



**UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA**

Sistemas Distribuidos

Informe Laboratorio I

Integrantes: Hernán Aravena y Matías Barolo
Profesor: Manuel Ignacio Manríquez
Ayudante: Ariel Madariaga
Sección: A-1

Santiago – Chile
1-2024

ÍNDICE DE CONTENIDOS

CAPÍTULO 1. Introducción	3
CAPÍTULO 2. Diseño de Solución.....	4
CAPÍTULO 3. Metodología.....	4
CAPÍTULO 4. Resultados y Discusión.....	5
CAPÍTULO 5. Conclusión.....	6

I. INTRODUCCIÓN

Este trabajo consiste en la aplicación e implementación de tres tipos de arquitecturas de sistemas, en los cuales se busca realizar una comparación de resultados, tiempo y rendimiento, para analizar y comprender cómo funciona y cómo se trabaja en una arquitectura de sistema ya sea distribuida o centralizada.

Se trabajará con un programa y este será implementado en los tres sistemas. Todos estos serán puestos en efecto en el lenguaje de programación Python y en uno de ellos se trabajará con la librería mpi4py o MPI.

El programa principal está basado en “The One Billion Row Challenge” el cual es un programa/desafío usado para medir la rapidez de códigos en el lenguaje de programación Java.

La idea general del programa es que se tiene un archivo de texto .txt el cual contiene una lista con información de estaciones meteorológicas y sus respectivas temperaturas con el siguiente formato:

<estación;temperatura>

Donde, primero se debe de leer este archivo, luego se deben procesar las temperaturas y por último se requiere mostrar cada estación junto con su temperatura mínima, máxima y su temperatura promedio.

II. DISEÑOS DE SOLUCIÓN

Para realizar la comparación de los sistemas, se implementará este programa en tres diferentes arquitecturas:

Monolítica: La cual consiste en que solamente un proceso se encargue de realizar todo el programa solicitado.

Basada en servicios: Aquí varios procesos deben encargarse de realizar el programa, es decir que debe haber una comunicación entre varios procesos para que entre todos ellos realicen los requerimientos solicitados.

Basada en eventos: El programa debe ser ejecutado por varios procesos y todos estos enrutados por un middleware. Utilizando la librería de Python mpi4py o MPI.

MPI (Message Passing Interface): Es un sistema estandarizado de paso de mensajes diseñado para funcionar en una variedad de computadores paralelos, la cual permite en este caso trabajar con Python explotando el trabajo en múltiples procesadores a través de la importación de la librería mpi4py y sus funciones.

III. METODOLOGÍA

Para la implementación de un sistema monolítico se creó un programa el cual contiene una clase Estación la cual cuenta con los atributos nombre y temperatura, y es aquí en esta clase donde se manejan los datos.

Luego una función se encarga de encontrar las estaciones por nombre en una lista y otra función se encarga de leer el archivo en modo lectura y de guardar las estaciones en una lista.

Por último el bloque principal de ejecución del programa o función Main, se encarga llamar a las funciones mencionadas anteriormente luego abre el archivo .txt y calcula las temperaturas mínimas, máximas y el promedio de dichas temperaturas para cada estación para posteriormente escribir los valores obtenidos en un archivo de salida. Además aquí es donde se mide el tiempo de ejecución del programa.

En el caso del sistema orientado a servicios se trabajó básicamente con dos clases las cuales son dos procesos distintos los cuales interactúan de forma en que la primera clase o servicio ReadFileService posee una función que se encarga de leer archivos y luego en el bloque principal se comunica y le pasa la información obtenida del archivo de entrada a otra clase llamada CalcularTemperaturaService, la cual se encarga de realizar los cálculos y los escribe en el archivo de salida.

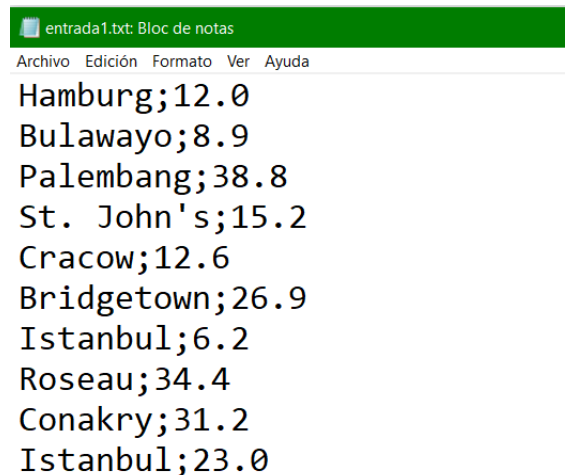
Por lo que en esta última arquitectura se trabajó con dos servicios en donde cada uno vive en su propio proceso y se comunican entre sí. En este caso son dos servicios los cuales se encargan del funcionamiento del programa.

Posteriormente en la implementación orientada a eventos donde se trabajó utilizando principalmente las funciones de la librería mpi4py.

Aquí hay una función que lee el archivo y devuelve una lista con la estación y su temperatura, otra función que se encarga de realizar los cálculos y la función main es la que se encarga de inicializar MPI y dividir en porciones (o chunks) los datos entre los procesos.

Específicamente los eventos que ocurren en este sistema son en los momentos donde se leen las temperaturas.

Un ejemplo de un archivo de entrada es de la siguiente forma:



```
entrada1.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
Hamburg;12.0
Bulawayo;8.9
Palembang;38.8
St. John's;15.2
Cracow;12.6
Bridgetown;26.9
Istanbul;6.2
Roseau;34.4
Conakry;31.2
Istanbul;23.0
```

Figura 1: Ejemplo de archivo de entrada.

Basándose en esta estructura del input, se construyó un generador de entradas aleatorio, el cual produce un archivo de entrada con un tamaño n de mediciones de temperaturas para su respectiva estación. Lo anterior se decidió para asegurar que se tiene una muestra que genere tiempos de ejecución para cada arquitectura más representativos.

IV. RESULTADOS Y DISCUSIÓN

En primer lugar, se generaron 3 archivos de entrada con tamaños 10.000, 100.000 y 1.000.000 para las pruebas de manera de observar cómo se comporta el tiempo de ejecución, para cada uno de estos archivos se realizó la ejecución un total de 5 veces para obtener un tiempo de ejecución medio por arquitectura, por archivo. En el caso de la arquitectura por eventos, las pruebas preliminares se realizaron utilizando un n de 5 hebras para la ejecución. Posteriormente se realizan pruebas en específico para esta arquitectura para analizar el comportamiento de este parámetro en cuanto a los tiempos de ejecución.

Es importante mencionar que las pruebas de ejecución de las arquitecturas fueron realizadas sobre una máquina con sistema operativo Windows 10 con procesador Intel Core I5-7500 (4 núcleos) y 16GB de memoria RAM.

De esta manera entonces, para el archivo 1 ($n = 10k$) se obtuvieron las siguientes mediciones:

Arquitecturas	Monolítica	Bas. Servicios	Bas. Eventos
Mediciones			
1	0,010014	0,007989	0,024003
2	0,009032	0,007996	0,028003
3	0,01	0,008	0,029003
4	0,009031	0,007971	0,03
5	0,009031	0,007971	0,029
T. medio [s]	0,0094216	0,0079854	0,0280018

Es interesante notar que para este tamaño de entrada, la diferencia entre arquitecturas es prácticamente despreciable, siendo la basada en servicios la más lenta.

Para el archivo 2 ($n = 100k$) se obtuvieron los siguientes resultados:

Arquitecturas	Monolítica	Bas. Servicios	Bas. Eventos
Mediciones			
1	0,961999	0,826031	1,278001
2	0,937123	0,859999	1,287999
3	0,956001	0,814031	1,289037
4	0,935025	0,836053	1,363001
5	0,951029	0,86397	1,415999
T. medio [s]	0,9482354	0,8400168	1,3268074

Nuevamente puede observarse una similaridad entre los tiempos de ejecución, sin embargo podemos notar que la arquitectura basada en servicios se aleja de las otras, siendo nuevamente la más lenta.

Por último, para el archivo 3 ($n = 1M$) se obtuvieron los siguientes resultados:

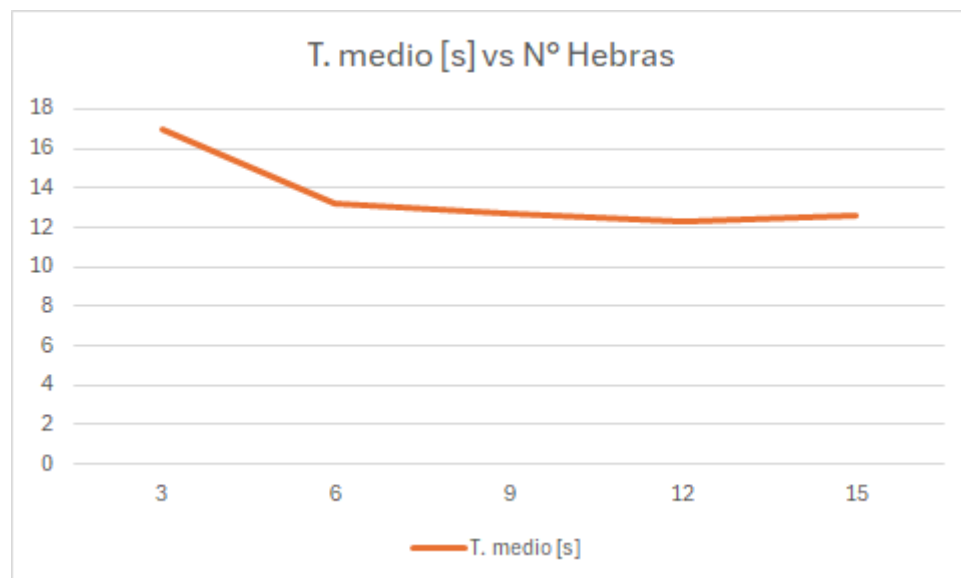
Arquitecturas	Monolítica	Bas. Servicios	Bas. Eventos
Mediciones			
1	9,418038	9,733852	17,193794
2	9,400011	8,238072	13,478256
3	9,30316	8,254305	13,575001
4	9,368017	8,225966	13,427
5	9,366998	8,201001	13,432818
T. medio [s]	9,3712448	8,5306392	14,2213738

Debido al volumen de los datos que procesa cada arquitectura se nota claramente un aumento en el tiempo de ejecución para las 3 arquitecturas, siendo la arquitectura basada en eventos la más notoria. Esto podría explicarse dado que a pesar de que los datos de entrada se

reparten entre las hebras debido a cómo fue implementada la arquitectura, las hebras deben aún comunicarse con la principal primero, al recibir la información que a cada una le corresponde procesar y luego para enviar la solución parcial. En consecuencia, el procesamiento de los datos en sí definitivamente debe tomar menos tiempo de cómputo, pero es esta comunicación la que posiblemente entonces está generando esta discrepancia en los tiempos.

Debido a esto, se decide analizar el comportamiento del número de hebras utilizadas por mpi para realizar el cálculo contra el tiempo de ejecución del programa con el archivo 3. De esta manera entonces, se obtienen los siguientes resultados:

Hebras	3	6	9	12	15
Mediciones					
1	17,243062	13,225014	12,386074	12,294999	12,631001
2	16,672	13,240023	12,956	12,384999	12,592
T. medio [s]	16,957531	13,2325185	12,671037	12,339999	12,6115005



En el gráfico y tabla anterior puede verse como el tiempo medio de ejecución tiende a bajar al aumentar el número de hebras utilizadas. Notar que llega a su mínimo con un $n = 12$ hebras para luego de nuevo ligeramente aumentar. Esto refuerza de alguna manera la teoría anterior de que es la comunicación entre las hebras la que produce la demora “extra” en el

tiempo de ejecución. Esto resulta lógico pues mientras más “hijos” deba atender la hebra original para construir la solución definitiva, más tiempo demora la ejecución total del programa, es decir el aumento de las hebras entonces trae consigo rendimientos decrecientes para con el tiempo de cómputo. Anecdóticamente, se realiza también la prueba con un $tn = 1000$ hebras, para lo cual se obtuvo un tiempo de ejecución de ~ 31 [s], lo que confirma lo anterior. (El sistema durante esta ejecución se quedó sin memoria RAM disponible, por lo que este valor podría estar distorsionado por el tiempo que demora el SO en liberarla para continuar la ejecución)

Por otra parte, a falta de mayor experimentación, bien podría ser posible construir una arquitectura basada en eventos optimizada para la naturaleza de este problema que pudiera resolver el cuello de botella que en esta supone la comunicación entre la hebra “padre” y las hijas, mejorando así los tiempos de ejecución.

V. CONCLUSIÓN

Como puede apreciarse a través de los resultados obtenidos en este laboratorio, puede notarse que en la elaboración e implementación de sistemas distribuidos es crucial no sólo la arquitectura utilizada, sino que también la lógica que existe detrás de ella. Específicamente fijándonos en el caso de las arquitecturas distribuidas implementadas (basada en servicios y basada en eventos) es necesario mencionar que este entorno académico no existen factores como por ejemplo la latencia geográfica, esto es, los servicios y (en el caso de la arquitectura basada en eventos) las hebras que ejecutan la lógica de programa se encuentran físicamente en la misma máquina, cosa que en la aplicación real de estos sistemas no tendría por qué ser así, por lo que entra en juego la habilidad que se tenga a la hora de diseñar este tipo de sistemas con el objeto de minimizar (o idealmente eliminar) los elementos perjudiciales a estos.

Además de esto se logra extrapolar que en un ambiente de aplicación real, se le debe dar mucha importancia a los sistemas distribuidos, ya que deben estar preparados para adaptarse a fluctuaciones impredecibles como la demanda, los problemas de latencia y de ancho de banda, y a su vez deben tener la capacidad de recuperarse de fallos dentro del mismo sistema, por lo que factores como la escalabilidad (ya sea en tamaño, geográfica o administrativa) y la tolerancia de fallos, y otras características como mostrar un cierto grado de transparencia son elementos cruciales y esenciales para diseñar un sistema distribuido apropiado, que pueda residir hoy en día en el mundo real fuera de un ambiente controlado como lo fue en este caso.