		Escuela Politécnica Superior Ingeniería Informática Prácticas de Sistemas Informáticos 2			
Grupo	2311	Práctica	2	Fecha	27/03/2025
Alumno/a		Gómez, Hernández, Marco			
Alumno/a		Haya, de la Vega, Juan			

Práctica 2: Uso de Jmeter, encontrar la saturación del servidor y pruebas de carga

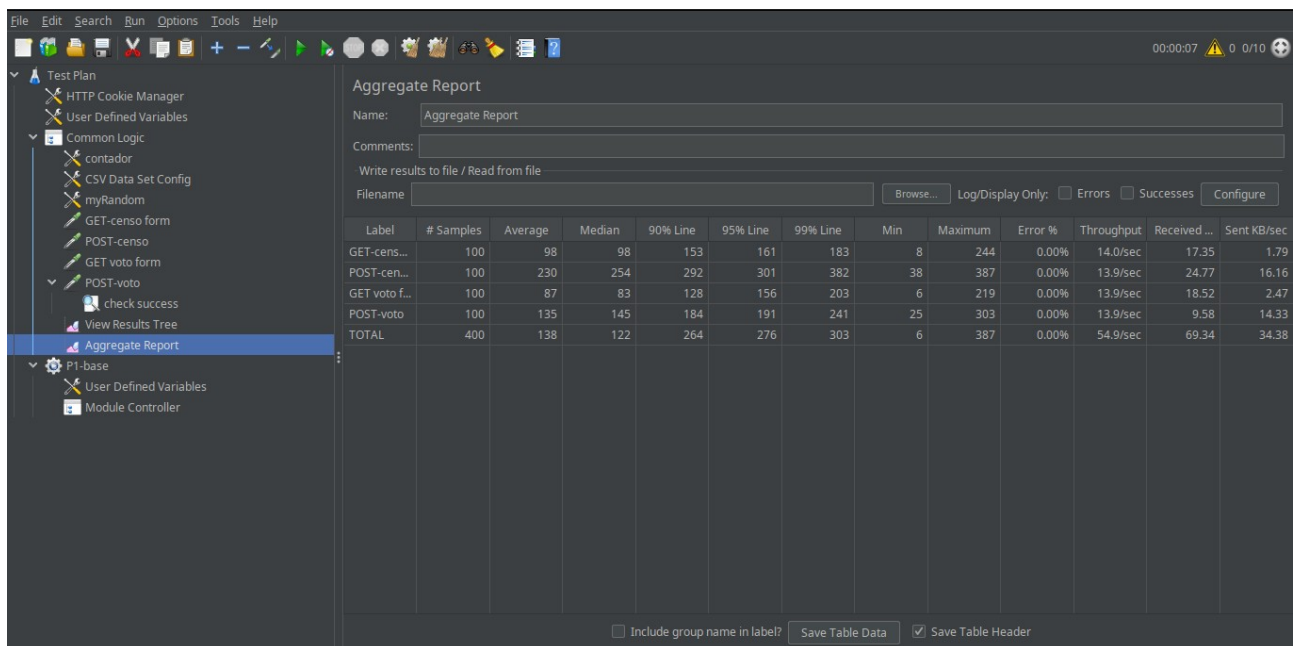
Ejercicio número 1:

Siguiendo todos los pasos anteriores, defina el plan completo de pruebas para el proyecto P1-Base. Guarda este plan en un fichero llamado P2_P1-base.jmx y entrégalo con la práctica. Adjunta en la memoria una captura del resultado de ejecutar el Aggregate Report y comenta el significado de cada columna.

Primero, creamos el plan de pruebas como se nos dice (adjunto en la entrega con el nombre que se pide) (se puede ver la estructura en la captura siguiente).

Segundo, configuramos Unicorn en la VM2 para que use el proyecto P1-base, borramos y volvemos a crear la base de datos en la VM1, hacemos el migrate y el collectstatic, y, por último, el populate (la base de datos debe estar limpia y sin votos para ejecutar Jmeter).

Ahora, procedemos a ejecutar el plan de pruebas



The screenshot shows the JMeter interface with the 'Aggregate Report' window open. The report displays the following data:

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received ...	Sent KB/sec
GET-cens...	100	98	98	153	161	183	8	244	0.00%	14.0/sec	17.35	1.79
POST-cen...	100	230	254	292	301	382	38	387	0.00%	13.9/sec	24.77	16.16
GET voto f...	100	87	83	128	156	203	6	219	0.00%	13.9/sec	18.52	2.47
POST-voto	100	135	145	184	191	241	25	303	0.00%	13.9/sec	9.58	14.33
TOTAL	400	138	122	264	276	303	6	387	0.00%	54.9/sec	69.34	34.38

En cuanto al significado de cada columna:

La columna **Label** representa el nombre de cada petición HTTP que se está evaluando. En este caso, se incluyen diferentes solicitudes como GET y POST para distintas acciones dentro de la prueba. La fila **TOTAL** representa la combinación de todas las solicitudes ejecutadas (para mostrar la combinación de cada una de las otras columnas).

La columna **# Samples** indica la cantidad total de muestras o peticiones realizadas para cada tipo de

solicitud. Un número mayor de muestras permite obtener resultados más representativos del rendimiento de la aplicación.

La columna **Average** muestra el tiempo de respuesta promedio en milisegundos para cada solicitud. Se calcula sumando todos los tiempos de respuesta y dividiéndolos entre el número total de muestras.

La columna **Median** representa la mediana del tiempo de respuesta, es decir, el valor central cuando todos los tiempos se ordenan de menor a mayor. Este dato es útil porque no se ve afectado por valores atípicos extremos.

Las columnas **90% Line**, **95% Line** y **99% Line** muestran los percentiles 90, 95 y 99 del tiempo de respuesta, respectivamente. Esto significa que el 90%, 95% o 99% de las solicitudes tuvieron un tiempo de respuesta igual o inferior a los valores mostrados en estas columnas. Son métricas clave para entender la estabilidad del sistema.

Las columnas **Min** y **Maximum** indican el tiempo de respuesta mínimo y máximo registrado en cada solicitud. Son útiles para detectar si hay tiempos de respuesta extremadamente bajos o altos en comparación con la media.

La columna **Error %** indica el porcentaje de errores ocurridos en las solicitudes. Un valor de 0% significa que todas las peticiones fueron exitosas, mientras que un porcentaje mayor indicaría fallos en algunas respuestas.

La columna **Throughput** mide el número de peticiones procesadas por segundo. Un throughput más alto indica que el sistema es capaz de manejar más solicitudes en el mismo tiempo.

Las columnas **Received KB/sec** y **Sent KB/sec** muestran la cantidad de datos recibidos y enviados por segundo en kilobytes. Estas métricas ayudan a analizar el consumo de ancho de banda de las peticiones y pueden ser útiles para identificar posibles cuellos de botella en la red.

Como comentarios sobre los resultados obtenidos, se puede ver que se hicieron 100 peticiones de cada tipo de petición (GET censo form, POST censo, GET voto form y POST voto) (ya que tenemos 10 users o threads con 10 iteraciones del bucle o loop count, haciendo cada user cada petición 10 veces), que los tiempos de respuesta y ancho de banda enviado eran mucho mayores en los POST que en los GET (lógico porque hay que procesar el cuerpo con POST y con GET no, y con POST nuestra app realiza más operaciones que con GET), que no hubo ningún error en las peticiones (los votos se registraron bien), y que el ancho de banda recibido era menor en POST voto (lógico porque no retorna ningún formulario y las otras solicitudes sí). En cuanto a los detalles, no se comentan porque se pueden ver en la captura.

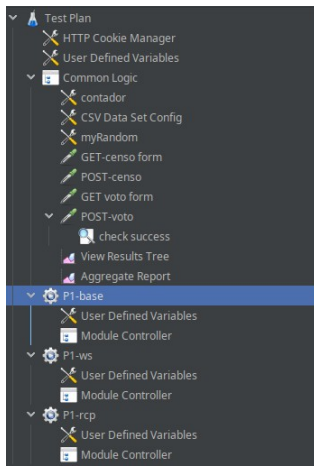
Ejercicio número 2:

Generaliza el plan de pruebas de forma que las pruebas se realicen sobre los tres proyectos creados en la práctica 1: P1-base, P1-rcp y P1-ws. Guarda este plan en un fichero llamado P2-projects.jmx y adjúntalo a la práctica. El plan de pruebas para P1-base ya lo tenéis creado y para los dos otros casos tendréis que duplicar el Thread Group P1-base, renombrar la copia adecuadamente y modificar los parámetros que almacenan el endpoint, host y port para que apunten a la aplicación web adecuada.

Primero, creamos el plan de pruebas como se nos dice (adjunto en la entrega con el nombre que se pide) (se puede ver las capturas siguientes).

Copiamos P1-base y lo pegamos, pero cambiando los nombres del thread group y las variables definidas por el usuario como se muestra en las siguientes capturas.

La estructura que nos queda es esta:



Y los tread groups son en ambos casos así, pero con el nombre correspondiente (se puede ver en la captura anterior):

Thread Group

Name:

Comments:

Action to be taken after a Sampler error:

☒ Continue
 ☐ Start Next Thread Loop
 ☐ Stop Thread
 ☐ Stop Test
 ☐ Stop Test Now

Thread Properties

Number of Threads (users):

Ramp-up period (seconds):

Loop Count: ☐ Infinite

☒ Same user on each iteration
☐ Delay Thread creation until needed
☐ Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):

Los module controllers son en ambos casos así (tambien usan el test fragment de lógica común ya que deben hacer las peticiones POST y GET de censo y voto que están ahí):

Module Controller

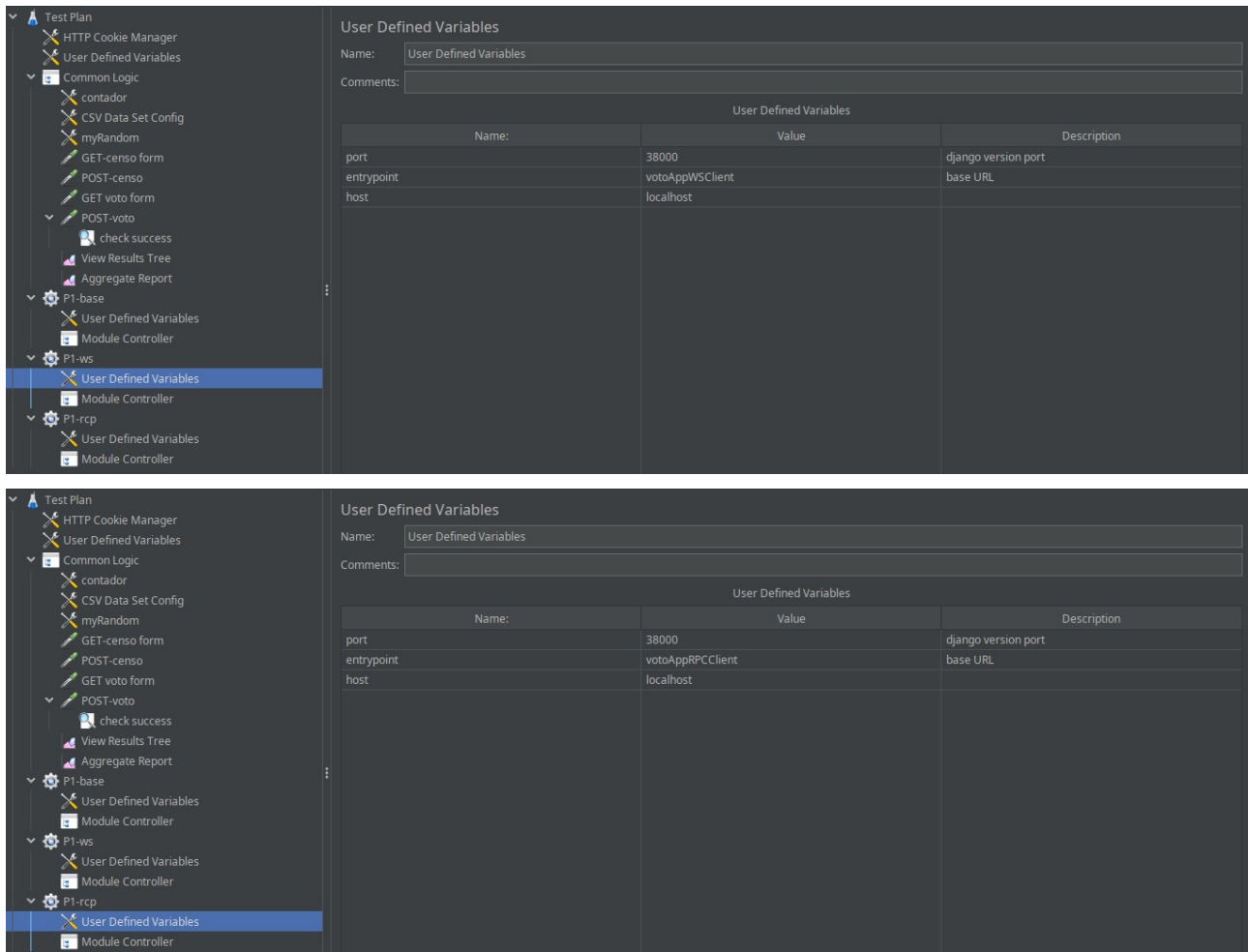
Name:

Comments:

Module To Run

- ☒ Test Plan
 - ☒ Common Logic
 - ☐ P1-base
 - ☐ P1-ws
 - ☐ P1-rpc

Pero en las variables definidas por el usuario de cada thread group, debemos poner el puerto 38000 (es donde estarán los clientes de las apps que usan API REST y RPC, en la VM3, estando ya el servidor y su base de datos donde toque, que será en otro ordenador), y el endpoint (URL base) votoAppRPCClient para el proyecto RPC y votoAppWSCClient para la API REST. Mantenemos como host localhost porque los tests se ejecutarán en el host que tiene en su VM3 el cliente:



Cuestión número 1:

Enumera las diferencias entre el Thread Group que prueba el proyecto P1-base y el que prueba el proyecto P1-ws.

Como se ha visto en el anterior ejercicio (ver explicación y capturas), ambos thread groups tienen el número de threads a 10 threads, el Ramp-Up Period a 2 segundos y el Loop Count al valor de la variable samples, pero en uno el nombre del thread group será P1-base y en el otro P1-ws. En cuanto al controlador del módulo, ambos usan el test fragment de lógica común ya que deben hacer las peticiones POST y GET de censo y voto que están ahí. En lo que cambian es principalmente en las variables definidas por el usuario, porque aunque ambos tienen como host localhost (ambos thread groups se ejecutarán desde el host que tiene en sus VM los clientes (en P1-base es lo mismo el cliente y el servidor)), P1-base tendrá el puerto a 28000 ya que el cliente y servidor (están unidos) estarán en la VM2 mientras que P1-ws lo tendrá a 38000 porque el cliente de la API REST estará en la VM3 (estando ya el servidor y la base de datos donde toque, que será en otro ordenador en la VM2 y la base de datos en la VM1), además de que en la variable entripoint (URL base) serán también distintos (votoApp en P1-base y votoAppWSCClient en P1-ws).

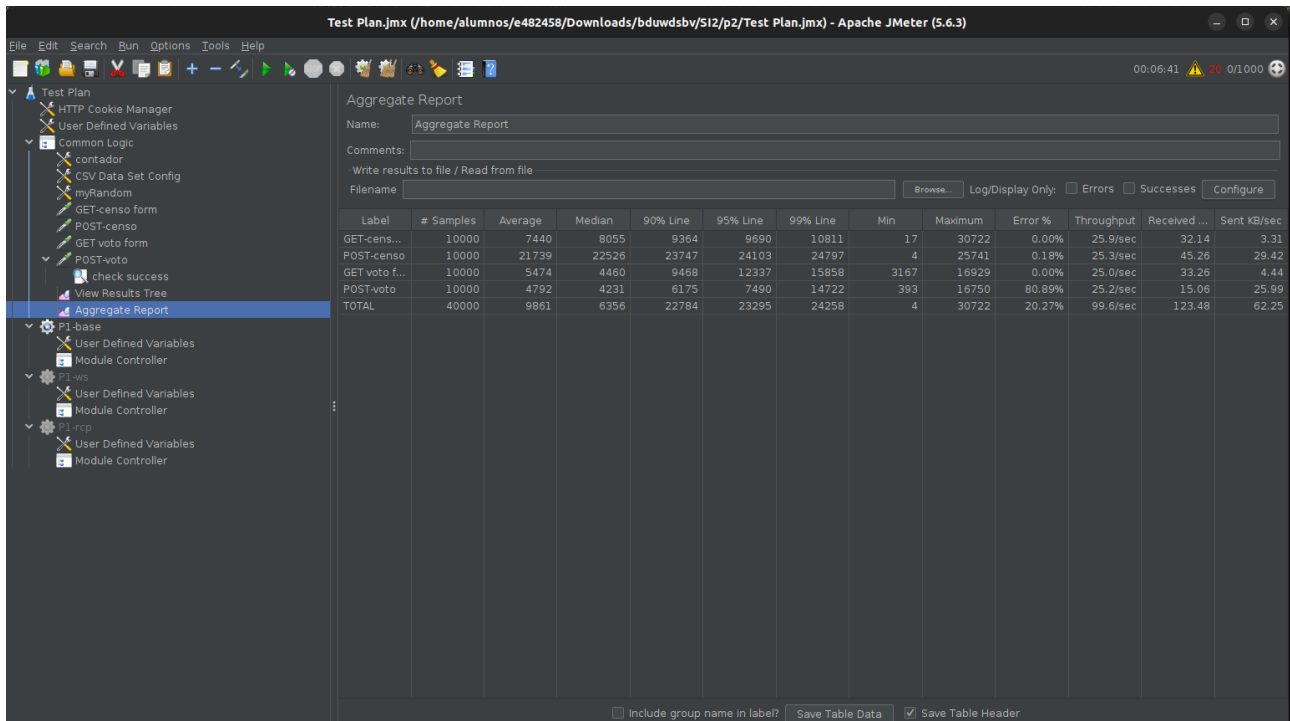
Ejercicio número 3:

Usando el Thread Group llamado P1-base si Ramp-Up = 1 sec y Number of Threads = 1000 se intentan crear 1000 solicitudes en 1 segundo. ¿En esta situación que ocurre en los ordenadores de los laboratorios? (a) JMeter no es capaz de obtener los recursos necesarios para crear 1000 peticiones en 1 segundo, (b) Aunque JMeter puede crear 1000 peticiones por segundo el throughput (esto, las peticiones atendidas por el servidor por segundo) será muy inferior aunque finalmente serán todas atendidas, (c) la capacidad del servidor se desborda y las peticiones se pierden o devuelven errores. A la vez que ejecutas las pruebas comprueba el uso de: memoria, disco, cpu y red en los distintos ordenadores (o máquinas virtuales) implicados en el cálculo. Para ello puedes usar la herramienta nmon (véase apéndice A). Argumenta tu respuesta e incluye una descripción de

los experimentos realizados para comprobar cual de las hipótesis es la correcta. Aporta capturas de pantalla para dar soporte a tu argumento

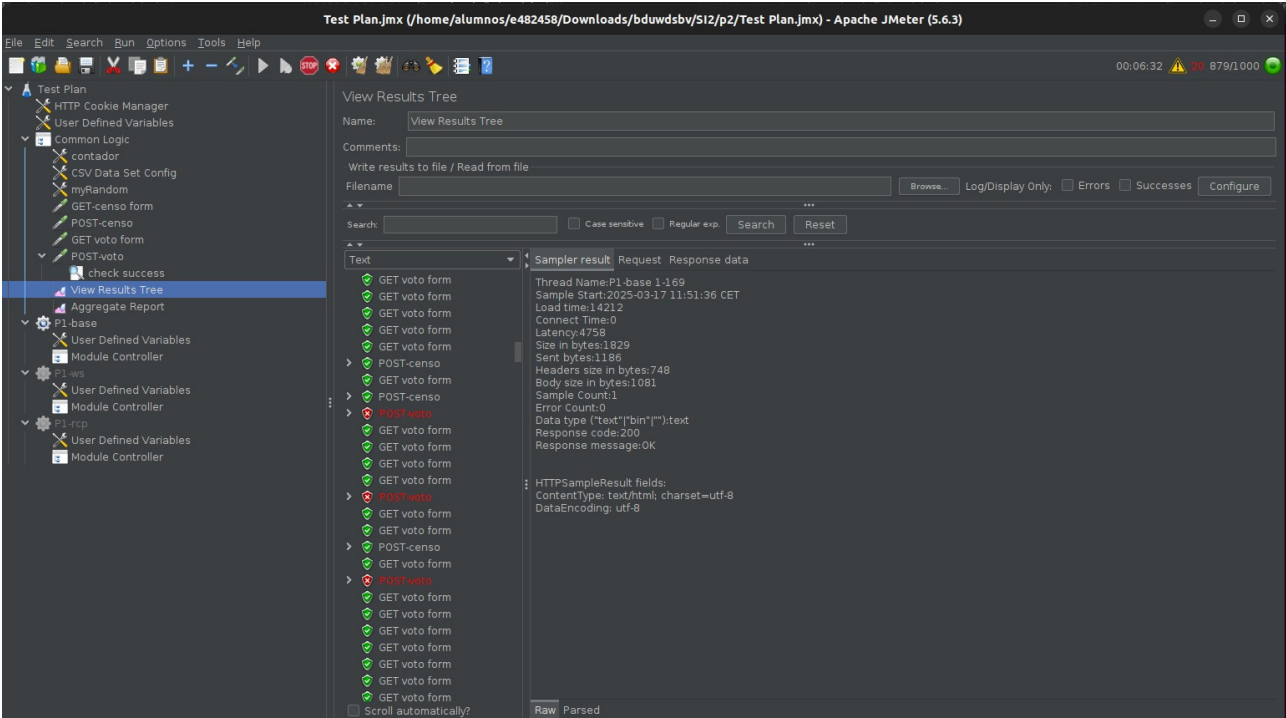
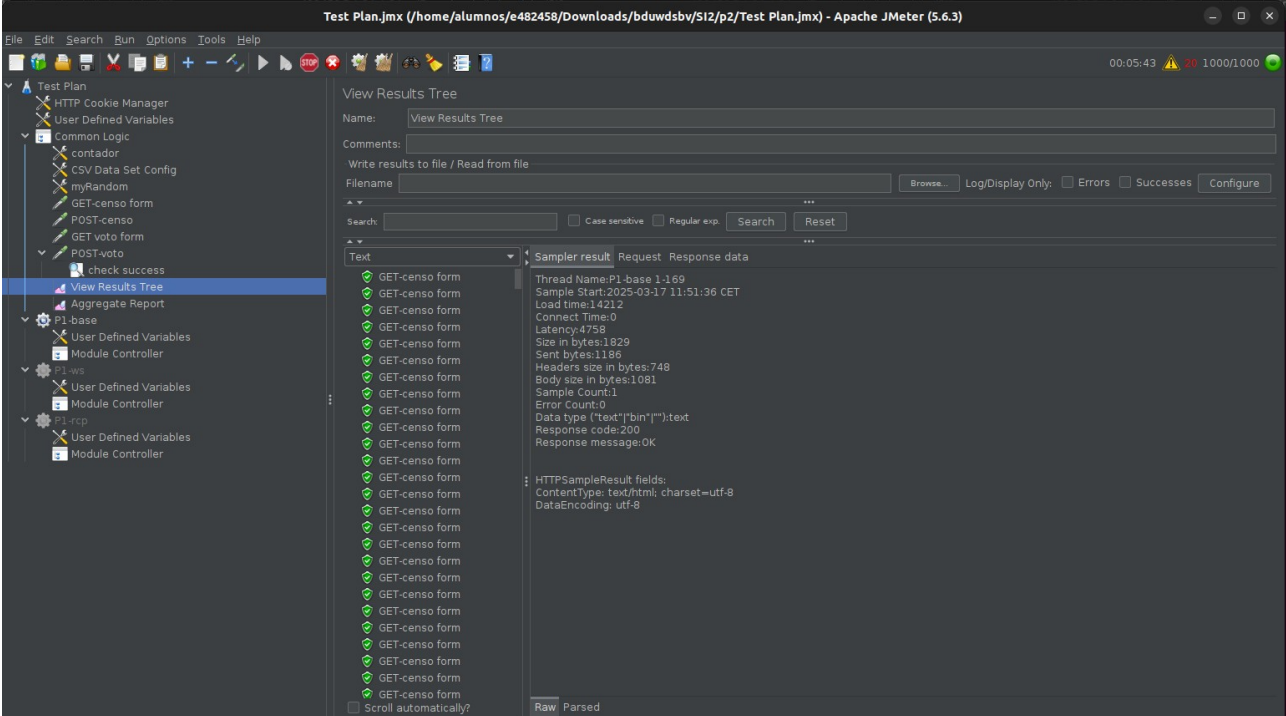
Para responder a esta pregunta, hacemos lo que se nos dice en un ordenador de los laboratorios, poniendo Ramp-Up a 1 y Number of Threads a 1000. Al ejecutar (tras unos minutos que tarda hasta terminar), obtenemos los siguientes resultados en la interfaz de jmeter:

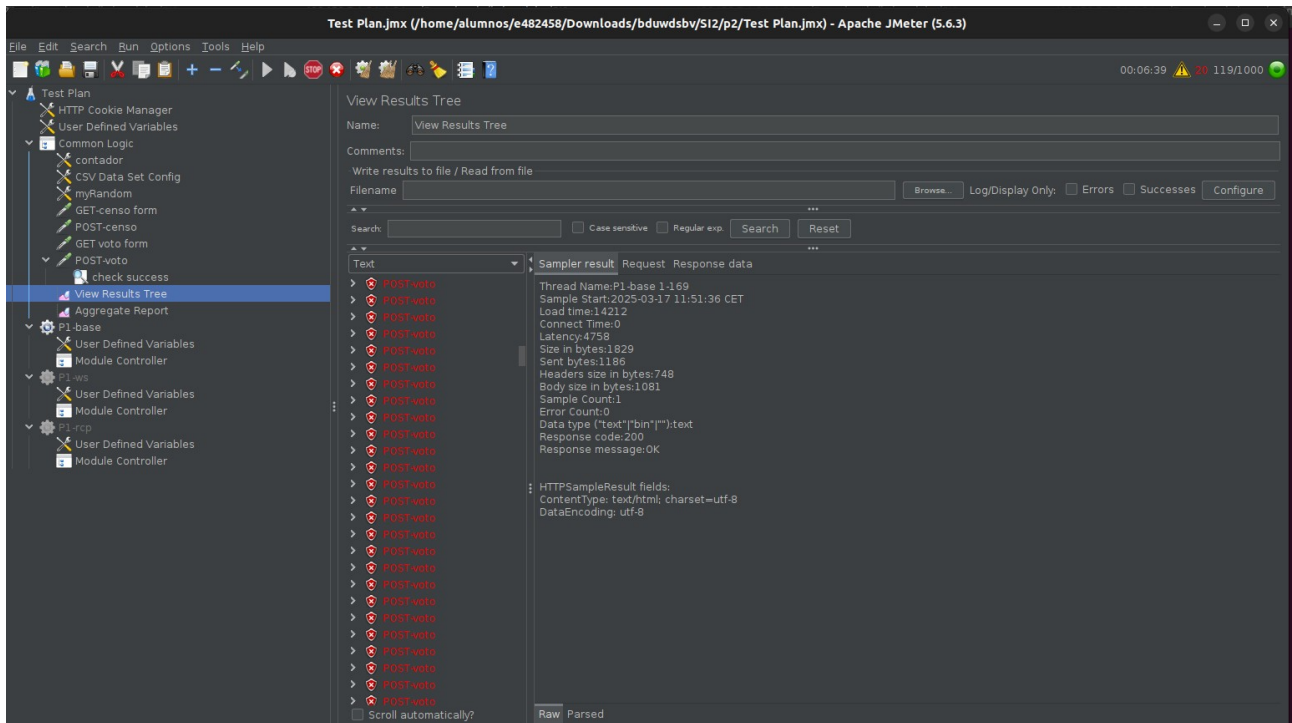
En el aggregate report se aprecia que en las peticiones de registrar el voto (POST-voto), se ha obtenido un porcentaje de error del 80.89%, y en verificar el censo (POST-censo), un 0.18% (20.27% en total). Esto nos hace pensar que la opción **(b) Aunque JMeter puede crear 1000 peticiones por segundo el throughput (esto, las peticiones atendidas por el servidor por segundo) será muy inferior aunque finalmente serán todas atendidas** no es correcta. Además, podemos apreciar también en la columna Samples que la opción **(a) JMeter no es capaz de obtener los recursos necesarios para crear 1000 peticiones en 1 segundo** no es del todo correcta ya que sí que se hicieron todas las peticiones solicitadas (10 de loop count o iteraciones del bucle por 1000 del número de hilos para cada tipo de petición, que son 10000):



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received ...	Sent KB/sec
GET-cens...	10000	7440	8055	9364	9690	10811	17	30722	0.00%	25.9/sec	32.14	3.31
POST-cens...	10000	21739	22526	23747	24103	24797	4	25741	0.18%	25.3/sec	45.26	29.42
GET voto f...	10000	5474	4460	9468	12337	15858	3167	16929	0.00%	25.0/sec	33.26	4.44
POST-voto	10000	4792	4231	6175	7490	14722	393	16750	80.89%	25.2/sec	15.06	25.99
TOTAL	40000	9861	6356	22784	23295	24258	4	30722	20.27%	99.6/sec	123.48	62.25

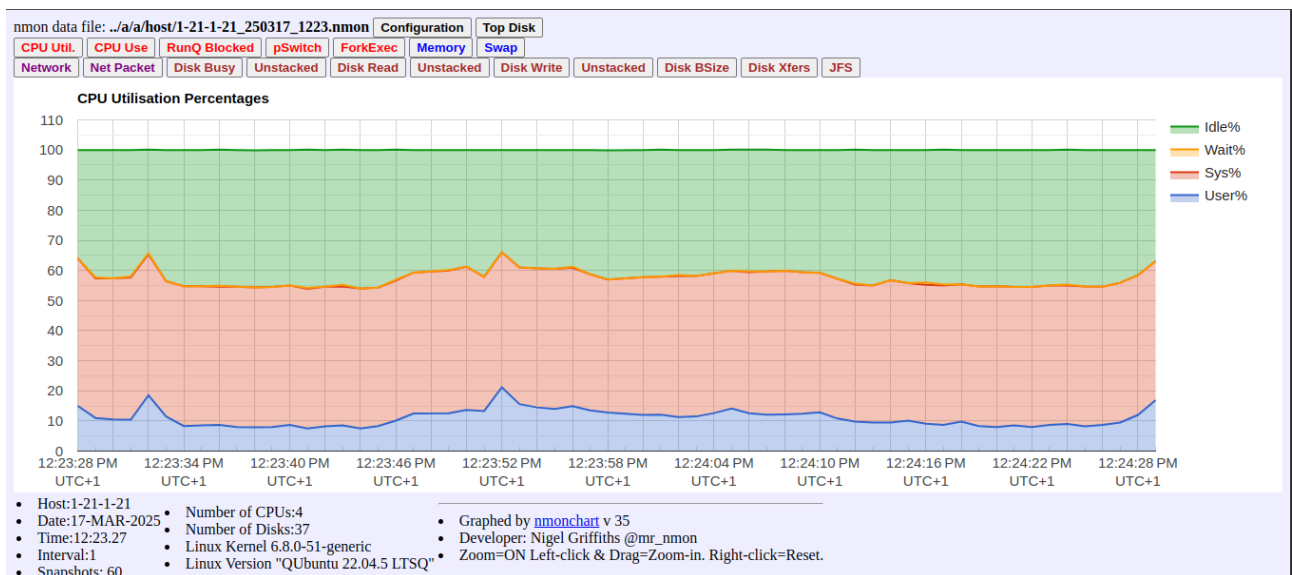
Viendo la vista de resultados en árbol, se ve que algunas se resolvieron bien y otras fallaron:

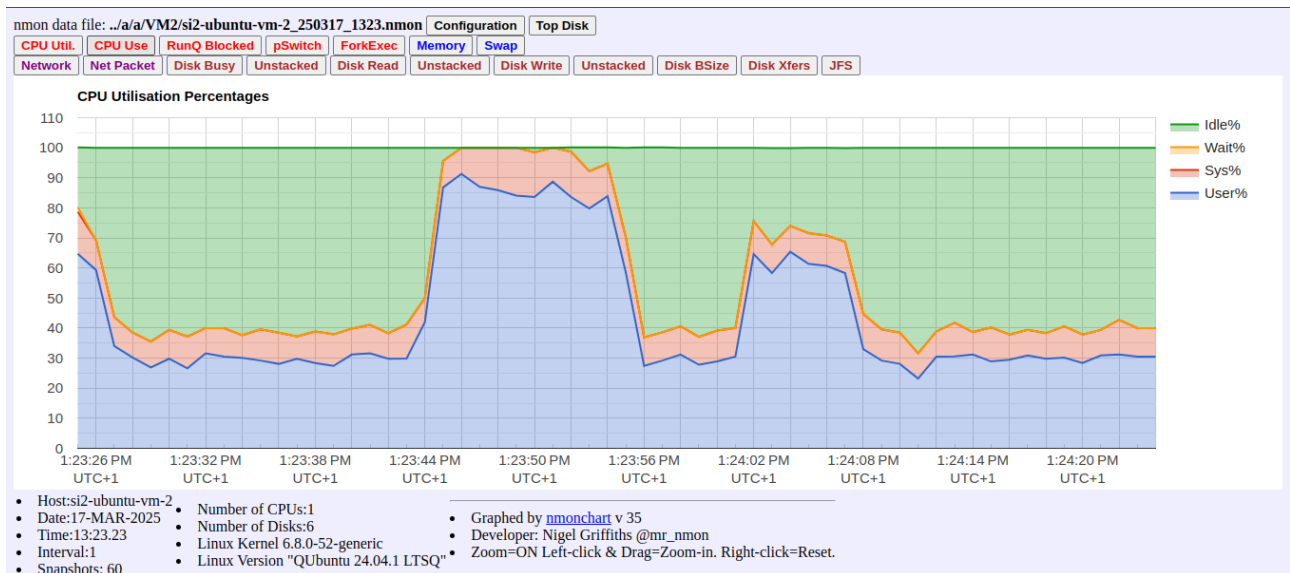
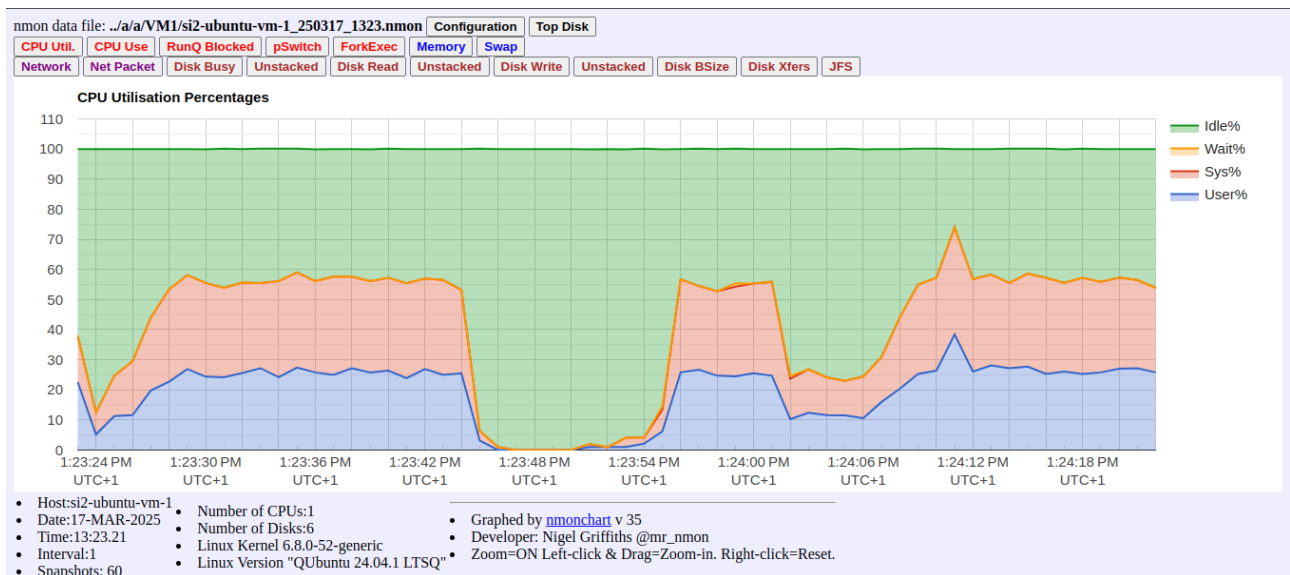




Además, pocos segundos después de que comenzasen las pruebas pusimos el comando `nmon -f -s 1 -c 60` para que se tomasen datos de memoria, disco, cpu y red cada segundo a lo largo de 60 segundos (pensar que la prueba duró varios minutos). El comando lo ejecutamos en el host (en el que se ejecutan las pruebas), en la VM1 (con la base de datos) y en la VM2 (con el servidor) al mismo tiempo. Ahora, pegaremos capturas de lo que vimos en cada uno de ellos e iremos explicando conclusiones que se pueden obtener.

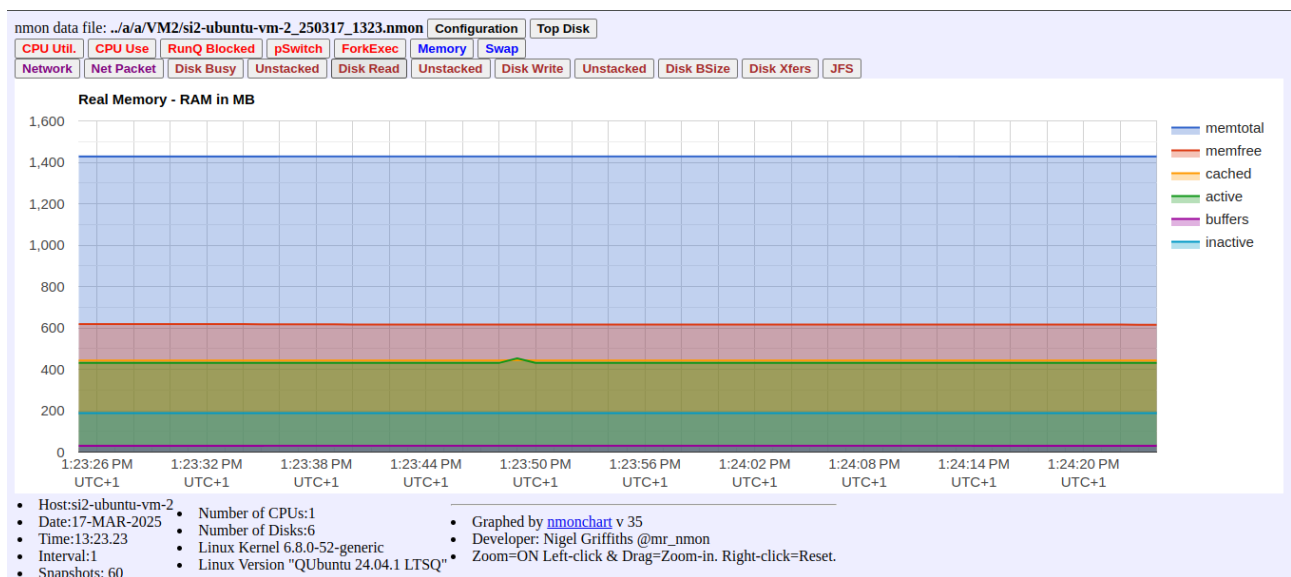
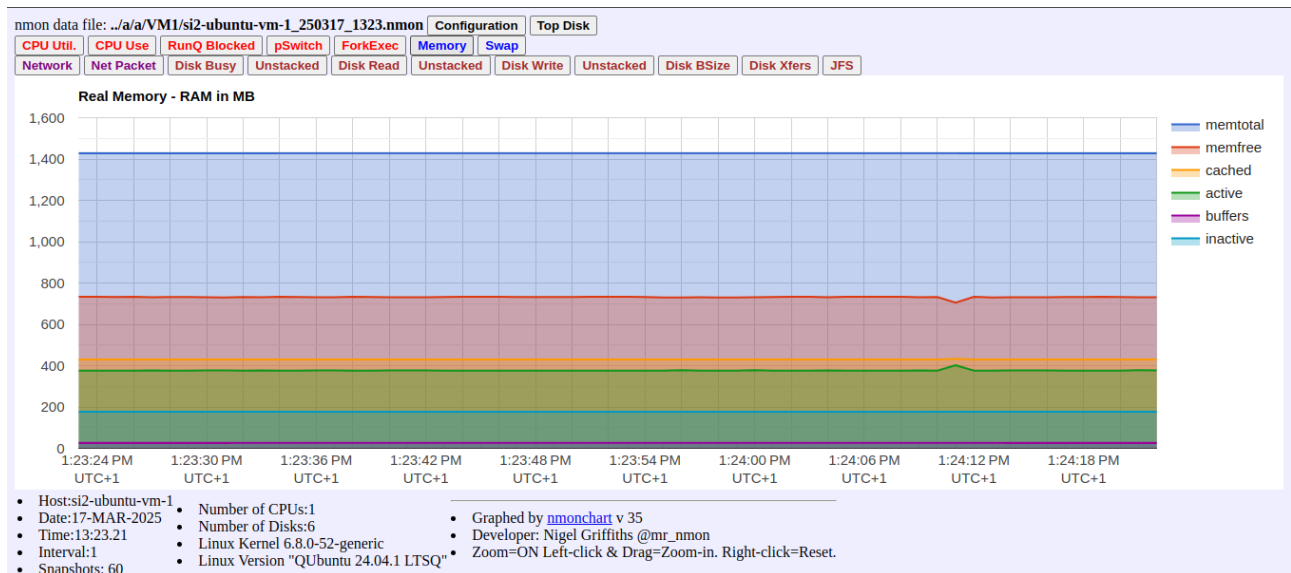
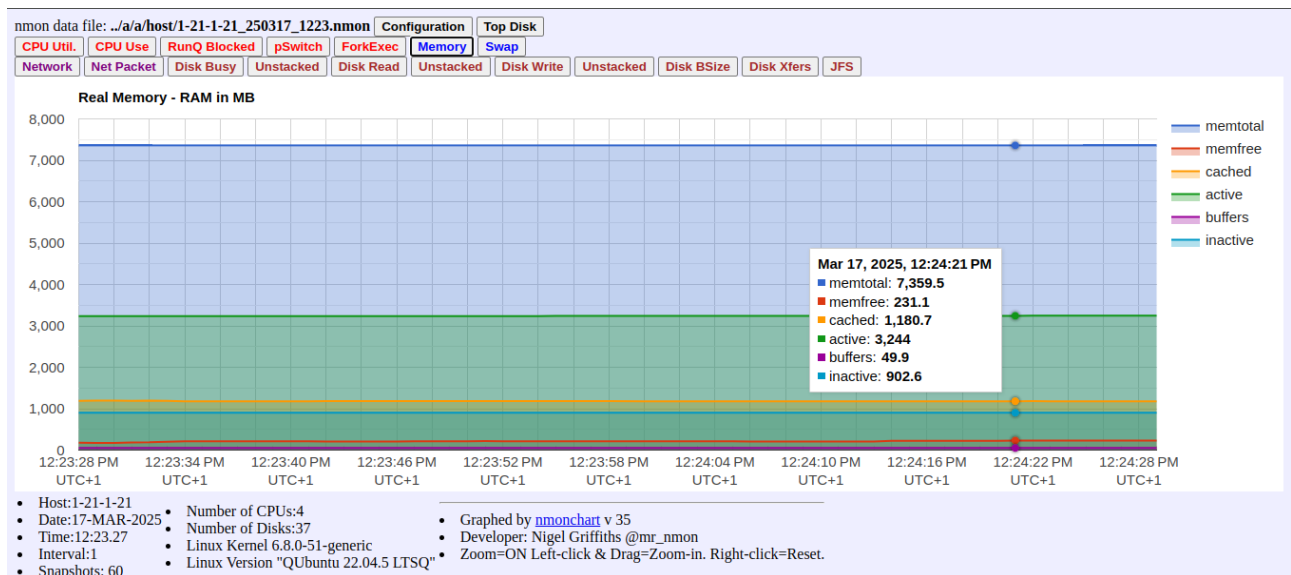
Uso de CPU a lo largo del tiempo (primero host, luego VM1 y luego VM2):





Se puede ver que en los tres el porcentaje de la CPU en wait o en modo system (iguales) supera al porcentaje en modo user, pero lo interesante a destacar es que se puede ver como mientras que en el host el uso de la cpu no varía demasiado al ejecutar los tests, en la VM2 hay dos picos del uso de CPU que coinciden con las bajadas drásticas en el uso de la CPU en la VM1. Esto se debe a que el servidor (VM2) se satura de procesar tantas peticiones, empezando a devolver errores (como se vio antes) y a no registrar estos votos. Al no registrarlos cuando se satura el servidor, la base de datos (VM1) no tiene que almacenar nada, con lo que la CPU de la VM1 se libera de la carga que tenía antes al almacenar constantemente datos. Además, podemos ver que el uso de la CPU del host al ejecutar jmeter no es tan gigantesco como para que no pueda hacer todas las peticiones, cosa que nos hace pensar que la opción **(a) JMeter no es capaz de obtener los recursos necesarios para crear 1000 peticiones en 1 segundo** no es del todo correcta.

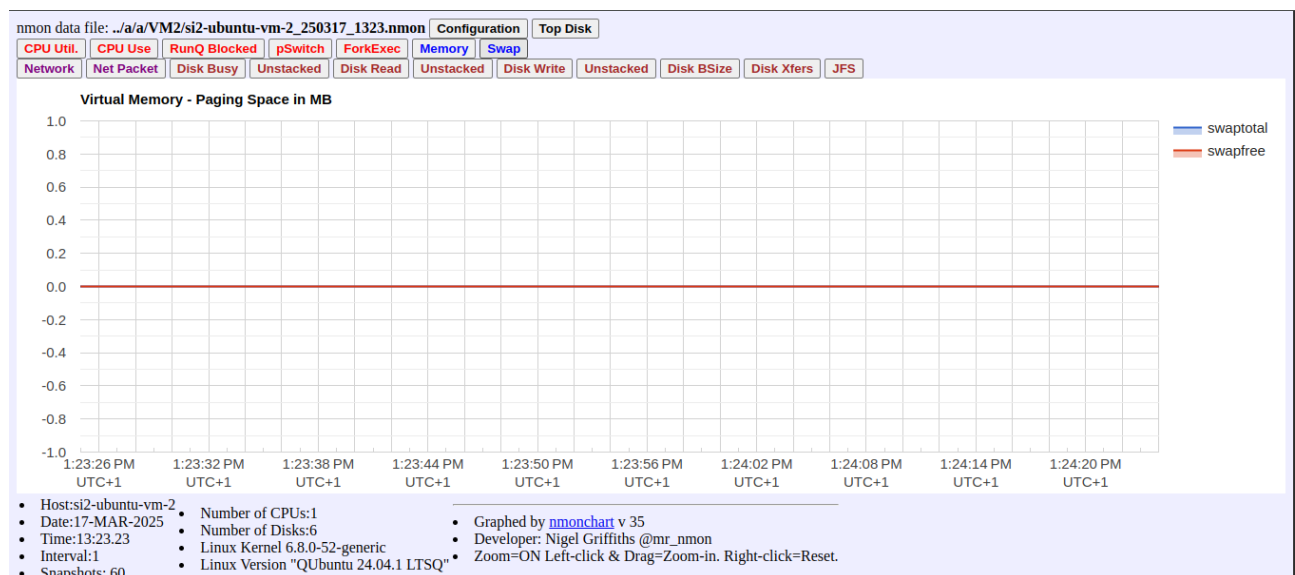
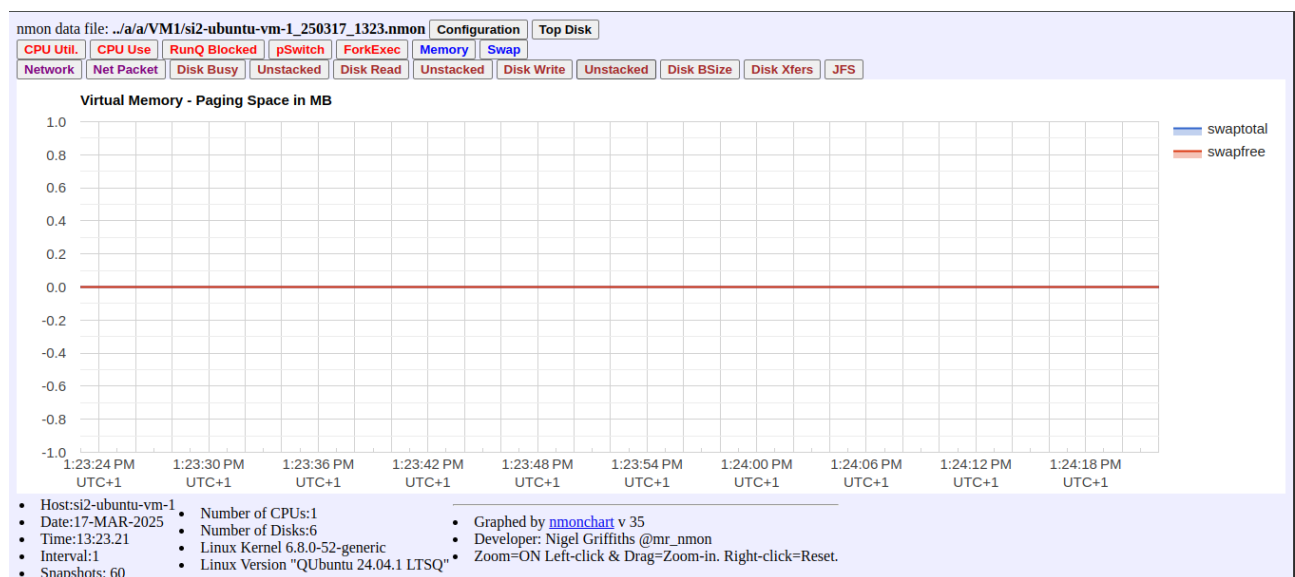
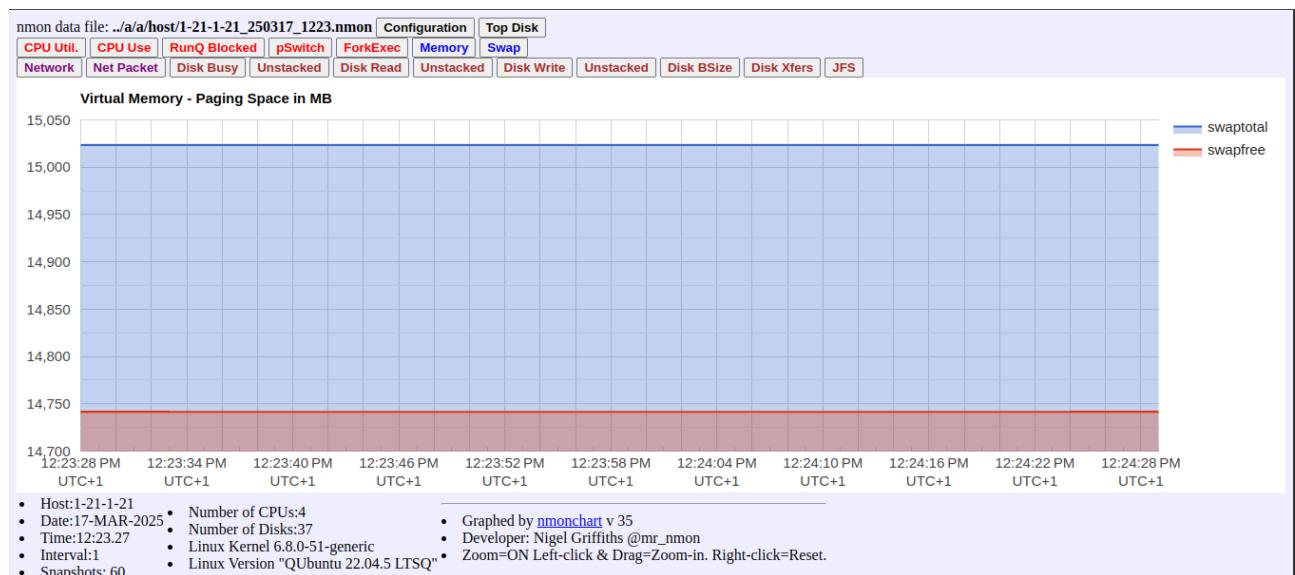
Uso de memoria real a lo largo del tiempo (primero host, luego VM1 y luego VM2):



En cuanto al uso de memoria real, no hay muchos datos interesantes para nuestra respuesta que podamos extraer (aunque no nos sirve, se ve que el host usa mucha más memoria real que las VM, cosa que es lógica ya que es la máquina real). Lo único es que podemos ver que el uso de la memoria real del host al ejecutar jmeter no es tan gigantesco como para que no pueda hacer todas las peticiones, cosa que nos hace pensar que la opción **(a) JMeter no es capaz de obtener los recursos necesarios para crear 1000**

peticiones en 1 segundo no es del todo correcta.

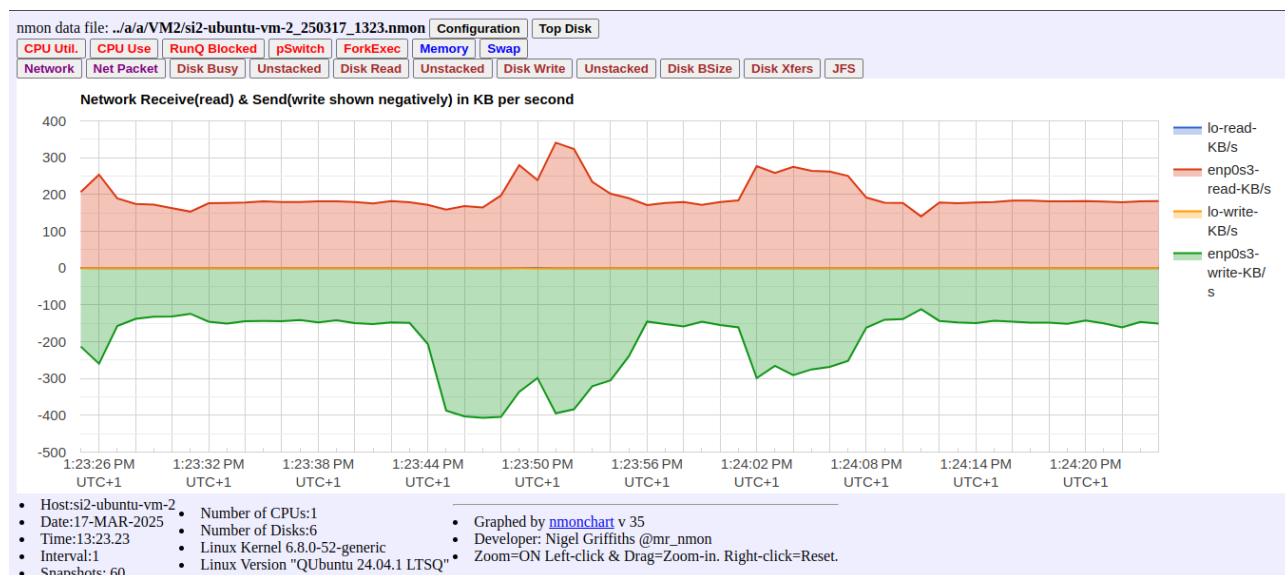
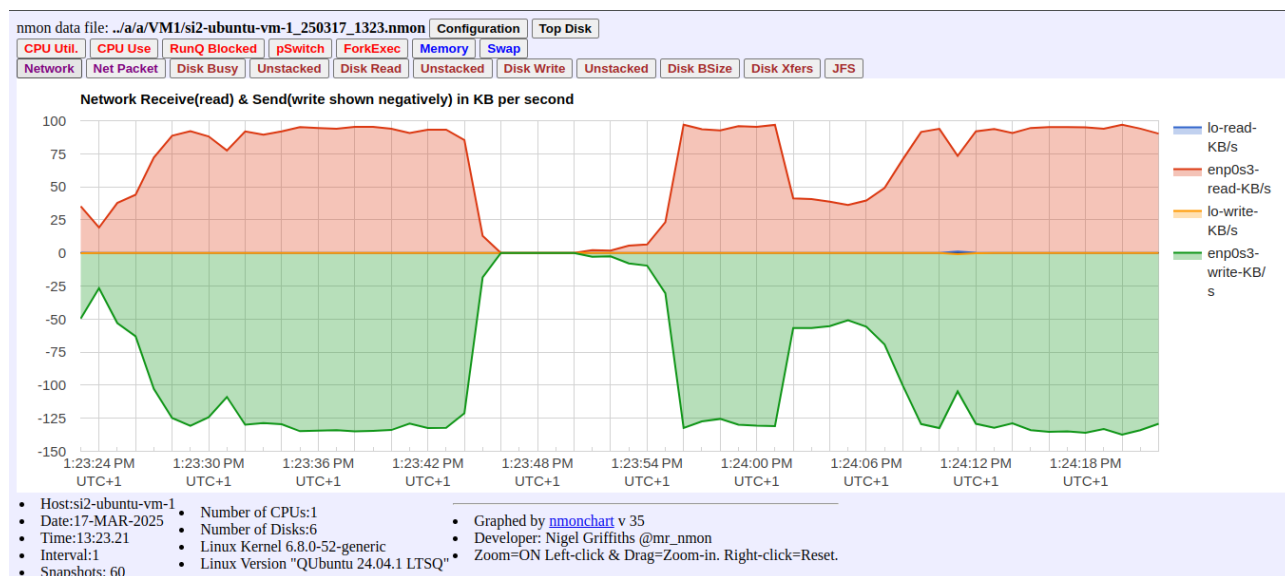
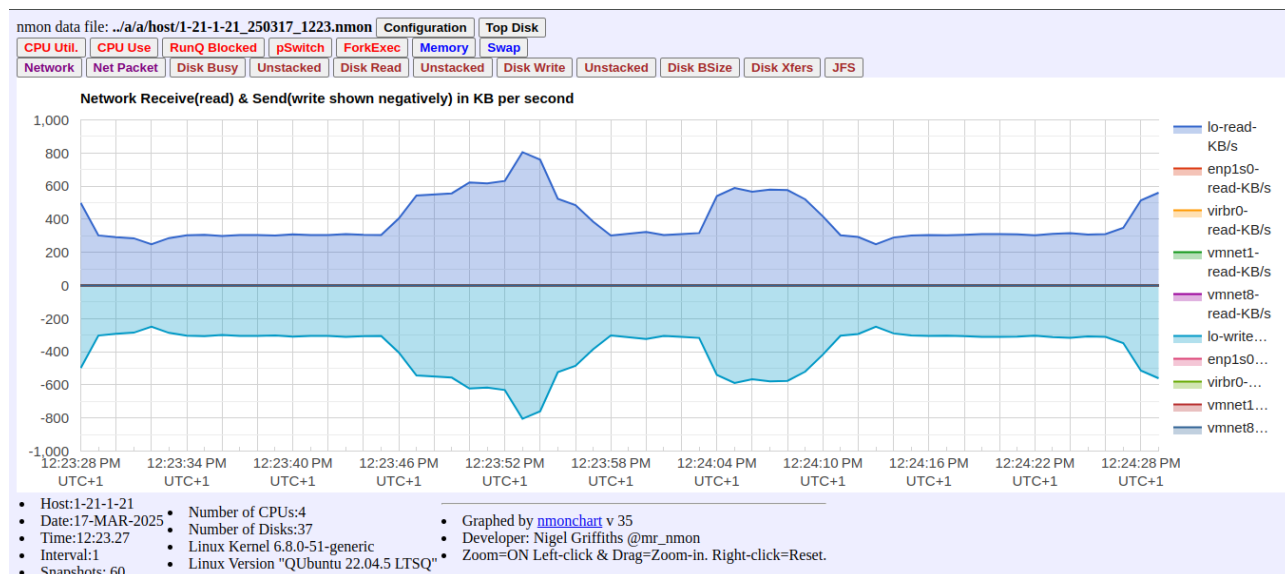
Uso de memoria virtual a lo largo del tiempo (primero host, luego VM1 y luego VM2):



En el caso de la memoria virtual, aunque no nos sirve para la respuesta, se puede apreciar que el host si

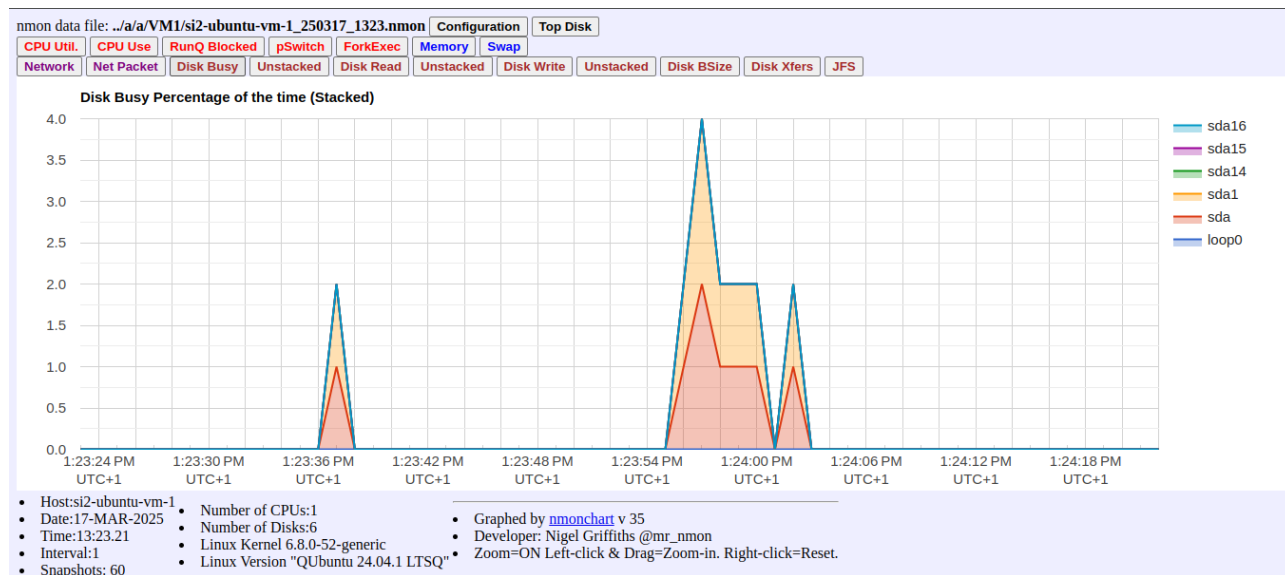
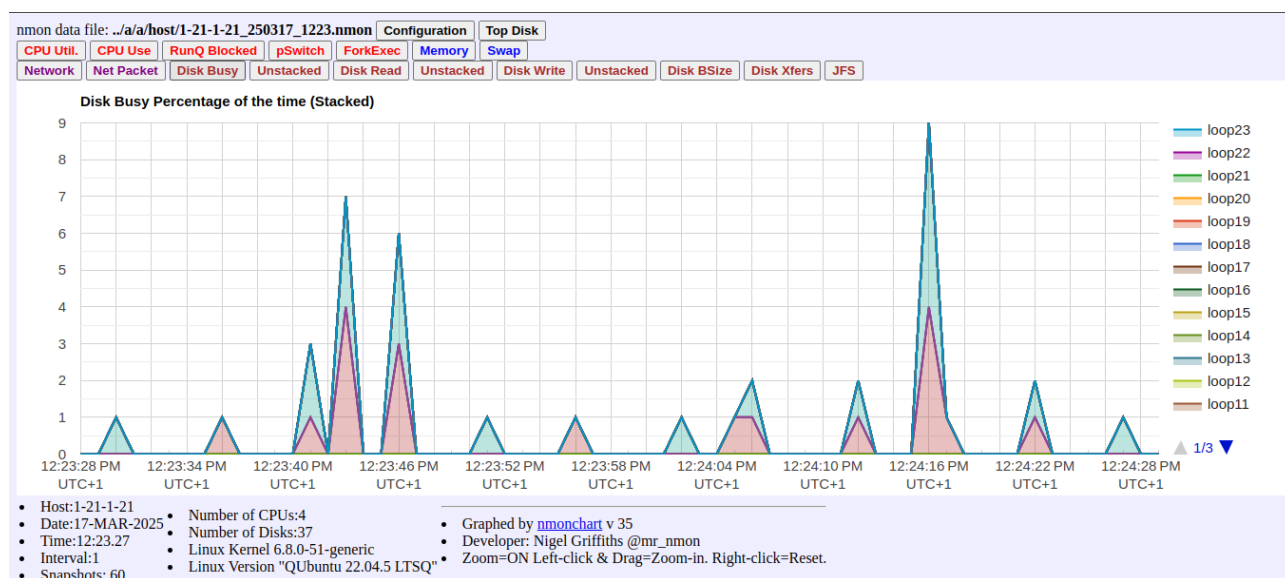
que hace uso de la misma, mientras que las VM no. Algo que podemos ver es que el uso de la memoria virtual del host al ejecutar jmeter no es tan gigantesco como para que no pueda hacer todas las peticiones, cosa que nos hace pensar que la opción **(a) JMeter no es capaz de obtener los recursos necesarios para crear 1000 peticiones en 1 segundo** no es del todo correcta.

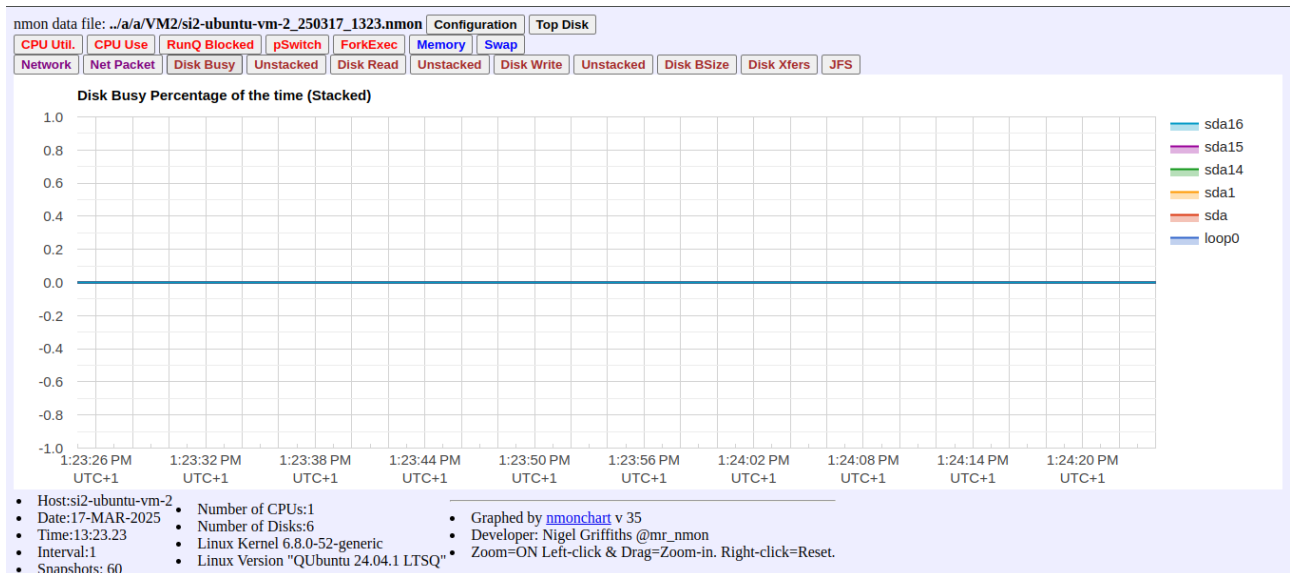
Uso de la red a lo largo del tiempo (primero host, luego VM1 y luego VM2):



Mirando el uso de la red, claramente se puede ver que en el caso del host es de la interfaz de loopback, que es con la que se comunica con el servidor para hacer las peticiones y recoger las respuestas (localhost, al que nos conectamos para acceder al servidor). Además, se puede ver que la forma de la gráfica del host y la de la VM2 son iguales (aunque en escalas distintas), lo que es lógico porque ambas se están conectando para que el host envíe peticiones con jmeter al servidor de la VM2 y recoja el resultado; pero, sin embargo, en la VM1, en los picos de las otras gráficas, esta tiene descensos pronunciados (hasta 0 de manera drástica en ocasiones). Esto se debe a lo explicado antes de que cuando el server de la VM2 se satura, deja de guardar votos en la base de datos y, por tanto, la comunicación con la VM1 (base de datos) disminuye momentáneamente. Además, podemos ver que el uso de la red del host al ejecutar jmeter no es tan gigantesco como para que no pueda hacer todas las peticiones, cosa que nos hace pensar que la opción **(a) JMeter no es capaz de obtener los recursos necesarios para crear 1000 peticiones en 1 segundo** no es del todo correcta.

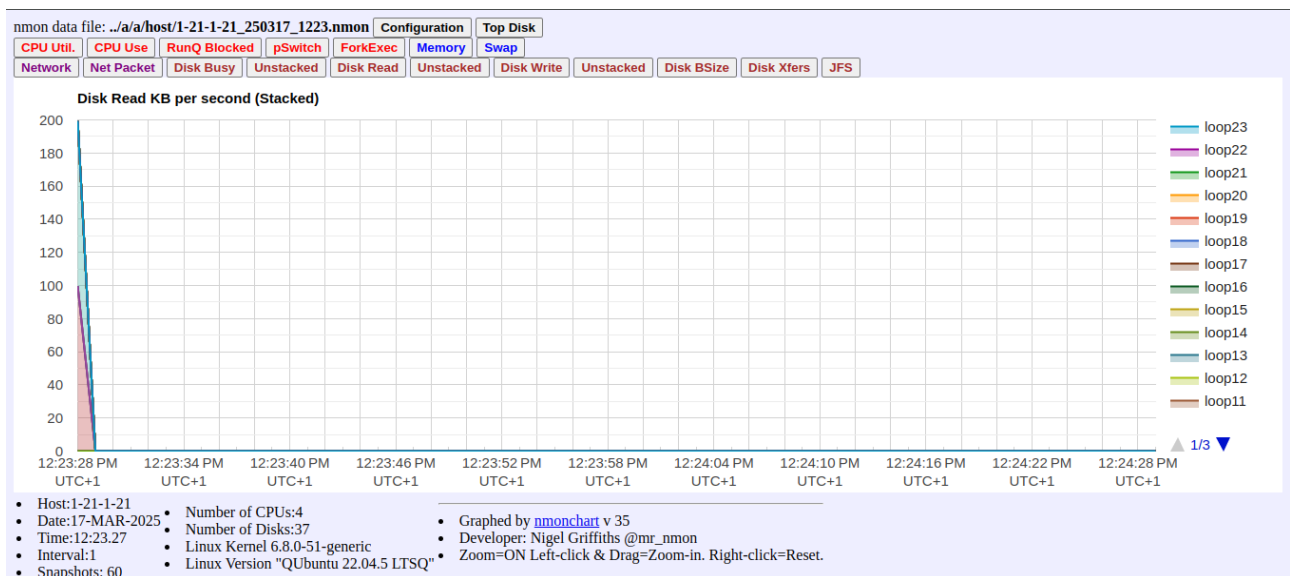
Uso de disco (busy) (primero host, luego VM1 y luego VM2):

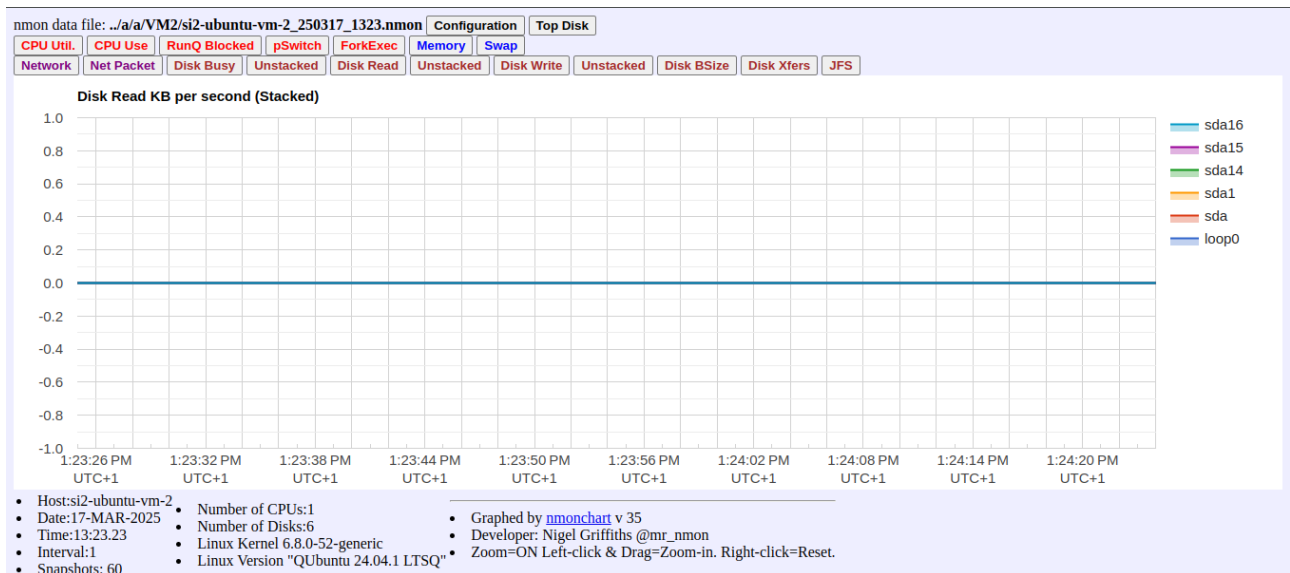
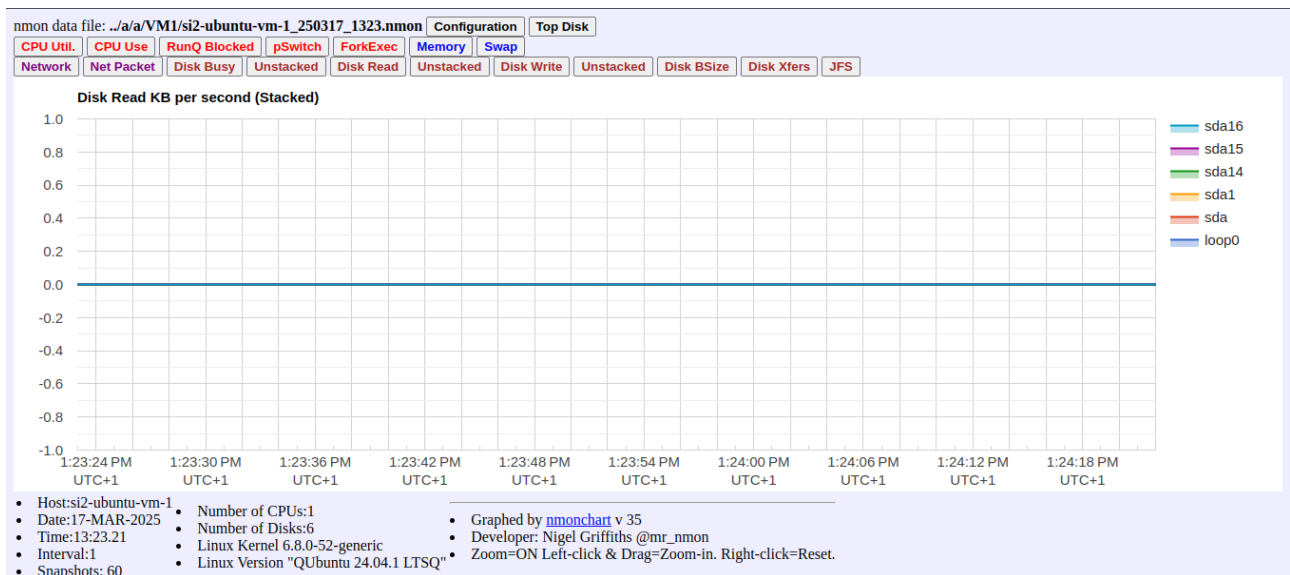




En cuanto al uso del disco en porcentaje del tiempo, en el host es irregular, pero esto no nos dice mucho (aunque no nos sirve, se ve que el host usa mucho más el disco que la VM1 viendo la escala del eje y, cosa que es lógica ya que es la máquina real). Lo que sí que nos es útil es el hecho de que en la VM1 (base de datos) hay momentos puntuales en los que se escriben cosas en disco. No se hace continuamente sino en momentos puntuales porque probablemente la base de datos esté configurada para acumular las actualizaciones y luego hacerlas todas de una en disco para minimizar los lentos accesos a este. En cuanto a la VM2, no hay uso del disco ya que no guarda nada en su disco, sino que se lo envía a la VM1 para que esta lo guarde en la base de datos.

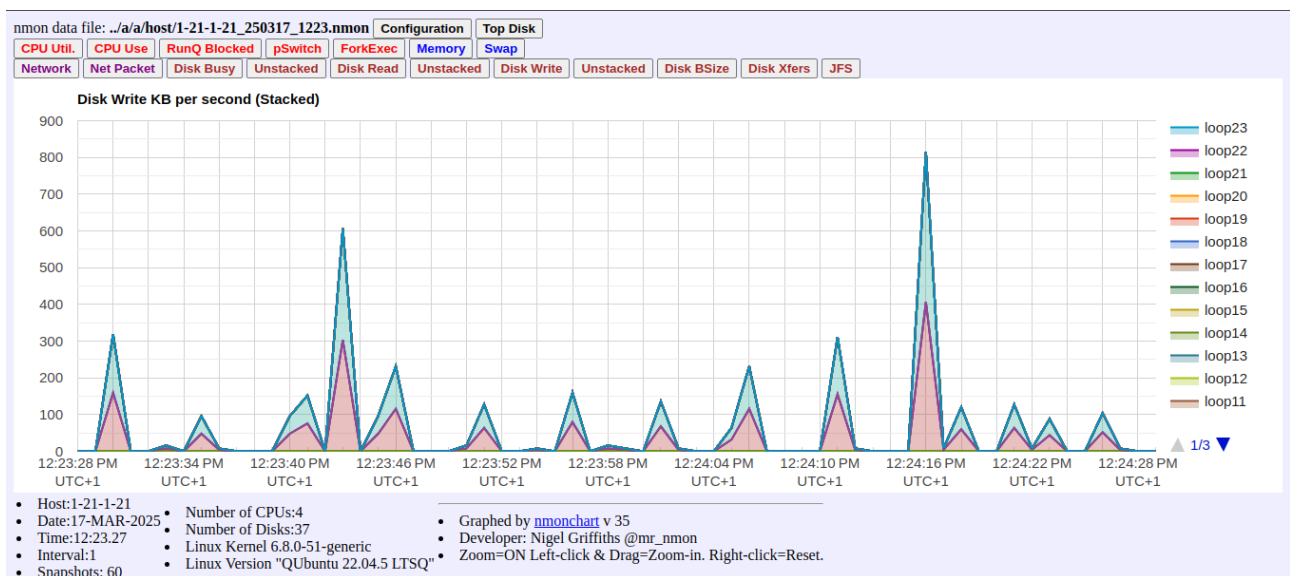
Uso de disco (read) (primero host, luego VM1 y luego VM2):

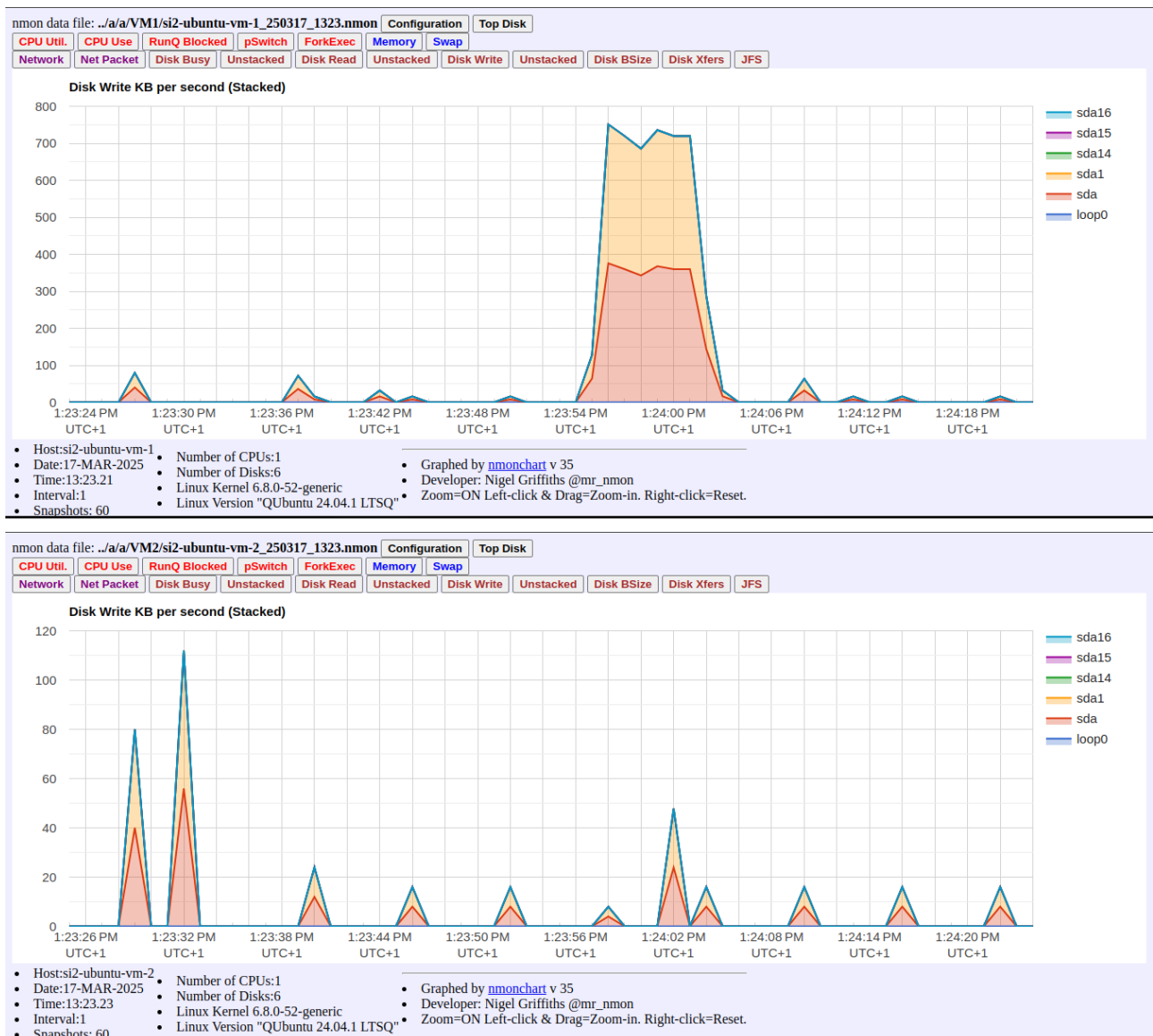




Mirando los tamaños de datos leídos de disco, ni en el host ni en las VM hay nada. En la VM2 se debe a que esta no usa el disco en lectura porque los datos de la app se guardan en la base de datos de la VM1.

Uso de disco (write) (primero host, luego VM1 y luego VM2):





En cuanto a las escrituras de disco, lo que principalmente podemos destacar es lo comentado antes: el hecho de que en la VM1 (base de datos) hay momentos puntuales en los que se escriben cosas en disco. No se hace continuamente sino en momentos puntuales porque probablemente la base de datos esté configurada para acumular las actualizaciones en RAM y luego hacerlas todas de una en disco para minimizar los lentos accesos a este. En cuanto a la VM2, no hay uso del disco apenas (ver que el eje y es menor que en las otras) ya que no guarda nada de la base de datos de la app en su disco, sino que se lo envía a la VM1 para que esta lo guarde en la base de datos.

Por lo visto hasta ahora, destacando principalmente los argumentos vistos en las capturas de la interfaz de jmeter y en las gráficas del uso de la CPU y del uso de la red, decidimos que la opción correcta es la respuesta:

(c) la capacidad del servidor se desborda y las peticiones se pierden o devuelven errores.

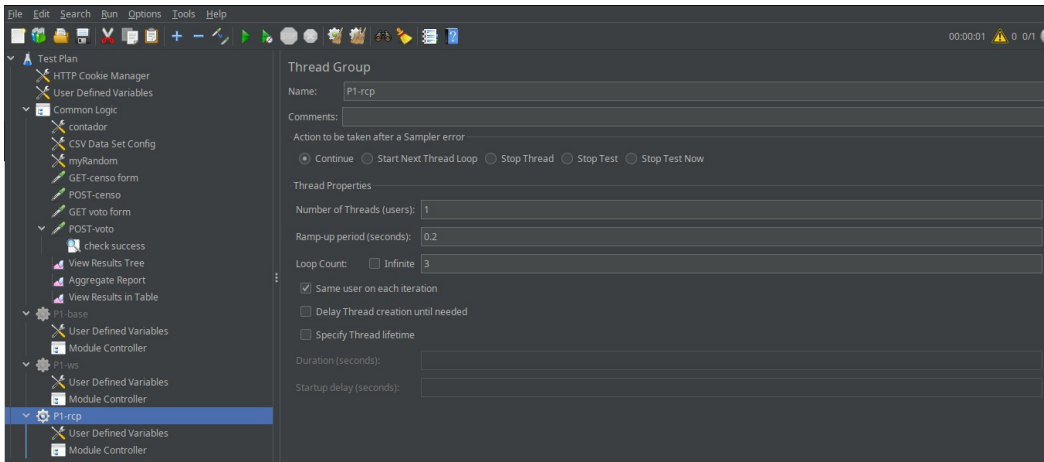
Ejercicio número 4:

En un Thread Group, la variable Loop Count dene cuantas veces cada thread (usuario) ejecuta una secuencia de peticiones. ¿Cuándo comienza cada loop (bucle)? (a) El segundo bucle empieza tan pronto como todas las peticiones del primer bucle se han creado o (b) El segundo bucle comienza tan pronto como todas las peticiones del primer bucle se han atendido. Argumenta tu respuesta incluye una descripción de los experimentos realizados para comprobar cual de las hipótesis es la correcta. Aporta capturas de pantalla para dar soporte a tu argumento.

Se destaca que hemos hecho este ejercicio con el proyecto RPC porque antes lo estábamos probando (nos funciona) y por esto salen los resultados con este nombre en vez de con P1-base (no ponemos el P1-base porque para esta pregunta no importa).

Primero, comprobaremos si un mismo hilo manda la primera consulta de su segunda iteración antes de que haya llegado la respuesta de la última petición de la primera iteración. Para esto, hemos añadido al Test Fragment Common Logic un listener View Results in Table ya que en este se nos mostrarán las peticiones hechas en detalle: el momento en el que se inició/envió cada petición (columna Start Time) y cuánto tiempo se tardó desde que se envió hasta que se recibió la respuesta (columna Latency), además de qué hilo hace cada petición (columna Thread Name) (para lo que explicaremos luego).

Ponemos un sólo hilo y 3 iteraciones para hacer la comprobación, junto con un Ramp-up de 0.2 segundos ya que la ejecución será muy rápida:



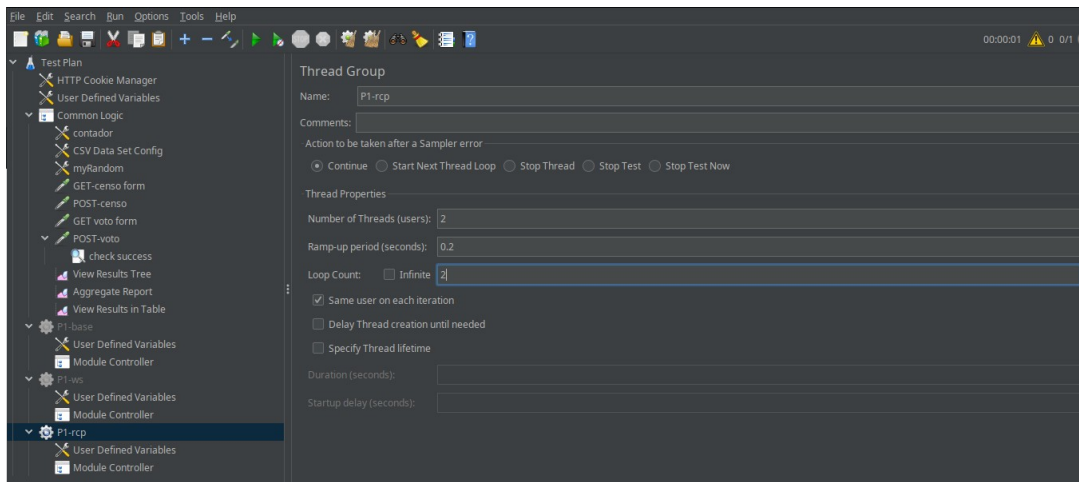
En la tabla de resultados, ordenamos de manera ascendente (ver flechita hacia arriba) por Start Time para detectar los comienzos de cada iteración (la primera petición de cada iteración está marcada en azul). Podemos apreciar que entre el comienzo/envío de la última petición de la primera iteración y el comienzo/envío de la primera petición de la segunda iteración hay 32 ms (743 ms – 711 ms), y que la latencia de la última petición de la primera iteración es de 31 ms, que es menor que 32 ms (llegó la respuesta de la última petición de la primera iteración justo antes de que se hiciese la primera petición de la segunda iteración). Si lo miramos con respecto a la segunda y tercera iteración, sucede lo mismo (29 ms (823 ms – 794 ms) de separación entre peticiones de las iteraciones, que es igual que la latencia de la última petición de la segunda iteración). De esto, por el parecido (diff start time >= latencia) de estas diferencias entre los start times y las latencias que acabamos de mencionar, podemos inducir que se cumple siempre que un mismo hilo espera a que hayan sido atendidas todas las peticiones de un bucle antes de comenzar el siguiente (opción (b) del enunciado). Probando más veces, se sigue cumpliendo esto, por eso lo inducimos.

The screenshot shows the 'View Results in Table' window in JMeter. The table displays the following data:

Sample #	Start Time	Thread Name	Label	Sample Time...	Status	Bytes	Sent Bytes	Latency	Connect Time...
1	16:09:23.647	P1-rpc 1-1	GET-censo form	8	✓	1272	140	8	1
2	16:09:23.656	P1-rpc 1-1	POST-censo	47	✓	1838	1208	39	0
3	16:09:23.703	P1-rpc 1-1	GET voto form	8	✓	1363	191	8	1
4	16:09:23.711	P1-rpc 1-1	POST-voto	31	✓	712	1090	31	1
5	16:09:23.743	P1-rpc 1-1	GET-censo form	7	✓	1272	140	7	0
6	16:09:23.750	P1-rpc 1-1	POST-censo	39	✓	1838	1199	33	1
7	16:09:23.789	P1-rpc 1-1	GET voto form	5	✓	1363	191	5	0
8	16:09:23.794	P1-rpc 1-1	POST-voto	29	✓	712	1066	29	1
9	16:09:23.823	P1-rpc 1-1	GET-censo form	6	✓	1272	140	6	1
10	16:09:23.829	P1-rpc 1-1	POST-censo	32	✓	1838	1222	26	1
11	16:09:23.861	P1-rpc 1-1	GET voto form	6	✓	1363	191	6	1
12	16:09:23.867	P1-rpc 1-1	POST-voto	37	✓	712	1078	37	1

Ahora, hemos hecho una prueba para ver si entre hilos cada uno de ellos espera a que el resto acaben cada iteración (reciban respuesta de su última petición de la iteración) cuando ellos la han acabado para

comenzar la siguiente. Hemos puesto ahora 2 hilos y 2 iteraciones del bucle (Loop-count), manteniendo el Ramp-up porque era adecuado para una prueba tan corta:



En la tabla de resultados, ordenamos de manera ascendente (ver flechita hacia arriba) por Start Time para detectar los comienzos de cada iteración (la primera petición de cada iteración está marcada en azul). Podemos apreciar que entre el comienzo/envío de la última petición de la primera iteración, que es del hilo 1, y el comienzo/envío de la primera petición de la segunda iteración, que es del hilo 2, hay 35 ms (347 ms – 312 ms), y que la latencia de la última petición de la primera iteración, que es la del hilo 1, es de 66 ms, que es mayor que 35 ms (el hilo 2 comenzó su segunda iteración antes de que llegase la respuesta/fuese atendida la última petición de la primera iteración del hilo 1; es decir, no le esperó). Esto nos muestra que entre hilos no se esperan entre iteraciones, como hemos dicho.

Sample #	Start Time	Thread Name	Label	Sample Time...	Status	Bytes	Sent Bytes	Latency	Connect Time...
1	16:28:38.209	P1-rpc 1-2	GET-censo form	9	✓	1272	140	9	2
2	16:28:38.211	P1-rpc 1-1	GET-censo form	13	✓	1272	140	13	1
3	16:28:38.219	P1-rpc 1-2	POST-censo	67	✓	1838	1208	33	1
4	16:28:38.225	P1-rpc 1-1	POST-censo	72	✓	1838	1241	55	0
5	16:28:38.286	P1-rpc 1-2	GET voto form	19	✓	1363	191	19	0
6	16:28:38.297	P1-rpc 1-1	GET voto form	15	✓	1363	191	15	0
7	16:28:38.306	P1-rpc 1-2	POST-voto	41	✓	712	1054	41	1
8	16:28:38.312	P1-rpc 1-1	POST-voto	66	✓	712	1066	66	1
9	16:28:38.347	P1-rpc 1-2	GET-censo form	37	✓	1272	140	37	1
10	16:28:38.378	P1-rpc 1-1	GET-censo form	12	✓	1272	140	12	1
11	16:28:38.384	P1-rpc 1-2	POST-censo	64	✓	1838	1186	31	1
12	16:28:38.390	P1-rpc 1-1	POST-censo	63	✓	1838	1236	53	1
13	16:28:38.448	P1-rpc 1-2	GET voto form	10	✓	1363	191	10	0
14	16:28:38.453	P1-rpc 1-1	GET voto form	10	✓	1363	191	10	1
15	16:28:38.458	P1-rpc 1-2	POST-voto	38	✓	712	1096	38	1
16	16:28:38.463	P1-rpc 1-1	POST-voto	81	✓	712	1066	81	1

Por tanto, para responder a la pregunta, si entendemos que se esta preguntando si entre hilos cada uno de ellos espera a que el resto acaben (reciban respuesta de su última petición de la iteración) cada iteración cuando ellos la han acabado para comenzar la siguiente, la respuesta es que no (la (a) del enunciado); pero si nos referimos a si un mismo hilo manda la primera consulta de una iteración después de que haya llegado la respuesta de la última petición de la anterior iteración, la respuesta es que sí (la (b) del enunciado). Como nos parece más lógico que se esté preguntando lo segundo, nuestra respuesta es la **(b) El segundo bucle comienza tan pronto como todas las peticiones del primer bucle se han atendido.**

Ejercicio número 5:

Calcula la curva de productividad para cada uso de los tres proyectos llamados P1-base, P1-ws-client/server y P1-rpc-client/server e incluye el resultado en la memoria. Esto es, se espera que incluyáis una grafica como la mostrada en la gura 18 para cada caso. ¿Qué implementación es más

eficiente? ¿Por qué? Cada experimento debe repetirse 10 veces (usad la variable samples para lograr esta repetición). El ramp up debe ser pequeño con respecto al tiempo que se tarda en ejecutar todas las peticiones. Finalmente, asegurarnos de que las pruebas (grupos de hilos) se ejecutan secuencialmente y no concurrentemente. Incluid en la memoria junto a la gráfica la tabla donde habéis apuntado los throughputs.

Para asegurar que los thread groups se ejecutasen secuencialmente, marcamos la siguiente opción:

The screenshot shows the JMeter Test Plan configuration window. The 'Name' field is 'Test Plan'. The 'Comments' field is empty. Below is the 'User Defined Variables' section with a table for Name and Value. At the bottom, there are checkboxes for 'Run Thread Groups consecutively (i.e. one at a time)' (checked), 'Run tearDown Thread Groups after shutdown of main threads' (unchecked), and 'Functional Test Mode (i.e. save Response Data and Sampler Data)' (unchecked). There is also a section for 'Add directory or jar to classpath' with 'Browse...', 'Delete', and 'Clear' buttons, and a 'Library' section at the very bottom.

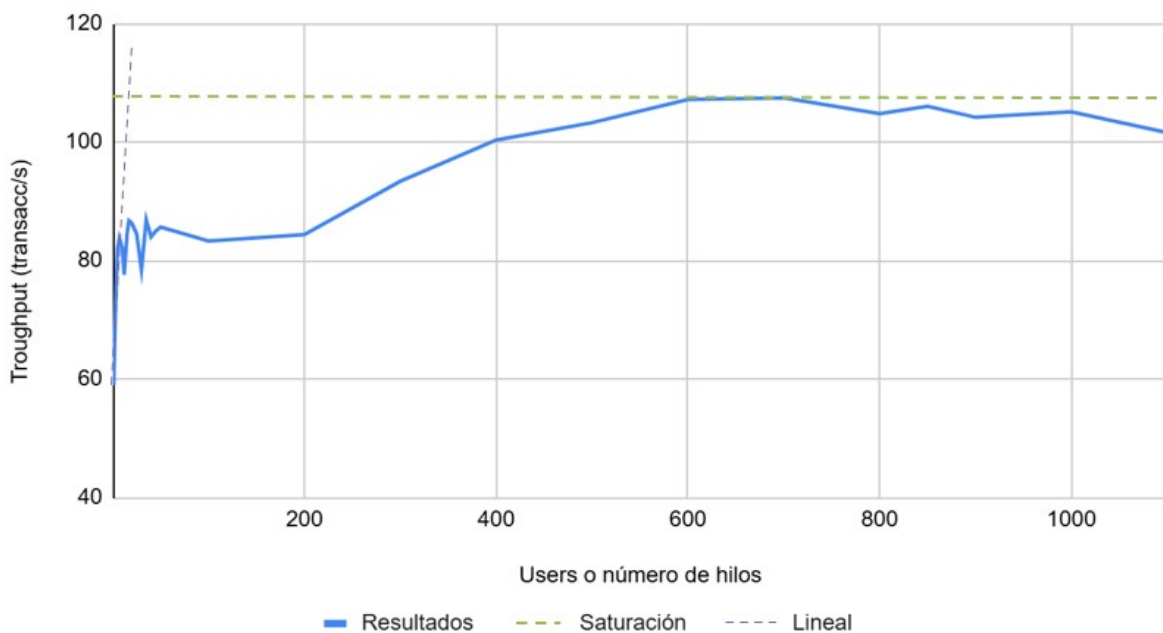
Configuramos todo como se nos dice y tomamos los tiempos uno a uno con el comando descrito en el enunciado, considerando la importancia de no ejecutarlo con la interfaz gráfica de Jmeter. Los resultados obtenidos para el proyecto P1-base con Unicorn y con 1 worker son los siguientes, mostrándose primero una gráfica con el límite de users a 1100 y luego otra con este límite a 50, para poder ver en esta última la fase lineal sin comprimirse tanto. Se puede apreciar que tras 200 users hasta los cuales parecía que ya se había estabilizado el throughput, vuleve a haber un crecimiento en el mismo. Esto creemos que se puede deber a que a partir de este punto empezaba a haber errores en las respuestas (lo vimos), con lo que aumentaba el throughput porque al devolver errores las respuestas que no se habían podido atender bien, Jmeter las considera para el número de respuestas devueltas por unidad de tiempo (throughput), y estas se devolvían más rápido que las peticiones procesadas correctamente, aumentando el throughput. Las medidas tomadas son las siguientes (hemos tomado medidas con números users muy altos de users para comprobar que no seguía creciendo el throughput, cosa que era errónea porque sí que aumentaba y que ahora se comentará):

Users o número de hilos	Throughput (transacc/s)
1	58,94
3	72,82
5	81,97
7	83,75
10	82,02
12	77,6
15	84,53
17	86,69
20	86,25
25	84,48
30	78,47
35	86,85

40	83,97
45	84,97
50	85,63
100	83,28
200	84,36
300	93,36
400	100,32
500	103,25
600	107,15
700	107,45
800	104,75
850	106,01
900	104,16
1000	105,07
1100	101,58

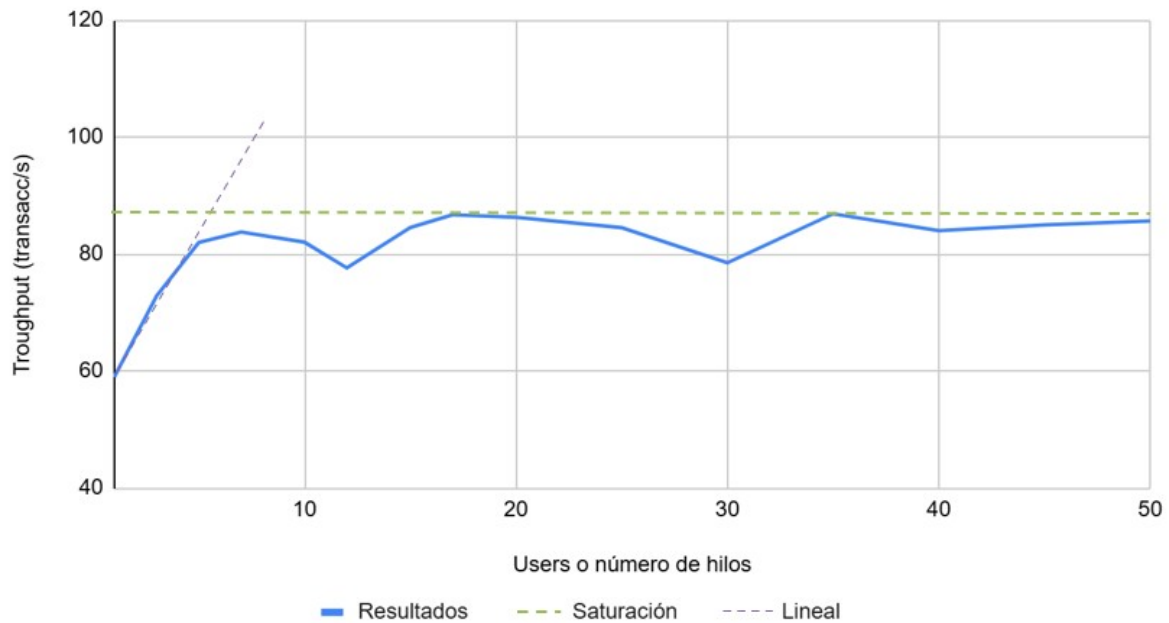
Como hemos dicho, por esto de que a partir de 200 users el throughput vuelve a crecer por los errores (creemos), parece que el punto de donde se cruzan la recta del punto de saturación y la de la fase lineal se encuentra en torno a 10 users y tiene un throughput de 107,45 transacciones por segundo (no es el resultado que tomaremos por lo de los errores y por su imprecisión).

Troughput vs usuarios, P1-base 1 worker unicorn



Si miramos la gráfica pero con un máximo de users de 50 para que no se vea el crecimiento de a partir de 200 users que hemos dicho (de 50 a 200 la recta se mantiene parecida, por eso no se muestra), parece que el punto de donde se cruzan la recta del punto de saturación y la de la fase lineal se encuentra en torno a 5 users y tiene un throughput de 86,85 transacciones por segundo. Estas medidas nos parece que son más acertadas de coger que las anteriores ya que aquí no había fallos en las peticiones (lo que hemos explicado).

Troughput vs usuarios, P1-base 1 worker gunicorn



Las gráficas de los otros dos proyectos no las hemos hecho debido a que no nos daba tiempo, pero funciona todo correctamente.

Ejercicio número 6:

¿Qué servidor es más eficiente gunicorn con una CPU o el servidor de desarrollo “python manage.py runserver 8000”? Calcula la curva de productividad para el caso “python manage.py runserver 8000” y comparala con la obtenida para gunicorn. Para realizar esta prueba tendrás que parar gunicorn y lanzar el servidor de desarrollo. Nota: Por defecto, “python manage.py runserver” solo es accesible desde el localhost (127.0.0.1). Para permitir conexiones desde otros dispositivos, usa “python manage.py runserver 0.0.0.0:8000”. Da una respuesta razonada e incluye en la memoria el resultado de las medidas realizadas para contestar esta pregunta.

Sabemos que el servidor de Django de desarrollo tiene un único hilo y worker.

Pramos el servidor de Gunicorn con `sudo systemctl stop gunicorn`, y corremos el servidor de desarrollo como se nos dice, con `python manage.py runserver 0.0.0.0:8000`.

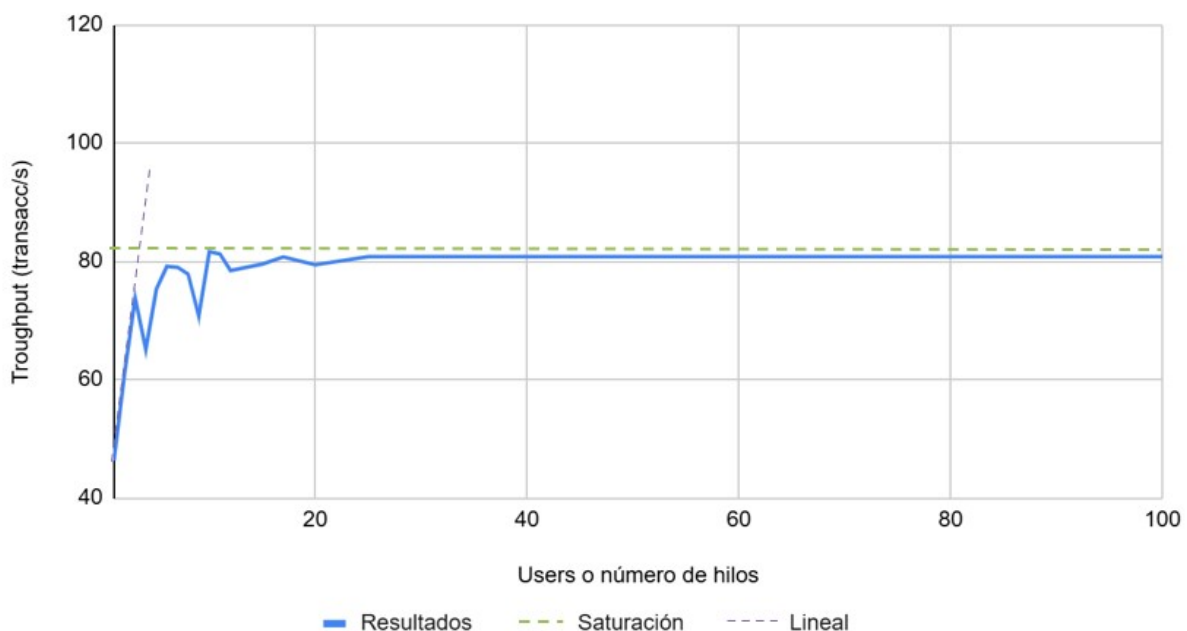
En este caso, también mostramos dos gráficas: una con el eje x ampliado y otro con la parte que más nos interesa. Cabe destacar, que con el servidor de django, a partir de 20 users, empezaba a quedarse colgado al hacer las pruebas, teniendo nosotros que pararlo con Ctrl+C, con lo que se nos dificultó tomar medidas altas (sólo fuimos capaces con 25 y con 100 users), aunque no nos preocupa porque parece que con los resultados obtenidos se puede apreciar que ya hemos llegado al punto de saturación (que se mantiene con 25 y con 100 users, lo que nos dio más confianza). Hay que mencionar que en este caso en ninguna medida hubo peticiones fallidas. Las medidas tomadas son las siguientes:

Users o número de hilos	Throughput (transacc/s)
1	46,26
2	61,07
3	73,68
4	65,06
5	75,26
6	79,1

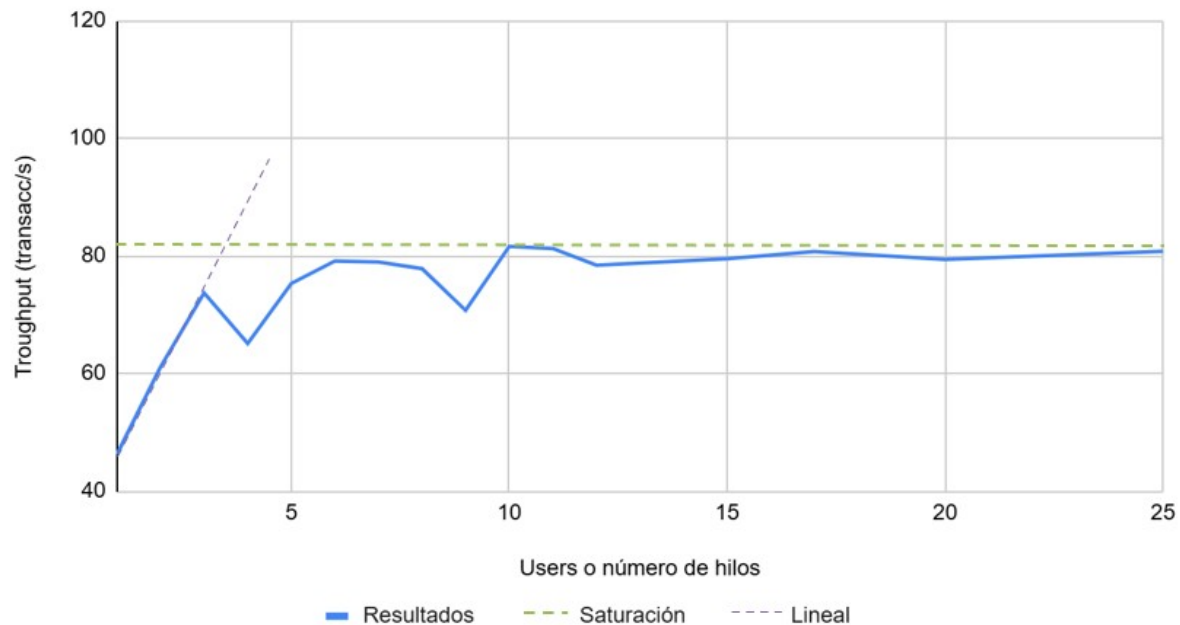
7	78,95
8	77,8
9	70,71
10	81,58
11	81,23
12	78,4
15	79,48
17	80,7
20	79,39
25	80,74
100	80,73

Se puede apreciar que, en ambos casos, parece que el punto de donde se cruzan la recta del punto de saturación y la de la fase lineal se encuentra en torno a 3 users y tiene un throughput de 81,58 transacciones por segundo, cosa que nos muestra que el servidor de Gunicorn es más eficiente ya que este de Django para desarrollo alcanza su punto de saturación con menos users y con menos transacciones por segundo (throughput); todo sin contar que a partir de 20 users empieza a quedarse colgado, como hemos dicho. Esto se debe a que claramente este no es un servidor pensado para la producción (es de desarrollo) y no es demasiado potente, mientras que el de Gunicorn es más potente al estar pensado para la fase de producción.

Troughput vs usuarios, P1-base 1 worker server django



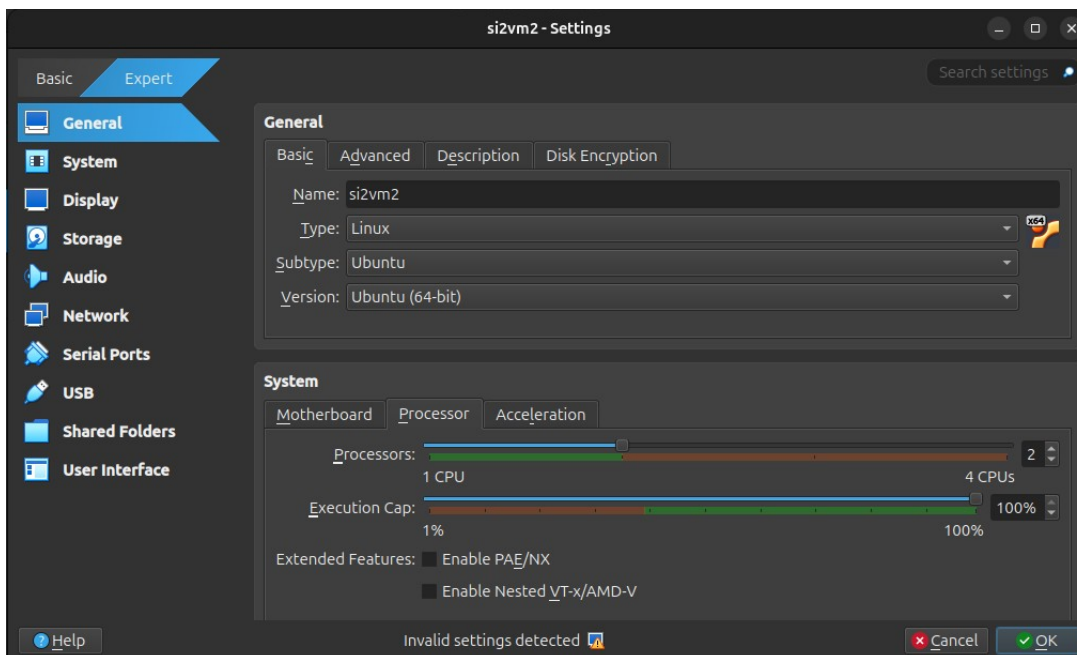
Troughput vs usuarios, P1-base 1 worker server django



Ejercicio número 7:

Calcula la curva de productividad de la aplicación P1-base usando gunicorn con 2 workers. Puesto que la maquina virtual está configurada para usar una única CPU incrementar el número de workers a dos no otorga a gunicorn más recursos. Modica la conguración de la máquina virtual para que tenga a su disposición 2CPUs en lugar de una. Incluye la curva calculada en la memoria y comenta el resultado. Nota: no olvides usar `SESSION_ENGINE=django.contrib.sessions.backends.db` en `settings.py` puesto que en caso contrario las variables de sesión pueden no ser recuperables.

Primero, en el fichero `/etc/systemd/system/gunicorn.service` ponemos workers a 2, continuamos poniendo `SESSION_ENGINE='django.contrib.sessions.backends.db'` en el `settings.py`, para finalmente irnos a la configuración de la VM2 (donde está el servidor) y poner processors a 2 en vez de a 1:



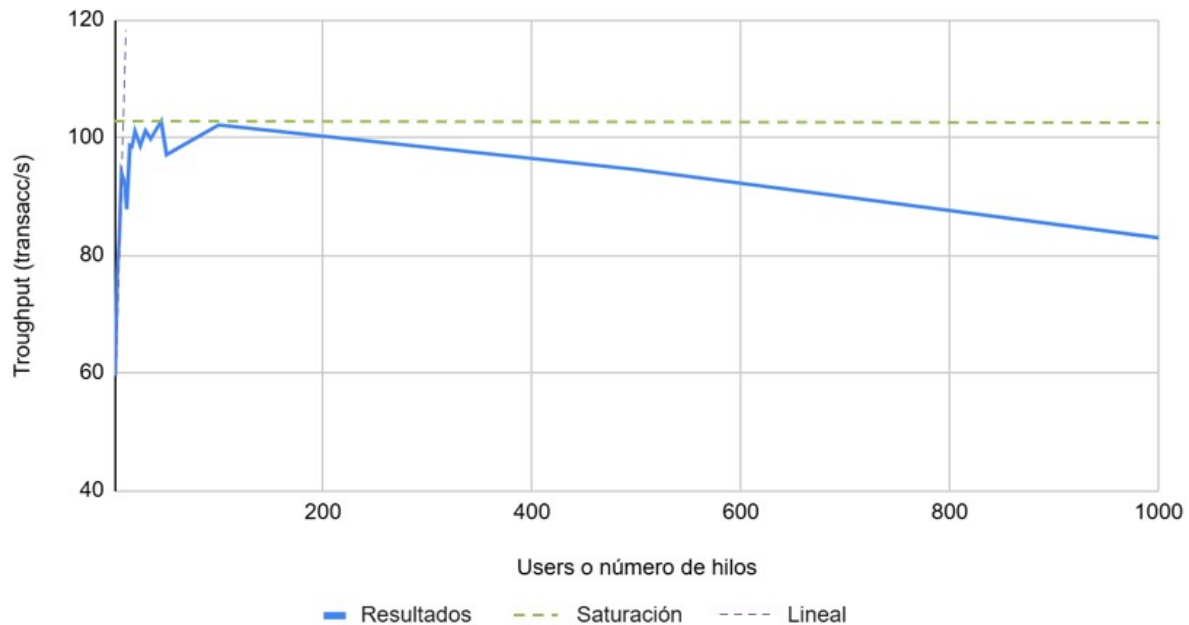
En este caso, también mostramos dos gráficas: una con el eje x ampliado y otro con la parte que más nos interesa. Hay que mencionar que en este caso en ninguna medida hubo peticiones fallidas (por esto el throughput disminuye a partir de 100 users, porque es la fase de degradación, pero sin devolver errores

como P1-base con Unicorn y 1 worker, en la que a partir de 200 users se empezaban a devolver errores y el throughput crecía). Las medidas tomadas son las siguientes (hemos tomado medidas con muchos users para comprobar que el throughput no seguía creciendo, viendo que desciende a partir de 100 users, pero no aumenta):

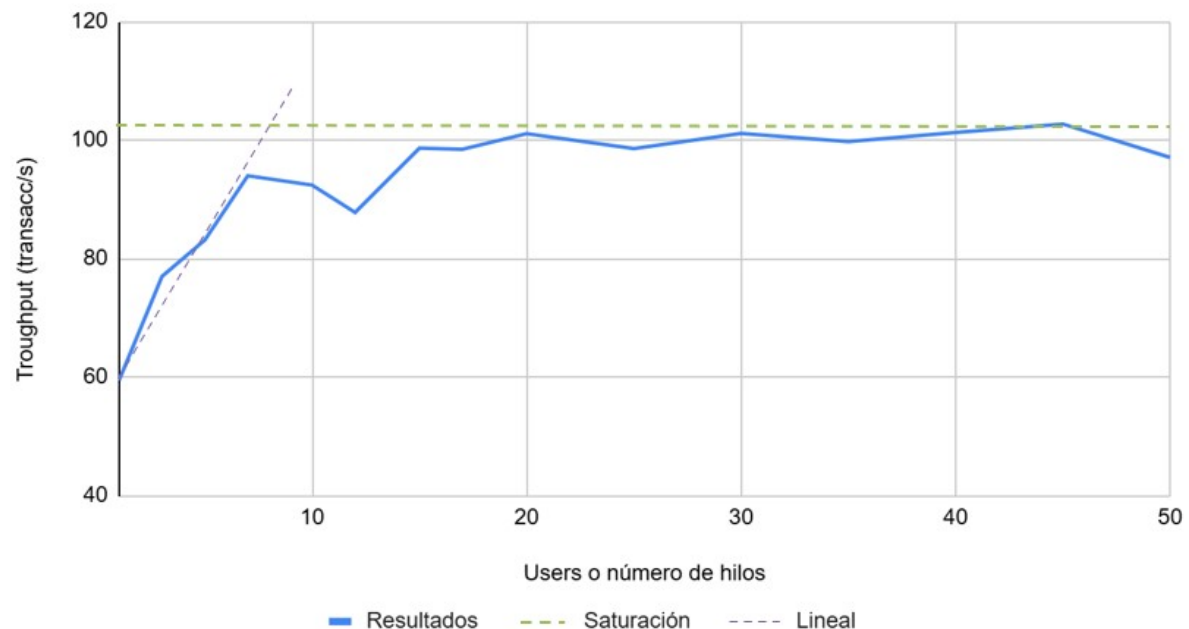
Users o número de hilos	Throughput (transacc/s)
1	59,46
3	76,99
5	83,13
7	93,96
10	92,39
12	87,77
15	98,61
17	98,41
20	101,05
25	98,55
30	101,1
35	99,73
40	101,27
45	102,69
50	97,05
100	102,09
500	94,51
1000	82,93

Se puede apreciar que, en ambos casos, parece que el punto de donde se cruzan la recta del punto de saturación y la de la fase lineal se encuentra en torno a 7 users y tiene un throughput de 102,69 transacciones por segundo, cosa que nos muestra que el servidor de Unicorn con 2 workers es más eficiente que con 1 worker ya que este alcanza su punto de saturación con más users y con más transacciones por segundo, además de que no devuelve errores donde con 1 worker sí que los devolvía (desde 200 hasta 1000 users). Esto es lógico ya que habrá más receptores de peticiones trabajando para servirlos con 2 workers que con 1.

Troughput vs usuarios, P1-base 2 worker gunicorn



Troughput vs usuarios, P1-base 2 worker gunicorn



Cuestión número 2:

Si repetimos el ejercicio 7 usando el proyecto P1-ws-client/master en lugar de P1-base, ¿en que fichero settings.py tenemos que modificar la variable `SESSION_ENGINE`? (a) en el del servidor, (b) en el del cliente, (c) en ambos, (d) en ninguno. Justica la respuesta.

Depende de si en el cliente ponemos 2 workers o 1 (en el servidor es indiferente). Si en el cliente sólo hay 1 worker, no será necesario en ningún lugar ya que el cliente guardará sus datos de sesión en una misma caché, y como se explicará ahora, este es el que gestiona todo el tema de las sesiones, no el servidor; haciendo que si sólo hay 1 worker sólo se acceda a una misma caché para guardar y leer estos datos de sesión, no habiendo problemas. En este caso, la respuesta, por tanto, sería la **(d) en ninguno**.

En caso de que el cliente tenga 2 workers, la respuesta es la **(b) en el del cliente** para que los datos de sesión que el cliente maneja (como ahora comentaremos), se guarden en un lugar de almacenamiento común para ambos workers, que es la base de datos en lugar de la caché (esta última es individual de cada worker). Esto se debe a que las API REST en teoría son stateless y no guardan datos de sesión, cosa que

sabemos que en este caso es cierta porque el único dato de sesión que manejamos en esta aplicación es que el id del censo verificado se envíe al registrar el voto para asignarlo al censo correspondiente, y sabemos que en la API REST el cliente envía explícitamente este censo al hacer el voto sin que la API REST se ocupe de nada de la sesión (de guardarlo entre peticiones). Es decir, el que se ocupa de mantener la sesión es el cliente, la API REST sólo atiende a lo que se le pida sin guardar ningún estado. Sin embargo, el hecho de poner `SESSION_ENGINE='django.contrib.sessions.backends.db'`, hace que el cliente trate de guardar los datos de la sesión en la base de datos, y al no tener podría fallar. Esto lo podríamos solucionar proporcionando al cliente la base de datos del servidor (base de datos de la VM1) o proporcionándole otra base de datos (ya sea la de sqlite de desarrollo o cualquiera otra). Tal cual lo tenemos hecho, ya de por sí le proporcionamos la base de dato de la VM1 para que los tests en anteriores prácticas funcionasen.