# Práctica 1: Conceptos y Mecanismos Básicos

Pedro Manuel Chaves Muniesa, 844390
Beatriz Emanuela Fetita, 840269

Universidad de Zaragoza

Sistemas Distribuidos-
Grado Ingeniería Informática

23/09/2025

# Descripción del Problema

La aplicación que se va a utilizar para esta práctica consiste en una única tarea que recibe una entrada, proporciona una salida y termina. Siendo más concretos, la aplicación busca los números primos dentro de un intervalo dado como argumento. Por la naturaleza de la app, será intensiva en CPU puesto que no requiere de gran capacidad de red de comunicación y mucho menos de almacenamiento. Para ello contaremos con un **cliente** que nos proporcionan terminado que envía peticiones / tareas con un intervalo que siempre es el mismo [1000, 70000]. Simplemente lo invocaremos con distintos parámetros para que genere cargas de trabajo diferentes. También tenemos un **servidor** que habrá que completar ya que contiene las funciones *IsPrime* y *FindPrimes*. La primera determina si un número entero dado es primo o no mientras que la segunda toma como entrada un intervalo y devuelve un array con todos los números primos en el intervalo.

# Análisis de Prestaciones de la Red

En este apartado se van a medir los tiempos de respuesta asociados a los cliente-servidor que nos son proporcionados y se tratará de extraer conclusiones. Para ello se ha tomado una muestra de tiempos para cada uno de los casos definidos en el **Apartado 2.1** del enunciado. Los datos que aparecen en las tablas 2.1 y 2.2 se corresponden a las medias aritméticas de la ejecución del escenario 10 veces.
Se puede apreciar que el tiempo tiende a ser menor cuando la máquina se despliega en una misma máquina, probablemente debido al tiempo extra que implica trabajar con recursos de red. Como sospechábamos, UDP es notablemente más rápido ya que no requiere de un proceso de "Handshake" previo, ni dispone de medidas de control y seguridad para la pérdida de paquetes a diferencia de TCP que de esta forma introduce ciertos retardos temporales.

| despliegue / resultado | **Fallo de Conexión** | **Conexión Correcta** |
|---|---|---|
| **Misma Máquina** | 658,917 µs | 1,1319 ms |
| **Distinta Máquina** | 755,1397 µs | 1,1991 ms |

***Tabla 2.1:*** *Tiempos de respuesta TCP*

| despliegue / resultado | **Fallo de Conexión** | **Conexión Correcta** |
|---|---|---|
| **Misma Máquina** | - | 448,962 µs |
| **Distinta Máquina** | - | 763,288 µs |

***Tabla 2.2:*** *Tiempos de respuesta UDP*

En esta red, probando la diferencia entre la ejecución local del algoritmo empleado en la práctica y la ejecución mediante cliente-servidor, se puede observar que el tiempo de transmisión es de un orden menor que el tiempo de ejecución de la función a ejecutar, por lo que podríamos considerar esta latencia despreciable, aspecto que nos interesa conseguir en programas que sean intensivos en CPU.

# Sincronización mediante Barrera distribuída

El objetivo principal de la barrera distribuida diseñada en esta práctica es el de asegurar que llegado cierto punto común del código, todos los procesos involucrados se informarán mutuamente, quedando a su vez bloqueados hasta recibir el mensaje homólogo de los mismos. Esto nos sirve para **sincronizar** un sistema en su inicialización de forma que todos los procesos esperarán a que estén activos el resto de procesos.
A nivel de implementación, cada nodo invoca una Gorutina *"Listener"* encargada de aceptar conexiones TCP de cada uno de los involucrados. Estos serán los canales por los que se recibirán los mensajes de llegada a la barrera *"Checkpoint"*. Una vez el *"Listener"* haya recibido todos los mensajes esperados, desbloqueará la barrera enviando un mensaje al método **main** a través de un canal síncrono de naturaleza bloqueante.

Por otro lado, en cada nodo se invoca una Gorutina por cada uno de los demás (N-1) procesos involucrados *"Senders"*. Cuando todos los *"Senders"* de un nodo hayan cumplido su misión, se escribirá a través de un canal síncrono para desbloquear el proceso **main**.

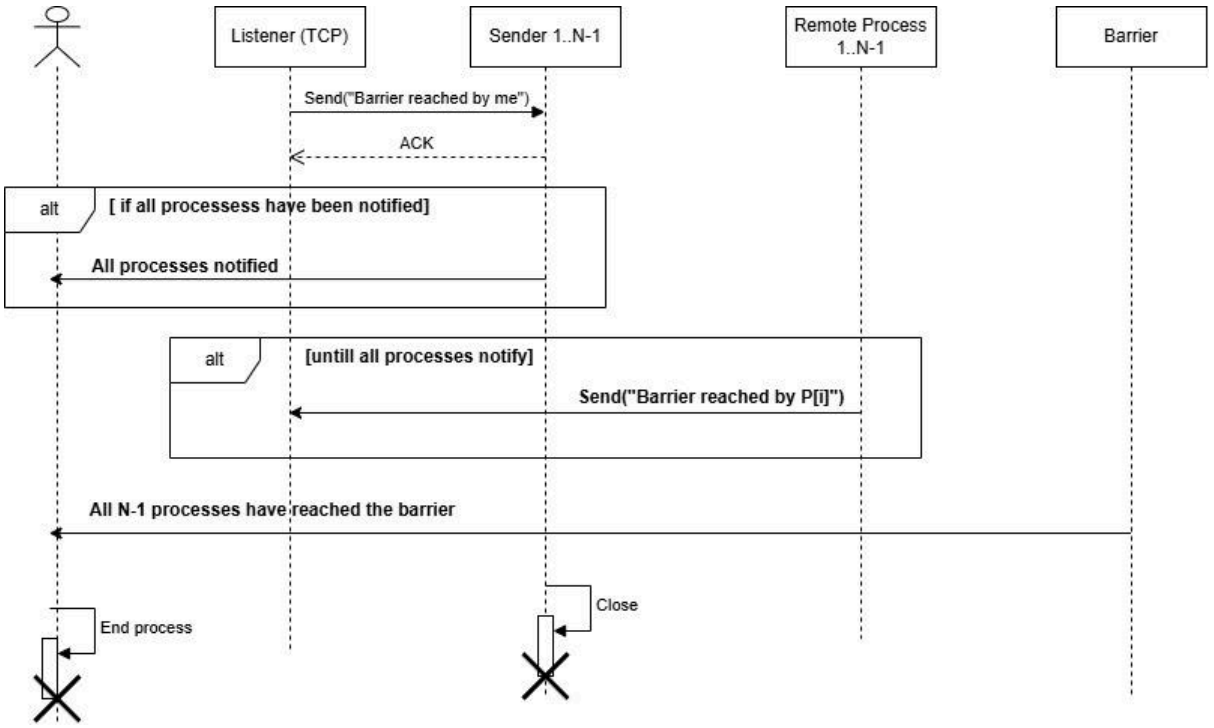


***Figura 3.1:*** *Diagrama de secuencia barrera distribuida*

# Diseño de las Arquitecturas

**Cliente-Servidor**

Esta arquitectura es una de las más utilizadas en Sistemas Distribuidos. Esta organización consta de un proceso servidor, que contiene la mayor parte de la funcionalidad, y un conjunto de procesos clientes que solicitan al servidor esta funcionalidad mediante el ya mencionado intercambio de mensajes. Existen diferentes variantes de esta arquitectura así que nosotros nos centraremos en implementar las siguientes variantes:

- cliente-servidor concurrente con un pool fijo de Goroutines
- máster-worker: En este caso, el máster arrancará los workers antes de operar mediante el uso del comando *ssh* (a partir de un fichero de texto plano que contiene un listado de máquinas o Endpoints ( IP + puerto -> 192.168.3.x:3125y) donde lanzar los workers.

**Cliente-Servidor secuencial**

Esta arquitectura es la más simple de todas, consiste en un servidor que atiende pe ticiones de forma secuencial, de una en una, de manera que cuando llegan varias peticiones, atiende una de ellas (a menudo la primera en llegar) y, una vez terminada, atiende la siguiente.

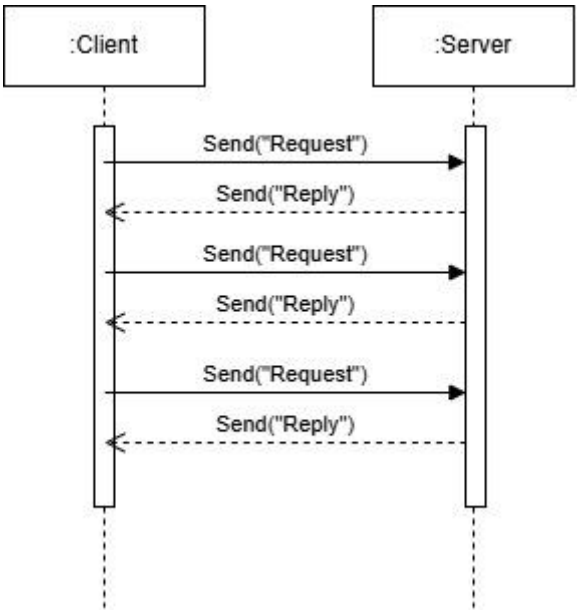


***Figura 4.1:*** *Diagrama de secuencia Cliente-Servidor secuencial*

**Cliente-Servidor concurrente con una GoRoutine por petición**

En este caso, se ha modificado el servidor, para que cada vez que llegue una petición de un cliente, lance una Goroutine que procese la petición de forma concurrente. Para ello únicamente es necesario añadir el ’go’ delante de la función que procesa la petición.

Un servidor concurrente que crea una Goroutine por cada petición del cliente permite procesar múltiples solicitudes simultáneamente, mejorando la concurrencia y evitando bloqueos al manejar una sola solicitud a la vez, como en el modelo secuencial. Esto resulta en una mayor escalabilidad, ya que las Goroutines son ligeras y pueden gestionar numerosas conexiones concurrentes de forma eficiente, mientras que un servidor secuencial procesa las peticiones de forma bloqueante y en orden, causando retrasos a medida que aumenta el número de clientes.

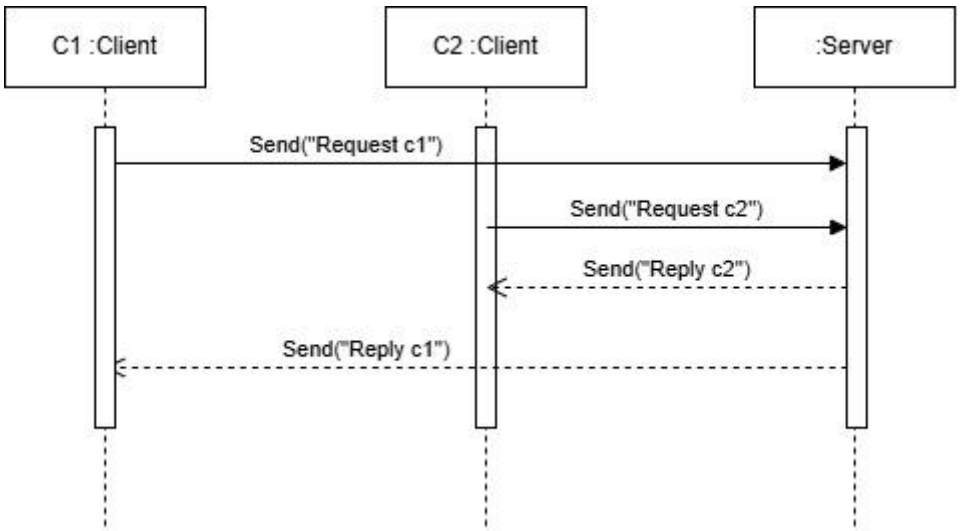


***Figura 4.2:*** *Diagrama de secuencia Cliente-Servidor concurrente*

**Cliente-Servidor concurrente con una pool fija de GoRoutines**

El uso de un pool fijo de Goroutines para procesar las peticiones de los clientes ofrece una serie de ventajas significativas frente a los modelos mencionados anteriormente. Comparado con el modelo concurrente sin límites, en el que se crea una Goroutine por cada solicitud, un pool fijo controla el uso de recursos, evitando la creación excesiva de Goroutines y el riesgo de sobrecargar el sistema. Esto garantiza una mejor eficiencia y estabilidad, evitando caídas por agotamiento de recursos en escenarios de alta carga, al tiempo que mantiene un equilibrio entre rendimiento y consumo de recursos.

La única diferencia respecto a la Goruotine por petición, es que aquí, previamente a escuchar peticiones de los clientes, se crea un número x de Goroutines, que son las que se encargan de escuchar las peticiones y procesarlas.

**Master-Worker**

La arquitectura Master-Worker se basa en un esquema donde el Master distribuye tareas entre múltiples Workers, quienes procesan las tareas de manera concurrente y autónoma. El Master recibe las tareas, las divide en subtareas más pequeñas y las asigna a los Workers, que las procesan independientemente. Finalmente, el Master recopila y combina los resultados. En comparación con los modelos anteriores, el Master-Worker ofrece una distribución más organizada y controlada de las tareas, a diferencia del modelo concurrente con Goroutines ilimitadas, donde las solicitudes simplemente generan una Goroutine sin una planificación explícita. Además, permite limitar el número de Workers, lo que proporciona un mejor control de recursos, similar al pool fijo de Goroutines, evitando la sobrecarga del sistema.

Un beneficio clave de la arquitectura Master-Worker es que cada Worker puede ejecutarse en una CPU diferente, lo que permite aprovechar los procesadores multinúcleo de manera eficiente. Esto maximiza el rendimiento al distribuir las tareas entre múltiples núcleos, mejorando significativamente el paralelismo y el tiempo de procesamiento. Comparado con el servidor concurrente, que crea Goroutines sin control explícito sobre los recursos físicos, el modelo Master-Worker optimiza el uso de las CPUs, aumentando la escalabilidad y eficiencia en entornos con alta carga de trabajo, especialmente cuando las tareas son intensivas en procesamiento.

El cálculo de los números primos ahora lo realizan los workers, que una vez están activos, avisan al máster para que sepa que ya puede empezar a enviarles tareas. El master divide la tarea en subtareas, y se las envía a los workers, y conforme van finalizando la subtarea, les envía otra. Ahora, es el master

el que distribuye las tareas entre los distintos workers, y como se puede observar en el diagrama de secuencias, el master env´ıa tareas a todos los workers para que las procesen de forma independiente, y le envíen únicamente la respuesta de cada uno y el master sea quien combine las respuestas.

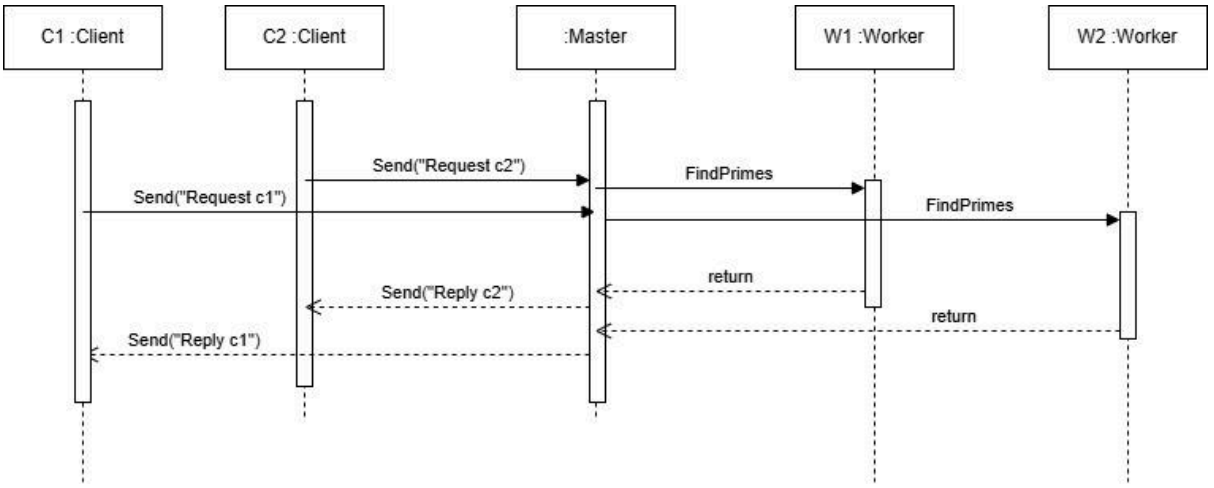


***Figura 4.3:*** *Diagrama de secuencia Master-Worker*