ordenador desde el que se ejecutan las pruebas para que calcule el tiempo de respuesta.
6. En caso de ser la versión con pausa o parada, el maestro pausa o para el contenedor.

Al igual que con las pruebas del tiempo de lanzamiento, todo este proceso se repite un número indicado de veces para cada combinación de implementación del contenedor de prueba y versión del proceso maestro.

Cabe destacar que, como se puede apreciar viendo el proceso de prueba definido, no se tiene en cuenta el tiempo que se tarda en parar o pausar el contenedor. Esto se debe a que, en un hipotético sistema de control distribuido, no nos afecta el tiempo de parada, sino que lo que nos interesa es ver el tiempo que tarda en responder desde un estado de "hibernación".

3. Resultados

He realizado las pruebas definidas en la sección anterior con 1000 iteraciones y he pasado los datos por dos *scripts* de R para analizar los datos y obtener representaciones gráficas de los mismos.

En las siguientes subsecciones se van a presentar los resultados obtenidos de las 2 pruebas y se van a discutir algunas de sus implicaciones.

3.1. Tiempo de lanzamiento

Primero, vamos a ver los resultados de las pruebas de tiempo de lanzamiento. Estos resultados los encontramos en el cuadro 1. Podemos apreciar como el tiempo medio de lanzamiento es notablemente superior en i386, aunque la desviación típica es algo inferior en esta plataforma, indicando que los tiempos son más consistentes.

Cuadro 1: Resultados de las pruebas de tiempo de lanzamiento en milisegundos

Plataforma	**Impl.**	**Media**	**Desviación**	**Min**	**Max**
arm32v7	C	1693	165	1472	2548
arm32v7	Go	1693	163	1445	2718
arm32v7	Rust	1701	156	1478	2378
i386	C	2606	159	1918	3496
i386	Go	2615	156	2010	3488
i386	Rust	2608	148	1961	3530

Otro aspecto que se aprecia es que no hay diferencias realmente notables entre las implementaciones en distintos lenguajes, obteniendo todos más o menos los mismos resultados para una misma plataforma. Esto se puede apreciar aún mejor en las gráficas de las Figuras 1 y 2. Comparando estas dos gráficas, se puede comprobar que los tiempos en i386, aunque son más altos, son más estables que los registrados en arm32v7, tal y como se ha indicado antes.

Para terminar con los resultados de estas pruebas, vemos en la Figura 3 una gráfica comparativa de los tiempos medios. Confirmamos lo que ya se veía en el cuadro anterior: los tiempos son más bajos en arm32v7 (en torno a 1.7 segundos frente a los 2.6 de i386) y no hay una diferencia notable entre los lenguajes usados para implementar el contenedor de prueba.

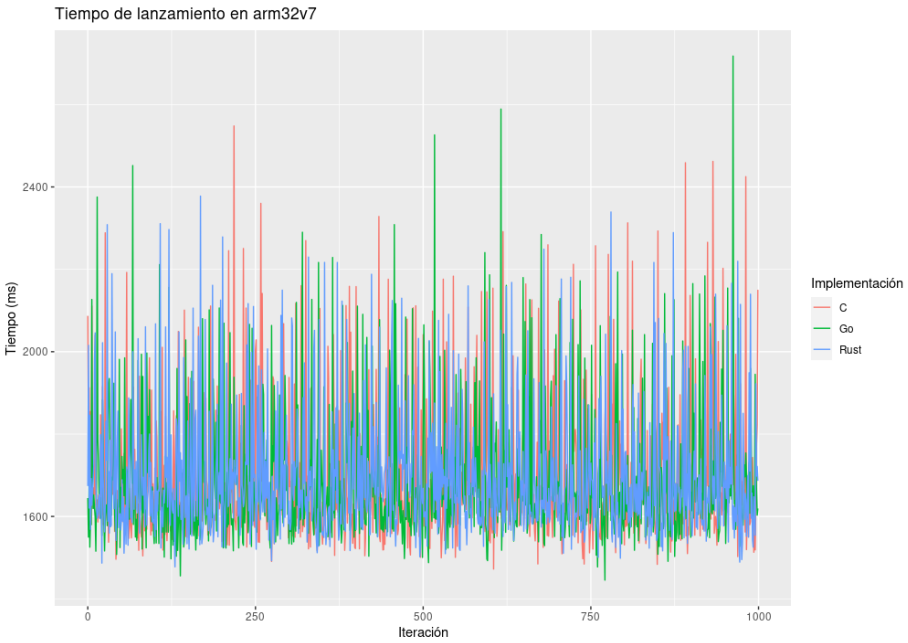

Figura 1: Tiempos de lanzamiento en armv7

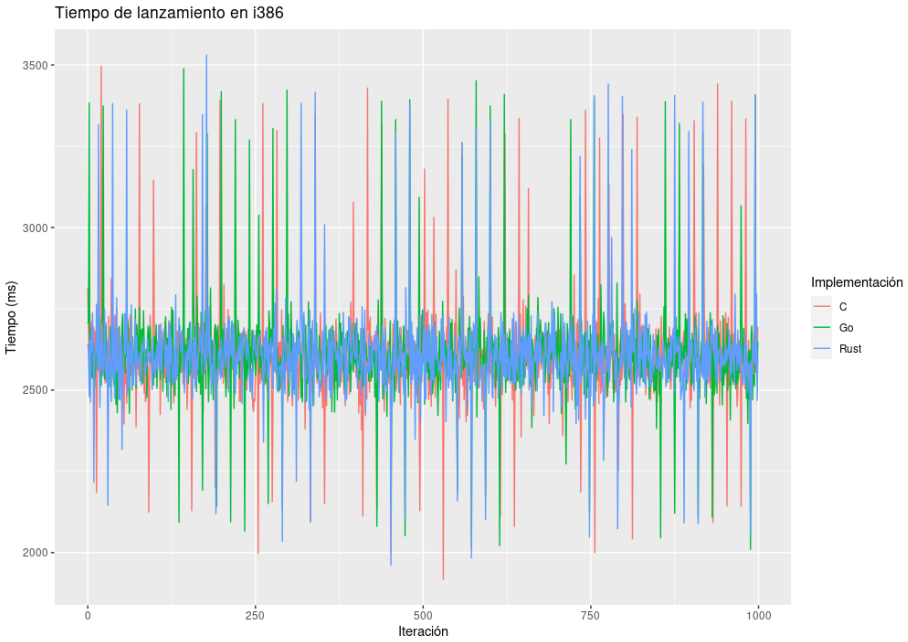

Figura 2: Tiempos de lanzamiento en i386

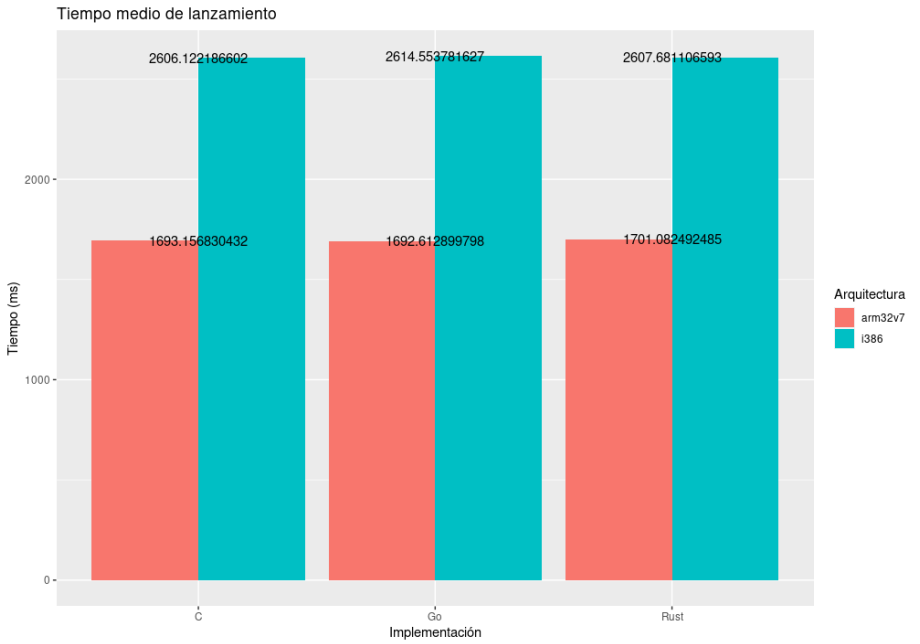

Figura 3: Tiempos de lanzamiento medios en ambas plataformas

3.2. Tiempo de respuesta

En el cuadro 2 encontramos los resultados que se han obtenido para las pruebas del tiempo de respuesta. Al haber tantas combinaciones posibles de plataforma, versión del proceso maestro y lenguaje de implementación del contenedor de prueba, se van a ir explicando y analizando los resultados con varias gráficas para que sea más sencilla su comprensión.

Vamos a comenzar con la plataforma arm32v7. En la Figura 4 encontramos los tiempos de respuesta para las 1000 peticiones cuando el contenedor se encontraba constantemente en ejecución. Hay pocas diferencias de tiempo entre los lenguajes usados, aunque estas diferencias son más notables cuando observamos los picos. C es el lenguaje que ha dado tiempos más constantes, mientras que Rust ha sido el más inestable y con los picos más altos. En la Figura 5 tenemos los tiempos para el contenedor parado que, como era de esperar, son los más altos de todas las pruebas con los distintos estados de partida para esta plataforma. Una vez más, no tenemos diferencias entre los tiempos de respuesta de los distintos lenguajes, teniendo también un comportamiento similar en cuanto a la consistencia en los tiempos de respuesta. Sí que destacamos que los tiempos son notablemente más inconsistentes a los obtenidos para el contenedor en ejecución, obviamente debido a la interacción con el motor de Docker para el lanzamiento del proceso parado.

Cuadro 2: Resultados de las pruebas de tiempo de respuesta en milisegundos

Plataforma	**Versión**	**Impl.**	**Media**	**Desv.**	**Min**	**Max**
arm32v7	En ejecución	C	35.9	3.42	34.2	112
arm32v7	En ejecución	Go	36.3	4.84	33.9	123
arm32v7	En ejecución	Rust	37.4	7.54	34.0	140
arm32v7	Parado	C	1330	95.3	1164	1942
arm32v7	Parado	Go	1322	112	1150	1963
arm32v7	Parado	Rust	1336	108	1170	1927
arm32v7	Pausado	C	88.5	22.4	74.7	306
arm32v7	Pausado	Go	87.0	19.5	74.6	292
arm32v7	Pausado	Rust	85.6	17.0	74.7	188
i386	En ejecución	C	1.60	0.961	1.16	28.2
i386	En ejecución	Go	2.13	0.518	1.40	5.83
i386	En ejecución	Rust	1.57	0.162	1.24	2.79
i386	Parado	C	2505	41.5	2402	3185
i386	Parado	Go	2507	34.9	2391	2624
i386	Parado	Rust	2511	37.1	2390	2622
i386	Pausado	C	164	7.79	149	304
i386	Pausado	Go	164	6.31	152	199
i386	Pausado	Rust	164	6.29	144	237

Terminando el análisis de resultados de arm32v7, tenemos los tiempos de respuesta con el contenedor pausado en la gráfica de la Figura 6. Aunque los tiempos son mayores que los obtenidos con el contenedor en ejecución, no se trata de valores excesivamente altos (alrededor de 50 milisegundos más, de media). Igual que con las pruebas para los otros dos estados, no se aprecian diferencias notables entre los lenguajes usados.

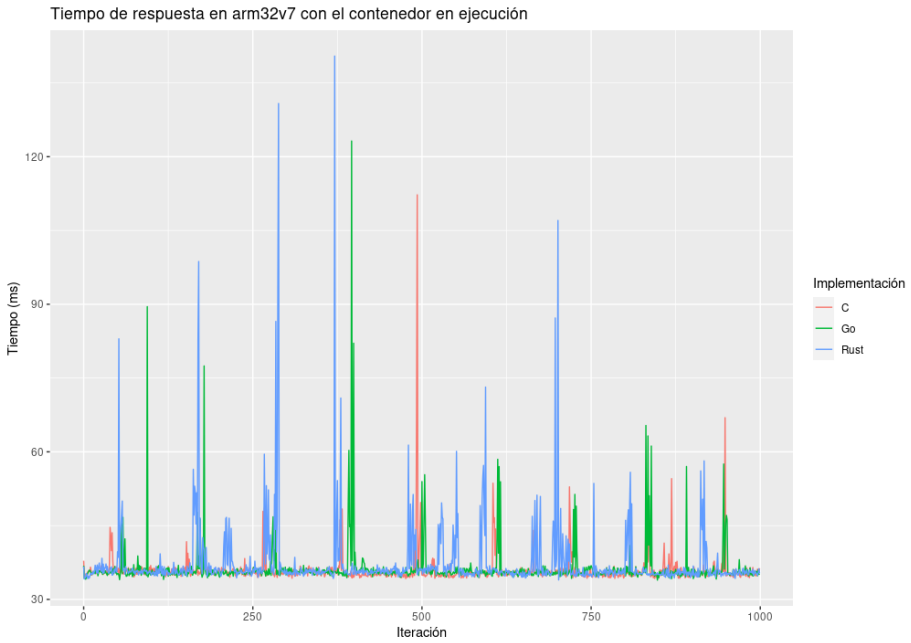

Figura 4: Tiempos de respuesta en armv7 en ejecución

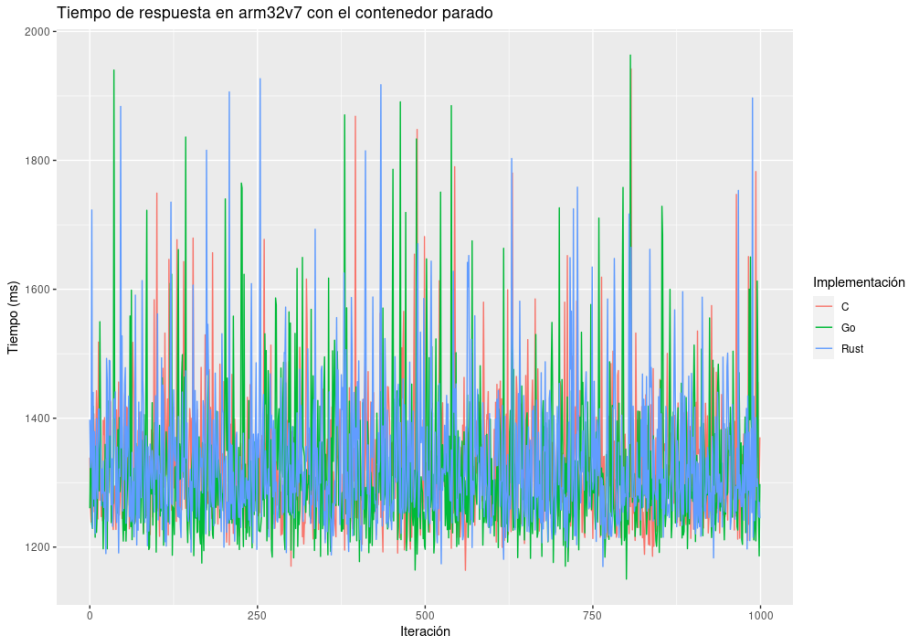

Figura 5: Tiempos de respuesta en armv7 partiendo de parada

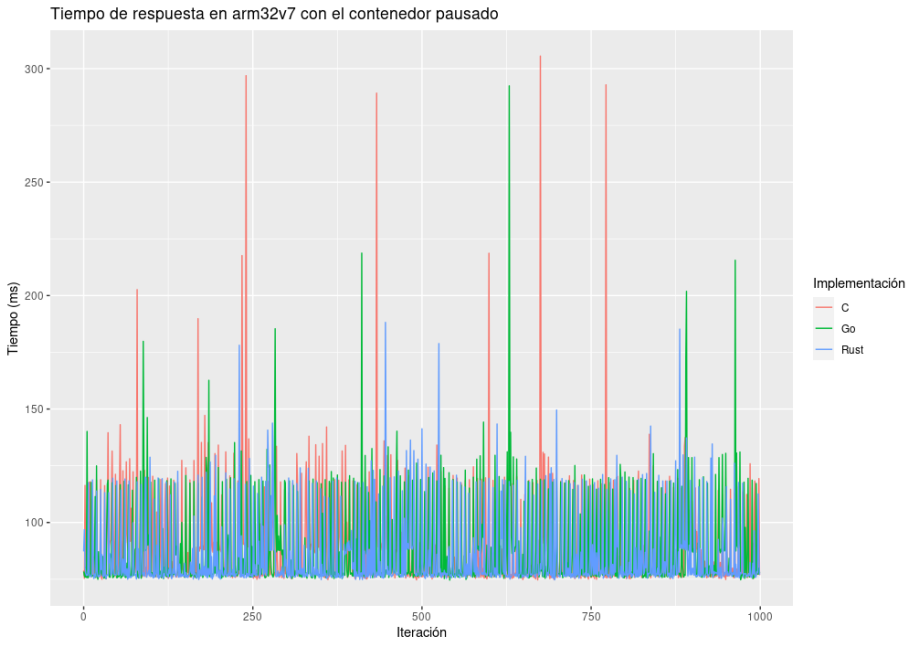

Figura 6: Tiempos de respuesta en armv7 partiendo de pausa

Ahora, comenzamos a ver los resultados para i386, empezando por el tiempo de respuesta con el contenedor en ejecución. La gráfica que muestra estos datos es la de la Figura 7. Lo primero que destacamos es que los tiempos son mucho más bajos y consistentes que los obtenidos en las mismas pruebas para arm32v7, aunque se hará una comparación más detallada entre las dos plataformas más adelante. Siguiendo con la tendencia de todas las demás pruebas, no hay diferencias notables entre los lenguajes usados, aunque el contenedor implementado en C es el que ha reportado más picos. Atribuir estos picos al lenguaje sería, sin embargo, precipitado, ya que solo se producen 3 y en solo en las primeras 500 peticiones, así que puede deberse a otras causas. En las pruebas para el contenedor parado y pausado, cuyos tiempos se pueden ver en las gráficas de las Figuras 8 y 9 respectivamente, estos tiempos son peores que los obtenidos para arm32v7, pero siguen siendo más consistentes. Existen algunos picos puntuales, aunque destacamos que Go es el único que no ha presentado picos notables en ninguno de los dos casos.

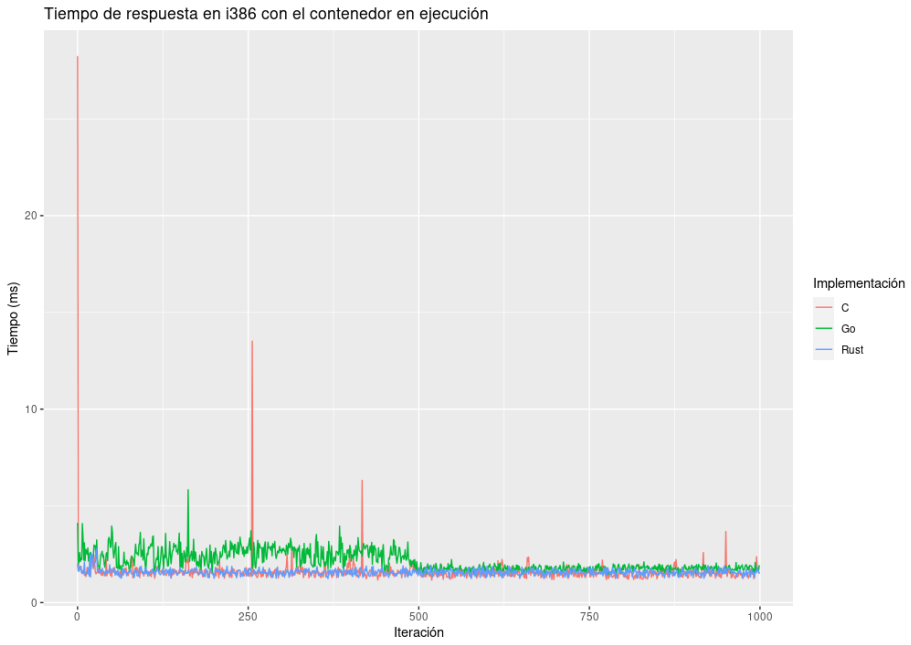

Figura 7: Tiempos de respuesta en i386 en ejecución

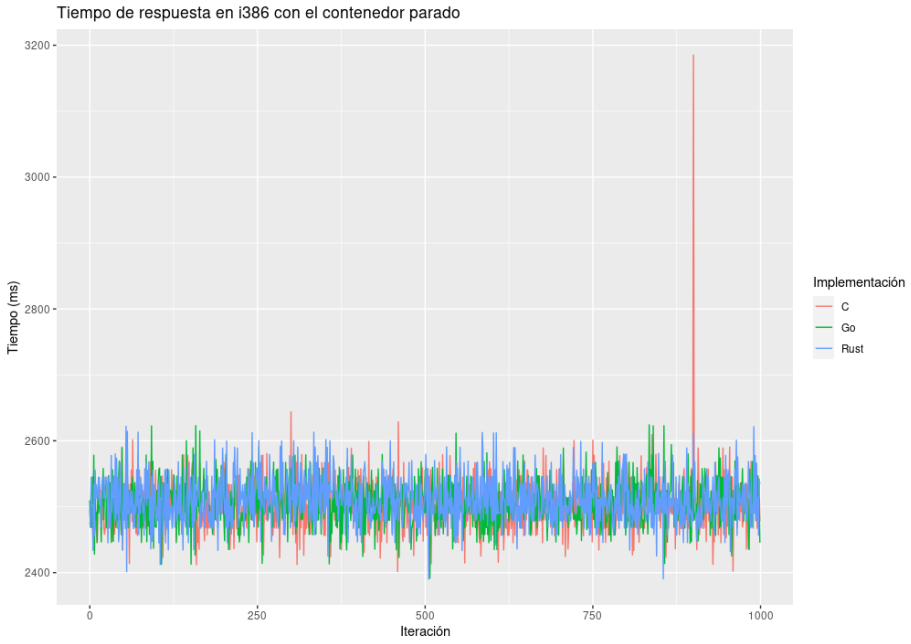

Figura 8: Tiempos de respuesta en i386 partiendo de parada

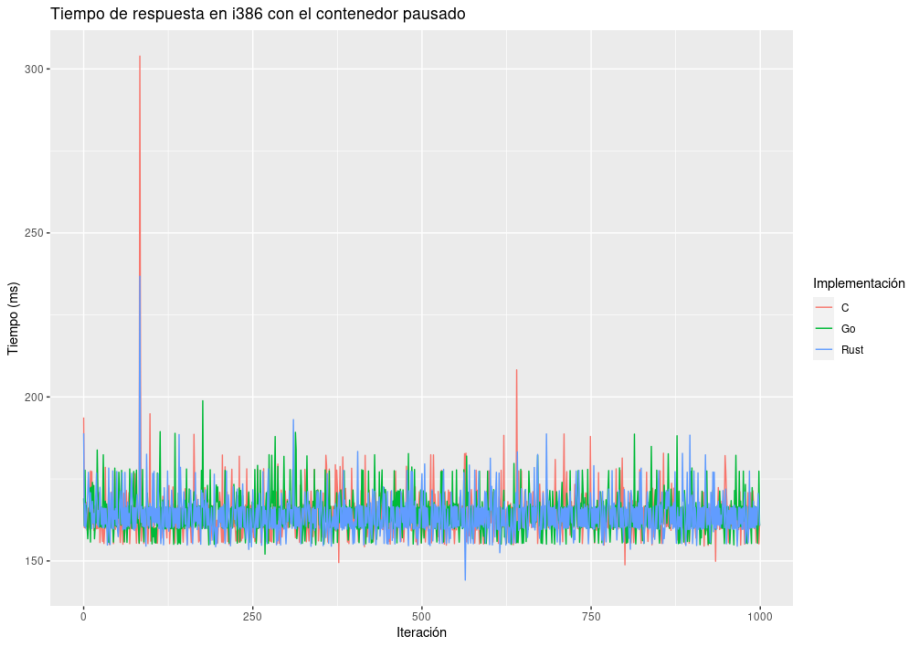

Figura 9: Tiempos de respuesta en i386 partiendo de pausa

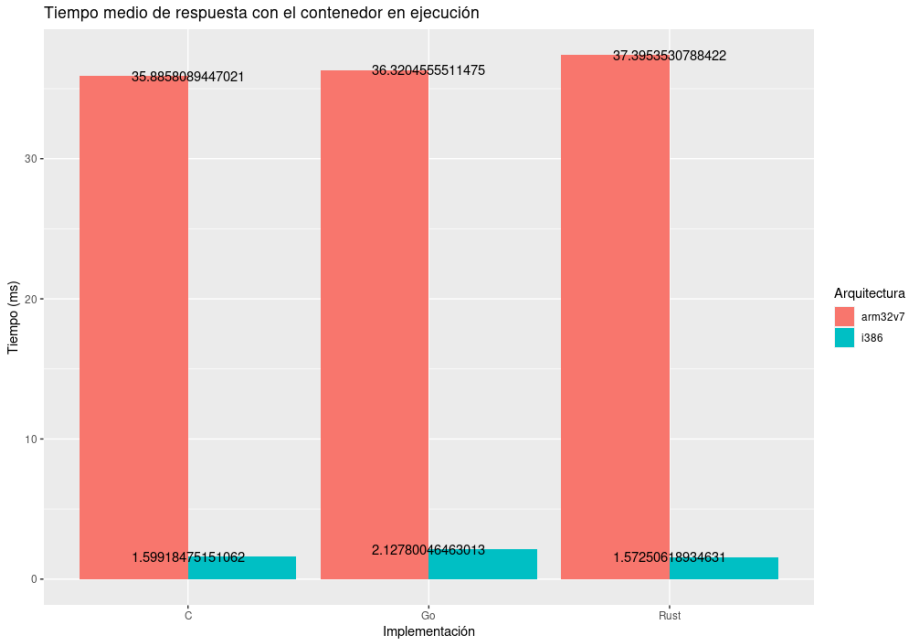

Figura 10: Tiempos medios de respuesta en ejecución para ambas plataformas

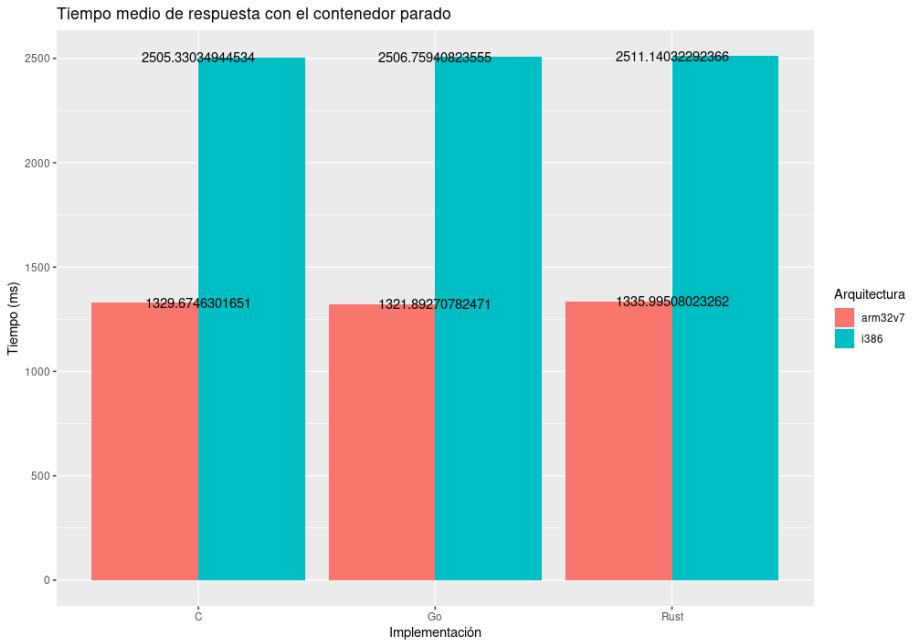

Figura 11: Tiempos medios de respuesta partiendo de parada para ambas plataformas

Para terminar, vamos a comparar los resultados en estas pruebas para ambas plataformas. En la Figura 10 vemos los tiempos medios de respuesta teniendo el contenedor en ejecución. La diferencia entre ambas plataformas es muy grande, estando el tiempo de arm32v7 en torno a los 36 o 37 milisegundos respecto a los 2 milisegundos de media que presenta i386, haciendo de este el único caso en el que i386 es más rápido que arm32v7. En arm32v7, además, tenemos diferencias apreciables entre las implementaciones, siendo Rust el lenguaje más lento y C el más rápido. En i386, Rust y C están a la par, mientras que es Go el más lento.

Teniendo el contenedor parado, vemos en la Figura 11 que en arm32v7 los tiempos de respuesta han sido, de media, casi la mitad de los obtenidos en i386 (en torno a 1330 milisegundos frente a 2505). En este caso, las diferencias en el tiempo son inexistentes para los distintos lenguajes. La situación es prácticamente idéntica cuando se tiene el contenedor pausado, obteniendo arm32v7 unos tiempos de respuesta medios que son la mitad de los de i386, además de ser estos tiempos casi idénticos para los 3 lenguajes usados. Los tiempos de respuesta medios para el contenedor pausado se pueden ver en el gráfico de la Figura 12.

Por último, en la Figura 13 tenemos una comparativa de los tiempos medios de respuesta para las tres situaciones y en ambas plataformas. Esta gráfica refuerza todo lo que hemos ido indicando a lo largo de esta subsección.

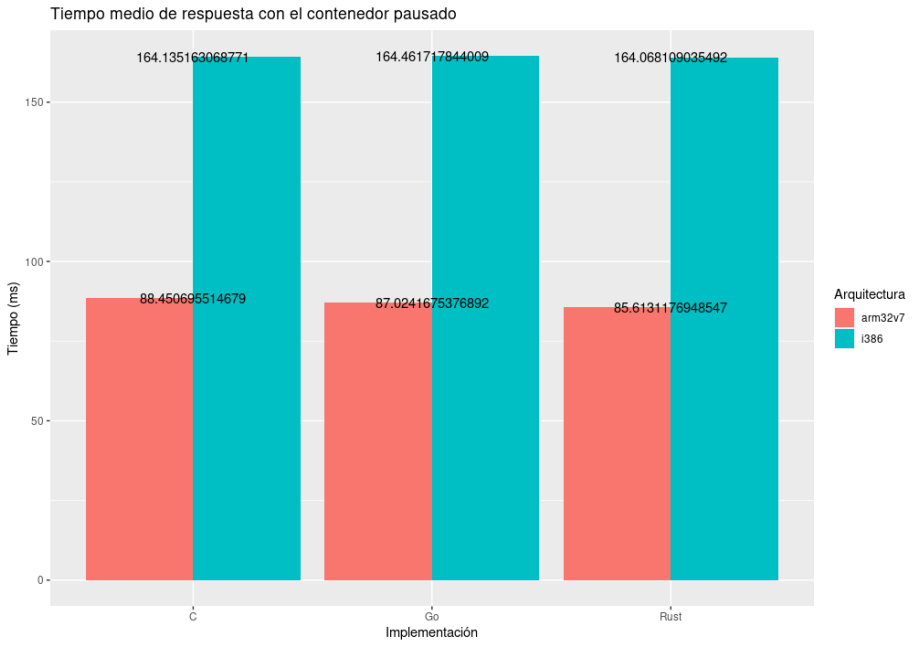

Figura 12: Tiempos medios de respuesta partiendo de pausa para ambas plataformas

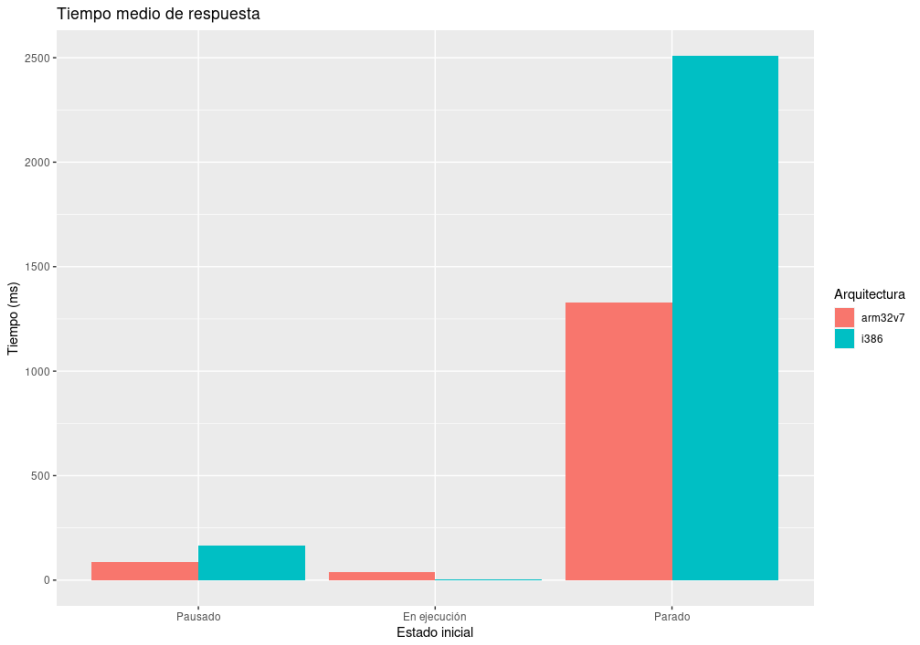

Figura 13: Tiempos medios de respuesta para ambas plataformas

4. Conclusiones

En este proyecto se han realizado muchas pruebas y se ha obtenido una gran cantidad de datos, de los cuáles se pueden extraer algunas conclusiones. La primera que salta a la vista viendo los tiempos de respuesta de los contenedores en ejecución, es que las aplicaciones contenerizadas pueden ser aptas para restricciones de tiempo real. En esta línea, si volvemos a observar los datos del cuadro 2, la desviación típica en los tiempos en i386 es considerablemente inferior a la que se aprecia en arm32v7 en cualquiera de las situaciones (en ejecución, parado o pausado). En los tiempos de lanzamiento también se aprecia ésto, aunque la diferencia de desviaciones es menor, tal y como se ve en el cuadro 1. Podríamos concluir entonces que i386 ofrece un comportamiento más determinista que arm32v7, al menos en los dispositivos usados.

Por otra parte, considero que los tiempos de lanzamiento que se han registrado son demasiado elevados para sistemas con restricciones de tiempo real "duras", aunque sí podrían ser aptos para los que tengan restricciones "blandas". El principal problema lo encontramos en la poca consistencia de tiempos en todas las tareas de gestión de los contenedores, tal y como se aprecia en los resultados de los tiempos de respuesta para los contenedores parados y pausados y de los tiempos de lanzamiento.

En definitiva, considero que los resultados son esperanzadores para la aplicación de las tecnologías de contenerización a la automatización de la industria,

aunque creo que sería necesario construir herramientas específicas para este caso de uso, como pueden ser motores y orquestadores de contenedores que tengan en cuenta y garanticen los requisitos de tiempo de cada aplicación contenerizada. Además, la plataforma de ARM necesita de mejoras en el campo del trabajo con contenedores para poder ser una opción fiable.