

PROGRAMACION DISTRIBUIDA Y TIEMPO REAL

TRABAJO PRACTICO N°2

CHILACA CASTRO, Sol Angela | 22277/2

KUMICHEL, Franco Leandro | 17773/0

1) Buscar y utilizar en vagrant una versión más reciente de Ubuntu LTS que la dada en la explicación de práctica. Identificar versión, origen y si tuviera diferencias, explicar brevemente. Mostrar con una captura de pantalla la identificación en VirtualBox de la máquina virtual y el inicio de la terminal en funcionamiento. Consulte a un asistente de IA (deje constancia de cuál) si hay alguna versión en particular que puede ser mejor que otra y por qué.

Investigando diversas fuentes y analizando el archivo vagrantfile provisto por la cátedra, se puede identificar lo siguiente acerca de la versión, origen y diferencias:

Versión

En el archivo vagrantfile utilizado en la explicación práctica se puede observar la línea `machine.vm.box = hashicorp/bionic64`, por lo que se puede inferir que las máquinas virtuales a crear serán con sistema operativo Ubuntu 18.04 LTS. Por otro lado, la versión LTS de Ubuntu más reciente es la 24.04 (que se suele llamar Noble Numbat).

Origen

La versión utilizada en el vagrantfile que dio la cátedra proviene de la [cloud de Hashicorp](#). Así mismo, para la versión más reciente de Ubuntu se puede utilizar la box provista por [Bento](#), el cual es un distribuidor recomendado por Hashicorp en su documentación. Para utilizarla, modificamos el vagrantfile de la siguiente forma: `machine.vm.box = bento/ubuntu-24.04`

Diferencias

Entre las dos versiones utilizadas, podemos citar las siguientes:

- **Soporte:** la versión 18.04 ya no posee soporte oficial, mientras que la versión 24.04 posee soporte hasta 2029.
- **Software:** la versión 24.04 utiliza versiones más nuevas de software, tales como gcc 13.x, python 3.12.3, en comparación a la versión 18.04, que utiliza gcc 7.x, python 3.6.
- **Kernel:** la versión 24.04 utiliza una versión de kernel linux 6.8, mientras que la versión 18.04 utiliza una versión de kernel linux 4.15. Utilizar la versión más actual beneficia en optimización en rendimiento, seguridad, soporte de hardware moderno y nuevas características.

Al consultar a *ChatGPT-5* acerca de que si hay alguna versión en particular que puede ser mejor que otra y por qué, respondió lo siguiente:

“La elección entre las dos LTS y sus boxes (hashicorp/bionic64 con Ubuntu 18.04 vs bento/ubuntu-24.04 con Ubuntu 24.04) depende de lo que más priorices: compatibilidad y simplicidad vs. modernidad y seguridad.

Vamos a desglosarlo 📌

🔍 Comparación general

Aspecto	Ubuntu 18.04 LTS (hashicorp/bionic64)	Ubuntu 24.04 LTS (bento/ubuntu-24.04)
Año de lanzamiento	2018	2024
Kernel por defecto	4.15 (LTS)	~6.8 (mucho más moderno)
Estado de soporte	ESM (Extended Security Maintenance, requiere suscripción) → fin de soporte público	Soporte activo hasta 2029
Compatibilidad software legacy	Alta (para paquetes y código C más antiguo)	Puede requerir ajustes para código muy viejo
Rendimiento red y CPU	Correcto pero con pila de red y planificador más antiguos	Mejoras importantes en TCP/IP, virtualización, uso de múltiples núcleos
Seguridad	Solo con parches ESM; sin nuevas mitigaciones	Seguridad reforzada, mitigaciones actualizadas, soporte continuo
Ecosistema de paquetes	Versión congelada (glibc, gcc antiguos)	Paquetes más modernos, mejor soporte para nuevas bibliotecas
Peso de la box	Menor, más liviana	Algo más pesada (kernel y paquetes más recientes)

En el marco de tu trabajo


Tu proyecto no depende de frameworks ni librerías muy recientes: es básicamente C, sockets, scripts bash. Lo que más te interesa es:

- Compatibilidad de compilación (gcc)
- Comportamiento de red (latencias, throughput)
- Reproducibilidad de resultados entre cliente y servidor

Recomendación

✓ Si buscás estabilidad, reproducibilidad con ejemplos vistos en clase o entornos académicos más antiguos → hashicorp/bionic64 (Ubuntu 18.04)

- Más sencillo si los docentes ya usaron esa versión.
- Código C clásico para sockets funciona sin cambios.
- Más liviano, arranca rápido en Vagrant.
- Útil si querés aislar tu experimento de optimizaciones modernas de la pila de red.

 Si buscás un entorno más realista, seguro y alineado con infra actual → bento/ubuntu-24.04

- Mejor rendimiento en pruebas de red, menos jitter en las mediciones.
- Mejor soporte de hardware/virtualización, ideal para VMs sobre hosts modernos.
- Más seguro y con soporte oficial por varios años.
- Puede requerir ajustar opciones de compilación si el código usa APIs obsoletas.

Conclusión clara

Para un laboratorio académico enfocado en reproducir ejercicios clásicos, Ubuntu 18.04 (bionic64) suele ser suficiente y más simple. Para un trabajo que busque resultados más cercanos a un entorno real actual, con mejor rendimiento y seguridad, Ubuntu 24.04 (bento) es la mejor opción.

En otras palabras:

- Si tu objetivo es compatibilidad con material de clase y cero sorpresas → Ubuntu 18.04.
- Si tu objetivo es realismo y futuro → Ubuntu 24.04."

2) Ejecutar los experimentos del ejercicio 4 a), b) y c) de la práctica anterior en dos computadoras físicamente diferentes. Se copia aquí el enunciado para

mayor comodidad: comunicar 10^1 , 10^2 , 10^3 , 10^4 , 10^5 y 10^6 bytes desarrollando experimentos que muestre el tiempo que toma en el cliente

a) La función `write(...)` para cada cantidad de datos. Tomando C como ejemplo:

`t0 = ...`

`write(...)`

`t1 = ...`

`tiempo = t1 - t0;`

b) La función `read(...)`. Tomando C como ejemplo:

`t0 = ...`

`n = read(sockfd, buffer, 255);`

`t1 = ... tiempo = t1 - t0;`

c) Grafique y explique los resultados obtenidos. En particular, identifique si las diferencias de tiempos son proporcionales a las cantidades de datos, por ej: ¿`write(...)` con 1000 bytes toma 10 veces más tiempo que con 100 bytes? Identifique si el tiempo de la función `read()` se mantiene constante, dado que involucra siempre la misma cantidad de datos.

Para la realización del experimento utilizamos 2 computadoras físicamente diferentes en una misma red local. Las especificaciones de las máquinas utilizadas son las siguientes:

PC de escritorio: CPU Intel Core i5-3330S (4) @ 2.70 GHz, RAM 8 GB, OS Ubuntu 24.04

Ambas máquinas tienen las mismas especificaciones mencionadas. Cabe destacar que para la automatización del experimento, se confeccionaron los scripts `run_client.sh` y `run_server.sh` para levantar cliente y servidor respectivamente que tienen un bucle para ejecutar para los distintos tamaños de mensajes solicitados en el enunciado. Luego de aquí se recolectan los resultados para volcarlos en una tabla y luego poder graficarlos. Además, en los scripts `client_prueba.c` y `server_prueba.c` realizamos varias iteraciones por cada potencia a modo de tener en cuenta las variaciones que puedan darse en la comunicación. Para la obtención de los tiempos finales, se realizó un promedio de los tiempos tomados de cada iteración por cada potencia. A continuación se dejan disponibles las tablas y gráficos confeccionados:

Cantidad de bytes	Promedio Write (ms)	Promedio Read (ms)
10	0.036151	8.891249
100	0.036459	13.139372
1,000	0.037549	10.531111
10,000	0.051548	14.479394
100,000	0.232546	90.086052
1,000,000	5.271850	373.750718

Tabla 1: Tiempos promedios de comunicación (ms) para 100 experimentos



Figura 1: Tiempos promedios de comunicación write (ms) vs tamaño de mensaje

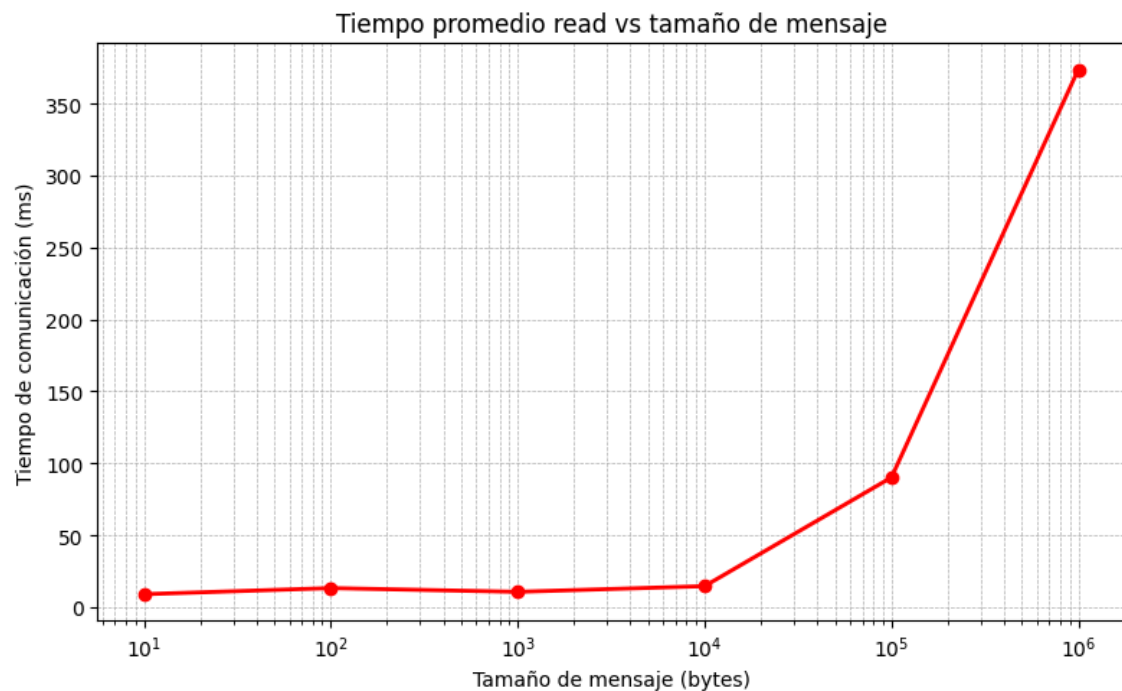


Figura 2: Tiempos promedios de comunicación read (ms) vs tamaño de mensaje

Observando la tabla 1 y los gráficos 1 y 2, podemos observar que para tamaños más pequeños (hasta 10000 bytes), tanto para el write como para el read, el tiempo en general es casi constante, recién a partir de tamaños más grandes (mayor a 100000 bytes) el tiempo empieza a crecer de forma notable. Esto muestra que los tiempos de ambas operaciones no son proporcionales al tamaño de los datos, sólo empiezan a aumentar cuando la cantidad de datos a transferir son lo suficientemente grandes.

3) Desarrollar los mismos experimentos de comunicaciones que en el caso anterior, con las mismas computadoras, pero ahora con la misma cantidad de datos en un sentido y en el inverso (experimento “ping-pong”). El tiempo total “de ida y vuelta” de los datos dividido por 2 es el que se utiliza para una estimación del tiempo de mensajes en una dirección. Comparar los tiempos del ejercicio anterior con los que se obtienen en este experimento para cada cantidad de bytes.

Para este experimento confeccionamos los códigos *client_pingpong.c* para el cliente y *server_pingpong.c* para el servidor. A comparación del experimento anterior, tuvimos en cuenta que el servidor devuelva la cantidad de datos procesados y que luego el cliente reciba estos mismos datos. A continuación se presentan los resultados en formato de tabla y gráfico:

Modelo	Tamaño (bytes)	Tiempo total (ms)	Tiempo one way side (ms)
Cliente	10	4.388485	2.194242
Cliente	100	6.932294	3.466147
Cliente	1000	10.821173	5.410587
Cliente	10000	10.140636	5.070318
Cliente	100000	98.768530	49.384265
Cliente	1000000	927.020380	463.510190

Tabla 2: Tiempos totales y tiempos one way side promedios (ms) para 100 experimentos

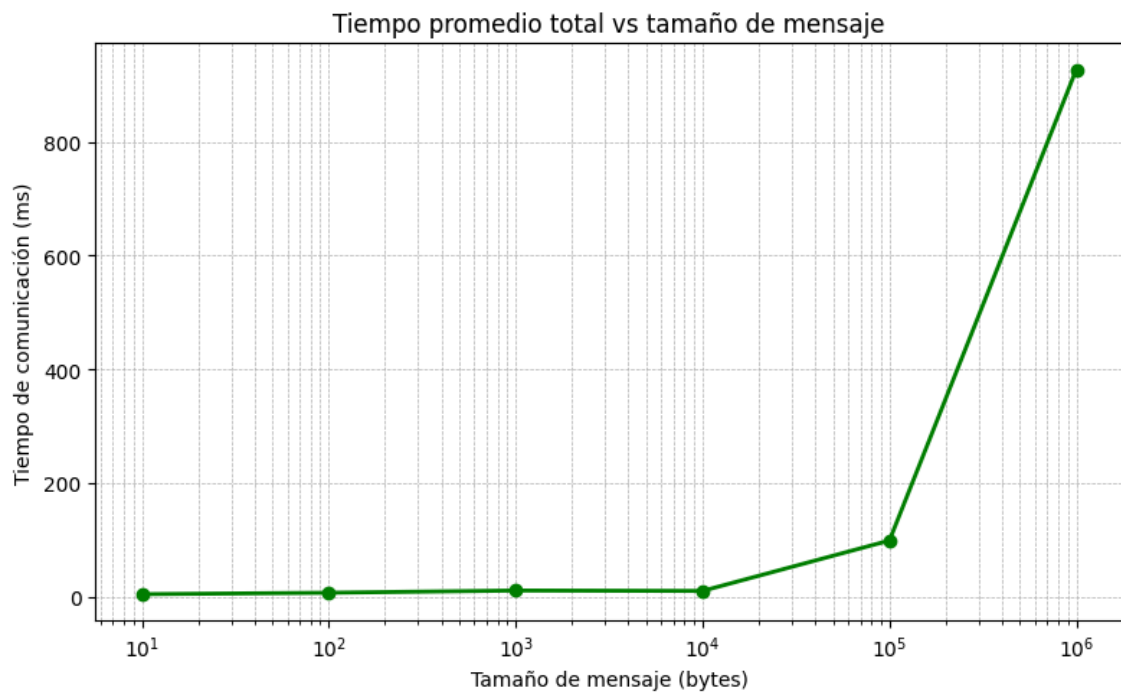


Figura 3: Tiempo promedio total vs tamaño del mensaje

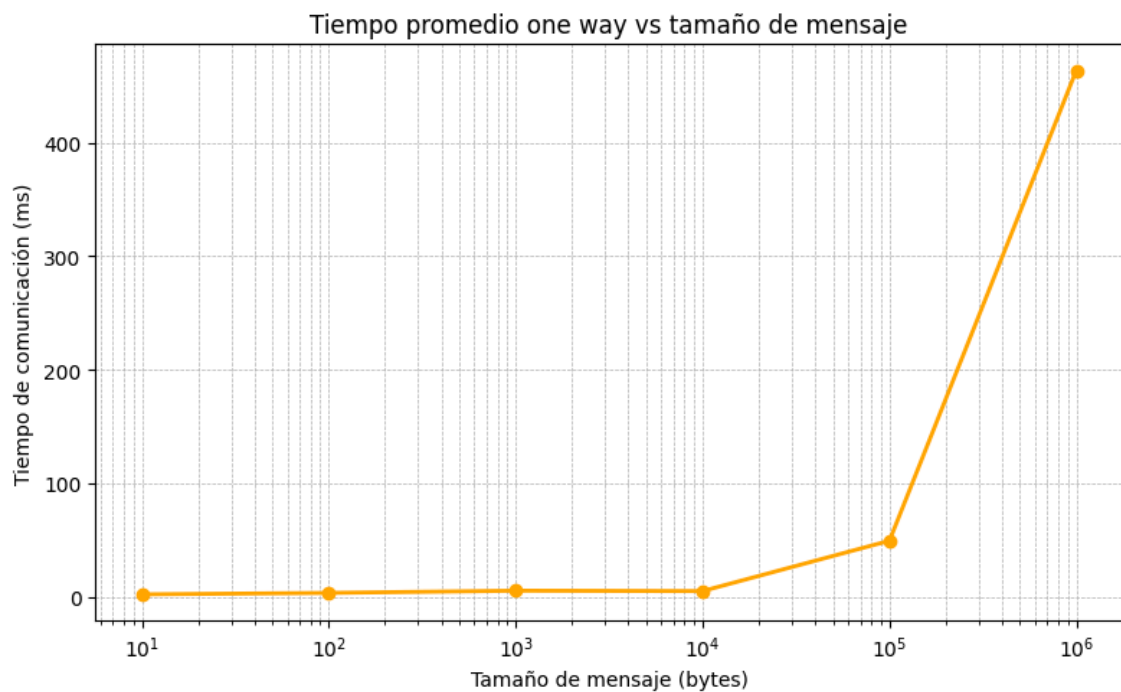


Figura 4: Tiempo promedio one way vs tamaño del mensaje

Al observar la Tabla 2, se nota que los tiempos obtenidos son mayores que los registrados en el Ejercicio 2. Además, las Figuras 3 y 4 muestran que los tiempos

de comunicación mantienen un patrón similar a medida que aumenta el tamaño del mensaje.

Estos resultados evidencian que al tener en cuenta el sentido de la comunicación inversa, es decir el envío de datos desde el servidor hacia el cliente, el tiempo total se incrementa. Esto se debe a que el servidor, al realizar las operaciones de write, debe manejar volúmenes de datos más grandes, mientras que el cliente debe procesar una mayor cantidad de lecturas, lo que introduce un sobre costo adicional en la comunicación.

4) Teniendo en cuenta los experimentos de tiempos realizados, desarrollar scripts para desplegar un ambiente de experimentación de comunicaciones en una computadora con Vagrant para los siguientes escenarios:

a- Dos máquinas virtuales, cada una con un proceso de comunicaciones.

b- Una máquina virtual con uno de los procesos de comunicaciones y el otro proceso de comunicaciones en el host.

En todos los casos deberían quedar los resultados disponibles para su posterior análisis. Se debe resolver el problema de "asincronismo" que genera un error si el proceso "cliente" inicia la ejecución antes que el proceso "servidor". Consultar con un asistente con IA cuál es sería el mejor método experimental para estimar el tiempo de comunicaciones entre dos computadoras diferentes y comente si le parece correcto o puede identificar algún problema en la respuesta del asistente con IA

Para llevar a cabo las experimentaciones se tomaron en cuenta los códigos desarrollados en los ejercicios 2 y 3, donde se tuvo en cuenta para este caso:

- Utilizamos la box provista en la explicación práctica, [hashicorp/bionic64](https://github.com/hashicorp/bionic64), dado que para los fines de estos experimentos resulta adecuada.
- En nuestro caso utilizamos una configuración de red privada ya que las interacciones serán solo entre VM's o VM y host.

Por otro lado, para resolver el problema del asincronismo, tanto para el caso a) como para el caso b) los scripts usados tienen la característica que se quedan consultando si el puerto al que se quieren comunicar se encuentra en estado LISTEN.

```
until nc -z $SERVER_IP 8080; do
    echo "Esperando a que el servidor esté en estado LISTEN..."
    sleep 1
done
```

Bajo el comando `nc -z` es posible hacer una espera activa para revisar si el servidor se encuentra escuchando en puerto 8080 (definido así de forma arbitraria) de forma que el bucle se repite hasta que la condición nos indique un `exit code 0`. De esta manera, el cliente inicia la ejecución una vez que el servidor esté en estado de LISTEN.

A continuación, mostraremos los resultados obtenidos

Cantidad de bytes	Promedio write (ms)	Promedio read (ms)
10	0.868278	1.802449
100	0.880778	1.648390
1000	0.769153	1.432331
10000	0.902178	2.387447
100000	2.917345	3.661835
1000000	15.919139	6.808372

Tabla 3: Tiempos promedios de comunicación write y read (ms) para 100 experimentos

Cantidad de bytes	Promedio total (ms)	Promedio one way (ms)
10	1.275058	0.637529
100	1.055274	0.527637
1000	0.973709	0.486854
10000	1.813757	0.906879
100000	3.202596	1.601298
1000000	16.212451	8.106226

Tabla 4: Tiempos promedios de comunicación totales y one way (ms) para caso con dos máquinas virtuales

Sobre los métodos para estimar el tiempo de comunicaciones

Según las IAs consultadas (Chat GPT-5, Gemini 2.5 Flash y Claude Sonnet 3.5) coinciden en que el mejor método experimental más sencillo de aplicar es el Round-Trip Time (por simplicidad, RTT) que consiste en medir el tiempo (generalmente en milisegundos) que tarda un paquete de datos en viajar desde la computadora de origen al destino. El procedimiento sugerido es el siguiente:

1. Realizar múltiples mediciones.
2. Probar con diferentes tamaños de mensajes para evaluar la latencia y cuellos de botella si existiesen.
3. Tener en cuenta factores externos para identificar valores atípicos (como la carga de red, tiempo del sistema operativo, etc)
4. Realizar el análisis en el que se calculan los promedios, mediana y desviación estándar.

La herramienta clásica provista tanto por Linux como Windows es mediante el uso de `ping <server_ip>`.

```
ping 8.8.8.8
Haciendo ping a 8.8.8.8 con 32 bytes de datos:
Respuesta desde 8.8.8.8: bytes=32 tiempo=57ms TTL=118
#... más pruebas
Estadísticas de ping para 8.8.8.8:
    Paquetes: enviados = 4, recibidos = 4, perdidos = 0
    (0% perdidos),
Tiempos aproximados de ida y vuelta en milisegundos:
    Mínimo = 47ms, Máximo = 242ms, Media = 142ms
```

Ejemplo de uso del comando ping para 4 repeticiones.

La limitación de esta forma es que asume el camino de ida y el camino de vuelta tienen el mismo retardo por eso también se ofrece el cálculo del **OWD** que sería el *Retardo Unidireccional*: mide el tiempo en un solo sentido, esencial para aplicaciones que dependen de la entrega en tiempo real y secuencial. Este método sin embargo es más difícil de aplicar porque requiere sincronización de relojes de las computadoras de forma precisa.

También se nos ofrecen herramientas como `iperf3` en el que por comandos se indica cuál computadora actuará como servidor y cual como cliente en el que se automatiza este proceso y proporciona estadísticas finales. Por parte del cliente se envía un gran volumen de datos al servidor y se mide la velocidad real de

transferencia en **Mbps** o **Gbps**. Ideal para saber el **rendimiento máximo** que la conexión puede sostener.

Las respuestas ofrecidas por las IAs coinciden más no hacen hincapié en el retardo que se pueden generar por factores como el propio hardware de las computadoras involucradas, el ambiente en que se despliegan (sean nativas o virtualizadas), el ancho de banda y el tráfico de información que se genera de forma implícita (aplicaciones background). Tampoco se hace sugestión de métodos para evaluar el tiempo de startup de las comunicaciones (caso de realizar un experimento pingpong con carga 0) o de la realización de los experimentos en diferentes periodos para que la carga de la red se mantenga constante.

5) Explique si los resultados de tiempo de comunicaciones del experimento del ej. anterior se ven afectados por el orden de ejecución y las diferencias de tiempo de ejecución iniciales de los programas utilizados ¿qué sucede si por alguna razón hay una diferencia de 10 segundos en el inicio de la ejecución entre un programa y otro? Sugerencia: incluya un `sleep` de 10 segundos entre la ejecución de diferentes partes de los programas y comente.

En los algoritmos proporcionados, el cálculo del tiempo (`t0` y `t1`) envuelve únicamente las operaciones de comunicación (`write` y `read`). Si los experimentos realizados son ejecutados desde el código bash propuesto se contempla el caso de la ejecución del cliente antes que el servidor, por lo tanto, el cliente no intentará ejecutar `connect` antes que el servidor se encuentre listo. Para el caso en que se ejecuten directamente los códigos fuente, el cliente lanzará una excepción en el que se comunica que la comunicación fue rechazada. De todas formas, la sincronización en ambos programas se llevará a cabo luego del `accept` (del lado del servidor) y el `connect` (del lado del cliente). Se modificará entonces este caso como más significativo para ver el impacto en colocar un `sleep` entre estas instrucciones.