Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2523 — Sistemas Distribuidos

Tarea 2

Simulación de algoritmos de consenso

Fecha de inicio: **Martes 23 de septiembre** a las **20:00:00** hrs

Plazo máximo para **formalizar parejas**: **Miércoles 1 de octubre** a las **20:00:00** hrs.

Formulario de inscripción: `https://forms.office.com/r/77eGVFMZNL` Es obligatorio responder este formulario si trabajarás con otro estudiante. En otro caso, se asumirá que trabajarás de forma individual y no se aceptará reclamos al respecto.

Fecha de entrega: **Lunes 6 de octubre** a las **20:00:00** hrs.

Fecha máxima de entrega (con atraso): **Miércoles 8 de octubre** a las **20:00:00** hrs.

Evaluación en el contexto del curso

Esta es una evaluación **de corrección automática** y con la opción de realizarla de forma **individual** o **en parejas**. Además, pretende rescatar evidencias del desarrollo de ciertos resultados de aprendizaje. Por eso, tras su realización y entrega, recibirás retroalimentación respecto a los diferentes *tests* que se esperaba que la tarea lograra exitosamente. En particular, los objetivos específicos de esta evaluación son:

- Implementar correctamente las reglas del algoritmo Paxos y del algoritmo Raft para consolidar una base de datos con nodos distribuidos.
- Manejar adecuadamente casos límite, como nodos inactivos, acciones inválidas o pérdida de liderazgo.
- Leer e interpretar archivos de entrada que simulan operaciones distribuidas.

Contexto

No sabes cómo, pero de repente te diste cuenta que tu profesor te dejó envuelto en una investigación compleja que involucra múltiples escenas del crimen y declaraciones contradictorias. Junto a Conan Edogawa te enfrentas a un gran desafío: la información de cada caso está fragmentada, distribuida entre diferentes equipos de detectives y, en algunos casos, desactualizada o incompleta.

Cada equipo ha reportado acciones relevantes del caso sobre distintas piezas de evidencia —como sospechosos, ubicaciones, horas clave o resultados forenses—, pero no existe un único registro maestro. Algunas acciones fueron aplicadas prematuramente, otras se contradicen, y otras tantas nunca fueron compartidas correctamente. Para evitar conclusiones erróneas, es necesario aplicar un algoritmo de consenso que permita validar qué acciones son aceptadas por la mayoría de los equipos de investigación, y por lo tanto serán consideradas verdaderas y aplicables al caso final.

Tu rol en esta misión será ayudar a Conan con un simulador capaz de aplicar algunos algoritmos de consenso sobre los registros recolectados, consolidar las acciones válidas, y reconstruir la información real del caso. Solo así será posible deducir con certeza quién es el culpable. Y nunca lo olvides, la verdad siempre es una sola.

Introducción

Esta evaluación se divide en 2 partes que deben ser desarrolladas en conjunto. Por un lado, deberás implementar las reglas de dos algoritmos de consenso: Paxos y Raft, para determinar cuáles acciones son aceptadas como válidas para la base de datos. Por otro lado, considerando únicamente las acciones aceptadas en la base de datos, deberás simular dichas operaciones para determinar el estado de la base de datos en diferentes instantes de tiempo.

Toda la tarea debe ser desarrollada **con Python** $3.X$ **con** $X \leq 12$.

1. Simulación de acciones

El objetivo de esta tarea es lograr determinar el estado de las diferentes variables de una base de datos tras simular un variado conjunto de operaciones a dicha base. Para esto, se te entregará diferentes acciones cuyo formato es el siguiente: `COMANDO-VARIABLE-VALOR`.

- `COMANDO` corresponde a un *string* que puede tomar únicamente 3 posibles valores:
  - `SET`: Asigna un valor a una variable, sin importar si la variable ya existe.
  - `ADD`: Modifica una variable para sumar (en caso de un `int`) o concatenar (en caso de un `str`). Si la variable a modificar no existe, este comando es análogo a aplicar el `SET`.

    Además, esta operación siempre intentará trabajar los datos como `int` para poder sumarlos, a no ser que alguno de los elementos a operar sea incapaz de ser transformado a `int`, es decir, que aplicar `.isdigit()` a alguno de los elementos a operar de como resultado `False`, en ese caso serán tratados como *string* y se concatenará.
  - `DEL`: Elimina una variable. Si no existir la variable, esta acción no produce ningún efecto.
- `VARIABLE` corresponde al nombre de la variable que se desea trabajar en la operación. Esta siempre será una palabra compuesta por caracteres del abecedario español (a-z y A-Z), espacios y/o guiones bajo.
- `VALOR` representa un valor que puede ser un número entero neutro o positivo (`int` >= 0) o una cadena de caracteres (`str`) que será utilizada en el comando. En el caso de ser una cadena, esta puede incluir cualquier carácter compatible con el estándar *Unicode*, tales como letras, números, símbolos, guiones, espacios, emojis, entre otros. Las únicas restricciones garantizadas son que la cadena no finalizará con espacios y no contendrá el carácter `#`, ya que este último se reserva para otros fines durante la evaluación.

  Finalmente, este dato está presente únicamente si el comando es `SET` o `ADD`.

Algunos ejemplos de posibles acciones son:

```
SET-sospechoso-KAITO KID 1412
SET-sospechoso-Kaito-Kid
ADD-sospechoso- ; Aoko-Nakamori
SET-nombre-Ai
SET-deuda-2
ADD-lugar-Beika
ADD-deuda-4442
ADD-deuda-1.0
ADD-conan-17
ADD-conan-c
DEL-nombre
```

Luego, simulando todas las acciones indicadas previamente en el mismo orden que están definidas. La base de datos quedaría en el siguiente estado:

Variable	Valor
sospechoso	Kaito-Kid ; Aoko-Nakamori
deuda	44441.0
lugar	Beika
conan	17c

2. Consenso de acciones

La problemática abordada en esta evaluación consiste en que numerosas acciones son consideradas inválidas por la base de datos. Esta situación se debe a que los distintos nodos encargados de aceptar dichas acciones inicialmente no logran alcanzar un acuerdo respecto a su incorporación en la base de datos. Por ello, se hace necesario aplicar un algoritmo de consenso, específicamente Paxos y Raft.

Para ambos algoritmos se proporcionará un conjunto de archivos con extensión `.txt`, donde cada archivo representa un caso o *test* distinto, y contiene el registro de todos los eventos ocurridos en un sistema distribuido. El objetivo es aplicar los algoritmos Paxos y Raft, tal como fueron estudiados en clase, para simular los distintos eventos y determinar cuáles acciones deben considerarse válidas para la base de datos; es decir, aquellas para las que la mayoría de los nodos haya logrado alcanzar un consenso.

2.1. Supuestos y reglas generales

Antes de detallar cada algoritmo, hay ciertos supuestos y reglas que debes tener en cuenta en la implementación que desarrolles.

- Cada nodo involucrado en estas simulaciones presentará un ID. Este ID estará formado por caracteres alfanuméricos en mayúscula o minúscula, y pueden contener tanto guiones como espacios. Lo único que se garantiza es que los ID no contendrán los símbolos de comentario `(#)`, el punto y coma `(;)` y la coma `(,)`.
- Al comenzar la ejecución de un archivo, todos los nodos se encontrarán activos.
- Similar a la tarea 1, simularemos que un nodo se cae y vuelve a levantar con los comandos `Start` y `Stop`. Durante la caída de un nodo, este únicamente deja de procesar cualquier mensaje de los demás nodos, pero su base de datos o cualquier atributo que tenga queda intacto. Luego, al momento de levantarse el nodo, este únicamente comienza a procesar los mensajes que le lleguen desde ese momento, pero no intentará hacer ninguna otra acción. Finalmente, si se hace `Start` a un nodo activo o `Stop` a un nodo inactivo, este evento no debe generar ningún error.
- El *script* desarrollado debe ser capaz de procesar archivos que incluyan comentarios. Estos comentarios deberán ser ignorados al momento de procesar la información. Un comentario inicia siempre con el carácter #, el cual puede estar al inicio de una línea o en una línea de evento o declaración de nodos. En cualquier caso, el comentario debe ser ignorado sin afectar la información del archivo.
- Cada archivo `.txt` debe procesarse de forma independiente. El estado de la base de datos no debe compartirse entre simulaciones.
- **NO puedes asumir que todo ID de nodo utilizado en los eventos corresponde a uno de los definidos en las primeras líneas de cada *test***. Por ejemplo, un nodo se puede llamar `Conan` y en un evento se usa el nodo `conan`, es decir, son *strings* distintos. En caso de que uno o más ID no existan, esos ID son ignorados del evento. Finalmente, si no es posible ejecutar el comando dada la ausencia de algún ID válido, entonces el evento es ignorado en su totalidad y no tiene ningún efecto en la simulación.
- Puedes asumir que los archivos `.txt` siempre cumplen con el formato descrito y que **fueron creados con un *encoding* de UTF-8**.
- La ejecución de cada evento debe **seguir estrictamente las reglas y restricciones específicas de cada algoritmo** visto en el curso.

2.2. Paxos

En esta evaluación implementaremos una pequeña variante del algoritmo Paxos. Esta versión seguirá todas las reglas estudiadas en el curso, con la incorporación de una condición adicional cuyo propósito es permitir la consolidación de múltiples acciones en la base de datos. Específicamente, cada vez que los nodos aceptantes participen en el proceso de consolidación de una acción (fase *learn*), deberán limpiar su registro interno; es decir, deberán considerar que no han realizado ninguna promesa ni aceptado previamente ninguna acción. Además, todos los nodos proponente deberán olvidan cualquier `Prepare` realizado.

Luego, para la ejecución de este algoritmo, cada archivo respetará el siguiente formato:

- La primera línea corresponde al ID de cada nodo aceptante. Los ID están separados por punto y coma (`;`).

  Un ejemplo de esta línea es: `NODO-A;NODO-B`.

- La segunda línea corresponde al ID de cada nodo proponente. Los ID están separados por punto y coma (`;`).

  Un ejemplo de esta línea es: `NODO-KAI;NODO-TO`.

- Todas las siguientes líneas corresponden a espacios en blanco, comentarios o son eventos que le ocurrieron a los nodos aceptadores o nodos proponente.

Respecto a los eventos que pueden ocurrir, estos siempre seguirán el siguiente formato: `COMANDO;ARGUMENTOS`. Donde el `COMANDO` puede tomar uno de los siguientes valores:

- `Prepare`: este evento representa la preparación para emitir una propuesta. Sus argumentos son dos: primero el ID del nodo proponente que desea realizar la preparación, seguido de un identificador único de la propuesta el cual será siempre un número entero positivo (`int` >= 1).

  Un ejemplo de este evento sería `Prepare;NODO-KAI;4`.

- `Accept`: este evento representa el envío de una propuesta. Sus argumentos son tres: en primer lugar, el identificador del nodo proponente; en segundo lugar, el identificador de la propuesta; y finalmente, la acción que se desea consolidar en la base de datos. Un nodo proponente emitirá este evento únicamente si previamente ha enviado un evento `Prepare` con el mismo identificador de propuesta y dicho evento fue aceptado por la mayoría de los nodos aceptantes.

  Un ejemplo de este evento sería `Accept;NODO-KAI;4;SET-DEUDA-4444`.

- `Stop`: este evento representa la caída de algún **nodo aceptante**. Su argumento es uno: el ID del nodo aceptante. Mientras el nodo esté caído, va a ignorar cualquier mensaje que le llegue excepto el evento `Start`.

  Un ejemplo de este evento sería `Stop;NODO-KAI`.

- `Start`: este evento representa la activación de algún **nodo aceptante**. Su argumento es uno: el ID del nodo aceptante. Al momento de activar, el nodo recuerda el ID del último nodo proponente que le envió un Prepare y cualquier acción aceptada previamente.

  Un ejemplo de este evento sería `Start;NODO-TO`.

- `Learn`: este evento representa el proceso de consolidación de una acción. Al ejecutarse, se verifica si existe una acción que haya sido aceptada por la mayoría de los nodos aceptantes. En caso afirmativo, dicha acción se registra como una acción válida y aceptada por la base de datos. Si no se alcanza el consenso, la acción se considera inválida.

  Además, como se indicó previamente, cada vez que se aplica el evento `Learn`, todos los nodos que participaron en el proceso de consolidación, es decir, los nodos aceptantes que están activos, deben reiniciar su estado, con el fin de permitir la realización de un nuevo proceso de consenso. Además, de que los nodos proponente olvidan cualquier `Prepare` realizado.

  Un ejemplo de este evento sería `Learn`.

- `Log`: este evento representa una consulta a la base de datos. Su argumento es uno: el nombre de una variable a consultar. En la sección *Log* de la base de datos se detallará el formato y cómo procesar este evento.

Recuerda que **deben aplicarse correctamente las reglas del algoritmo Paxos vistas en clase** para determinar cuándo un evento `Accept` o `Prepare` es válido para los nodos aceptantes, junto con determinar correctamente la acción que será finalmente almacenada en la base de datos, es decir, es consolidada.

2.3. Raft

Para la ejecución de este algoritmo, cada archivo respetará el siguiente formato:

- La primera línea corresponde a cada nodo del servidor distribuido. Cada nodo se representa como una tupla donde el primer elemento corresponde a su ID, mientras que el segundo elemento es un número entero positivo que representa su *election timeout*. En el archivo, cada nodo se encuentra separado por punto y coma (`;`), mientras que cada tupla del nodo estará separada por una coma. Además, puedes asumir que el *election timeout* será único.

  Un ejemplo de esta línea es: `NODO-A,3;NODO-I,2;NODO-HAIBARA,10`.

- Todas las siguientes líneas corresponden a espacios en blanco, comentarios o son eventos que le ocurrieron a los nodos.

Respecto a los eventos que pueden ocurrir, estos siempre seguirán el siguiente formato: `COMANDO;ARGUMENTOS`. Donde el `COMANDO` puede tomar uno de los siguientes valores:

- `Send`: este evento representa el envío de una acción al líder. Este evento solo utiliza un argumento: la acción que el líder debe almacenar en su *log*.

  Un ejemplo de este evento sería `Send;SET-sospechoso-Kaito-kid`.

- `Spread`: este evento representa la propagación de todas las acciones que tiene el líder a determinados nodos. Este evento solo utiliza un argumento: una lista de ID de los nodos a quien le enviará las acciones que tiene hasta el momento. Puede pasar que la lista esté vacía, incluya el ID del nodo líder, ID de nodos que tienen `STOP` o ID de nodos inexistentes; en cualquiera de estos casos, el comando no debe provocar ningún error en el algoritmo y solo propagar a los nodos que corresponde recibir el mensaje.

  Un ejemplo de este evento sería `Spread;[NODO-I,NODO-A, NODO-NO-EXISTE]`.

- `Stop`: este evento representa la caída de algún nodo. Su argumento es uno: el ID del nodo caído. Mientras el nodo esté caído, va a ignorar cualquier mensaje que le llegue excepto el evento `Start`. En caso que el nodo caído sea el líder, se debe aplicar inmediatamente las reglas del algoritmo Raft para determinar el siguiente líder.

Un ejemplo de este evento sería `Stop;NODO-HAIBARA`.

- `Start`: este evento representa la activación de algún nodo. Su argumento es uno: el ID del nodo caído. Al momento de activarse, el nodo recuerda todos las acciones que recibió a antes de la caída.

  Un ejemplo de este evento sería `Start;NODO-I`.

- `Log`: este evento representa una consulta a la base de datos. Su argumento es uno: el nombre de una variable a consultar. En la sección *Log* de la base de datos se detallará el formato y cómo procesar este evento.

Para este algoritmo, se asume que ya hay un líder elegido, según las reglas del Raft, justo antes de ejecutar el primer evento de cada archivo. Además, puedes asumir que el nodo elegido como líder siempre enviará un *heartbeat* con un periodo menor a todos los *election timeout*.

Es importante mencionar que, en caso de no existir un nodo líder al momento de ejecutar un evento `Send` o `Spread`, dicho evento no tendrá efecto y se deberá procesar al siguiente evento.

Finalmente, recuerda que **deben aplicarse correctamente todas las reglas del algoritmo Raft vistas en clase** para determinar cuándo una acción puede ser aceptada e incluida en la base de datos, es decir, es consolidada. No olvides determinar correctamente cuando una acción es consolidada directamente o indirectamente.

2.4. *Log* de la base de datos

Para cada *test* ejecutado, se deberá crear un archivo `.txt` con el registro de cada evento `Log` y con cada variable existente dentro de la base de datos **final**. Si este archivo ya existe, debes sobre-escribirlo con el resultado de la nueva ejecución.

Parte 1 - Registro de evento Log

Cada vez que aparezca la instrucción `Log;Variable`, se debe registrar **el valor consolidado de la variable indicada**. Dicho valor debe reflejar el estado de la variable tras ejecutar todas las acciones que fueron consolidadas correctamente, según las reglas del algoritmo simulado, hasta justo el momento que apareció la instrucción `Log`.

Las reglas de este registro son las siguientes:

- La primera línea del archivo siempre será la palabra `LOGS`.
- Las siguientes n líneas corresponde al resultado de los eventos `"log"`.
  - Si la variable ha sido consolidada, se escribir el nombre de la variable, seguido de un igual (=) y finalmente el valor que tenga dicha variable en el momento de aplicar el evento `"Log"`.
  - Si la variable no existe en la base de datos, se debe escribir el nombre de la variable, seguido de un igual (=) y finalmente el siguiente texto: `Variable no existe`.
  - Cada línea del archivo de salida debe reflejar, en el mismo orden, los valores registrados por el evento `"Log"` del archivo de entrada correspondiente.
  - En caso de $n == 0$, es decir, nunca se haya solicitado el evento `"Log"`, entonces el archivo debe incluir la línea `"No hubo logs"`

Parte 2 - Registro de Base de datos

Una vez registrado todos los eventos `Log`, se debe registrar cada variable existente en la base de datos final, es decir, la base de datos que queda después de ejecutar cada evento del *test*.

Las reglas de este registro son las siguientes:

- La siguiente línea debe ser la frase `"BASE DE DATOS"`
- Las siguientes k líneas corresponden a cada dato de la base de datos, donde se debe escribir el nombre de la variable, seguido de un igual (=) y finalmente el valor que tenga dicha variable finalizada toda la ejecución del *test*. El orden de las variables queda a criterio del programador (orden de creación, orden alfabético, etc).
- En caso de $k == 0$, es decir, la base de datos está vacía, entonces el archivo debe incluir la línea `"No hay datos"`
- **No se deben incluir** mensajes adicionales, líneas en blanco o comentarios en el archivo de salida aparte de lo solicitado en el enunciado.

Para ejemplificar, asumiendo que la base de datos está en el siguiente estado:

```
| Variable    | Valor      |
|------------ |----------- |
| sospechoso  | kaito kid  |
| deuda       | 4444       |
| lugar       | Beika      |
```

Luego, se ejecutan los siguientes eventos:

```
Log;sospechoso
Log;hola
Log;lugar
Log;deuda
```

El archivo resultante deberá ser:

```
LOGS
sospechoso=kaito kid
hola=Variable no existe
lugar=Beika
deuda=4444
BASE DE DATOS
sospechoso=kaito kid
deuda=4444
lugar=Beika
```

En cambio, asumiendo que la base de datos esté vacía y no se hizo ningún evento de `Log`, el archivo resultante deberá ser:

```
LOGS
No hubo logs
BASE DE DATOS
No hay datos
```

Respecto a la ruta donde se guarda el archivo y su nombre. Este se explicará con detalle en la sección Ejecución de la tarea.

3. Ejemplos

En esta sección se presenta un ejemplo, por algoritmo, de posibles secuencias de eventos.

3.1. Ejemplo Paxos - *Prepare* interrumpido

Dada un sistema distribuido de tres nodos aceptantes y dos nodos proponentes. Vamos a simular una situación donde el orden de algunos comandos entre dos nodos proponentes pueden provocar que algunas acciones no aceptadas.

```
A;B;C               # Nodos aceptantes
Takagi;Kogoro       # Nodos proponentes

Accept;Kogoro;1;SET-sospechoso-yukiko            # Acción 1
# Acción 1 (de Kogoro) es inválida porque no hizo prepare con dicho identificador.
# El nodo Kogoro no enviará dicho accept.

Prepare;Kogoro;10
# Kogoro prepara acción 2

Prepare;Takagi;20
# Takagi prepara acción 3

Accept;Kogoro;10;SET-sospechoso-gin              # Acción 2
# Acción 2 será rechazada porque los nodos aceptan solo acciones con identificador >= 20.

Accept;Takagi;20;SET-lugar-Distrito Beika         # Acción 3
# Takagi propone acción 3 y logra que se la acepten

Prepare;Kogoro;100
# Kogoro prepara acción 4, pero se entera que la acción 3 ya fue aceptada.

Accept;Kogoro;100;ADD-lugar- | Casa de Agasa      # Acción 4
# En vez de enviar la acción 4, el algoritmo obliga que se envíe la acción 3
# que ya fue aceptada con n=20.


Log;lugar      # Se verifica la variable antes de consolidar.
Learn          # Se consolida acción 3 y se reinician los nodos activos.
Log;lugar      # Se verifica la variable después de consolidar.
Log;sospechoso # Se verifica otra variable.
```

El archivo resultante debería ser

```
LOGS
lugar=Variable no existe
lugar=Distrito Beika
sospechoso=Variable no existe
BASE DE DATOS
lugar=Distrito Beika
```

3.2. Ejemplo Raft - Acciones no consolidadas

Dada un sistema distribuido de tres nodos, vamos a simular una situación donde una acción logra la mayoría, pero no se consolida por no cumplir las reglas del algoritmo Raft.

```
A,1;B,5;C,3 # Empieza elección de líder: A.

Send;SET-arma homicida-cuchillo           # Acción 1
Spread;[A,B,C]
# Acción 1 logra consolidación directa.

Send;SET-criminal-vecino de la víctima    # Acción 2
# Enviar Acción 2 a A, pero no propaga. No hay consolidación.

Stop;A
# Se cae el líder, se elige el siguiente: C

Send;SET-criminal-pareja de la víctima    # Acción 3
# Enviar Acción 3, pero no propaga. No hay consolidación.

Start;A
Stop;C
# Vuelve A y se cae el líder. Se elige el siguiente: A

Spread;[A,B,C]
# A propaga sus acciones al nodo B y C aunque C está caído.
# Aunque hay mayoría en la acción 2, esta no cumple con los criterios
# de consolidación directa o indirecta.

Log;criminal
Log;arma homicida

Send;DEL-arma_homicida                    # Acción 4
Send;DEL-criminal                         # Acción 5
Spread;[A,B,C]
# Acción 4 y 5 logra consolidación directa. La acción 2 logra consolidación indirecta.

Log;criminal
Log;arma homicida
```

El archivo resultante debería ser

```
LOGS
criminal=Variable no existe
arma homicida=cuchillo
criminal=Variable no existe
arma homicida=cuchillo
BASE DE DATOS
arma homicida=cuchillo
```

4. Archivos

Para el desarrollo de esta evaluación, se provee de los siguientes archivos y carpetas:

- **Modificar** `main.py`

  Este es el archivo que se ejecutará para simular cualquiera de los 2 algoritmos en un *test* específico.

  El formato para ejecutar este archivo es `python3 main.py [ALGORITMO] [RUTA]`, donde `ALGORITMO` puede ser `Raft` o `Paxos` y `RUTA` es el *path* relativo (desde la ubicación de `main.py`) del *test* a simular.

  Un ejemplo de ejecución es `python3 main.py Paxos casos_Paxos/test_01.txt`.

- **No modificar** `ejecutar_tests.py`

  Este es un archivo encargado de ejecutar `main.py` para todos los *tests* públicos existentes y luego contrastar la respuesta generada por `main.py` con la solución esperada.

  Un ejemplo de ejecución es `python3 ejecutar_tests.py`.

  Este archivo va a ser diferente al momento de corregir las tareas. Únicamente se mantendrá que este archivo utilizará `os.path.join` para garantizar que cada *path* confeccionado por *test* sea compatible al sistema operativo y que ejecutará uno por uno cada *test*.

  Solo se permite modificar la línea 4 (constante `COMANDO_PYTHON`) para que se utilice el comando de Python correspondiente al PC del estudiante.

- **No modificar** `casos_Paxos/`

  Esta carpeta contiene todos los archivos `.txt` con los diferentes *test* públicos para el algoritmo Paxos. Esta carpeta solo existe para probar la tarea durante el desarrollo de ella, no puedes asumir que existirá al momento de corregir la tarea.

- **No modificar** `casos_Raft/`

  Esta carpeta contiene todos los archivos `.txt` con los diferentes *test* públicos para el algoritmo Raft. Esta carpeta solo existe para probar la tarea durante el desarrollo de ella, no puedes asumir que existirá al momento de corregir la tarea.

- **No modificar** `logs/`

  Esta carpeta, inicialmente vacía, será utilizara para guardar todos los `.txt` con los *logs* de cada casos ejecutado. Al momento de ejecutar la tarea, el cuerpo docente se asegurará de crear esta carpeta vacía.

- **No modificar** `logs_esperados/`

  Esta carpeta contiene los *logs* esperados para todos los *tests* incluidos junto a este enunciado. Lo esperado es que, luego de finalizar completamente la tarea, la carpeta `logs/` se vea exactamente igual que `logs_esperados/`.

5. Ejecución de la tarea

Cada vez que se ejecute `main.py`, este *script* recibe el nombre del algoritmo a ejecutar y el *path* relativo del *test* a ejecutar. Utilizando el *path*, el *script* deberá simular el algoritmo **en un máximo de 1 segundo**. Además, deberá almacenar el nombre del archivo `.txt` que será utilizado para generar un archivo de salida en la carpeta `logs/` cuyo nombre siempre debe seguir el siguiente formato:

```
[ALGORITMO]_[NOMBRE_ARCHIVO_ENTRADA].txt
```

Donde `[ALGORITMO]` será `Paxos` o `Raft` y `[NOMBRE_ARCHIVO_ENTRADA]` será únicamente el nombre del *test* a ejecutar sin considerar nada respecto al directorio en donde se encuentre el archivo.

Por ejemplo, asumiendo que una entrega tiene la siguiente estructura:

```
main.py
logs/
    [vacio por ahora]
CARPETA_NUEVA/
    caso_01.txt
    conan.txt
    ...
casos_Raft/
    caso_412.txt
    hernan4444.txt
    ...
```

Se ejecuta 2 veces el archivo `main.py` de la siguiente forma:

1. `python3 main.py Paxos CARPETA_NUEVA/conan.txt`
2. `python3 main.py Raft casos_Raft/hernan4444.txt`

El directorio de la tarea deberá quedar del siguiente modo:

```
main.py
logs/
    Paxos_conan.txt
    Raft_hernan4444.txt
CARPETA_NUEVA/
    caso_01.txt
    conan.txt
casos_Raft/
    caso_01.txt
    hernan4444.txt
```

Puedes asumir que **siempre** se escogerá el algoritmo correspondiente al archivo que se va a ejecutar, es decir, no se intentará simular Paxos en un archivo cuyo contenido es para Raft, o viceversa. Además, puedes asumir que el *path* entregado apuntará a un archivo que realmente existe haciendo directamente `open(path, encoding='UTF-8')`, es decir, no debes modificar nada de dicho *path* para poder acceder al *test* correspondiente.

Finalmente, puedes crear más archivos `.py` si lo encuentras pertinente. No obstante, debes asegurar que la forma de ejecutar la tarea se mantenga tal como indica el enunciado.

6. Corrección automatizada

La corrección de toda la evaluación será **únicamente mediante *test cases*** para evaluar diferentes casos borde que puedan existir. En consecuencia, entregar una evaluación que no pueda ejecutarse implicará que esta será calificada con la nota mínima y sin posibilidad de corregir el error. Es responsabilidad del estudiante entregar un código ejecutable.

Adicionalmente, **no se permite utilizar ninguna librería adicional a las ya incluidas al inicio de los archivos entregados**. Si se detecta su uso o alguna tarea no se puede ejecutar porque se hizo `import` de alguna librería no autorizada, dicha tarea será evaluada con nota mínima.

Respecto a la corrección de ambos algoritmos, se realizarán diferentes eventos para consolidar operaciones en la base de datos, junto con diversos eventos del tipo `Log` para verificar que la base de datos esté correctamente consolidada en cualquier momento.

Dado lo anterior, es **imperativo** asegurar que los eventos del tipo `Log` estén correctamente implementados para que los *test cases* pueden verificar el desarrollo de la tarea exitosamente.

Es importante recordar que los ***tests* públicos son solo un subconjunto de todos los casos que se evaluarán**. Queda a responsabilidad del estudiante probar la tareas con otros casos para validar la correcta implementación de ambos algoritmos.

Desglose de puntaje

- **(0.5 puntos)** Con el algoritmo Paxos, se aprueban todos los casos donde se tiene únicamente 1 nodo proponente y los nodos aceptantes nunca se caen.
- **(1.0 punto)** Con el algoritmo Paxos, se aprueban todos los casos donde se tiene únicamente 1 nodo proponente y al menos 1 nodo aceptante se cae temporalmente.
- **(0.5 punto)** Con el algoritmo Paxos, se aprueban todos los casos donde se tiene 2 o más nodos proponentes y los nodos aceptantes nunca se caen.
- **(1.0 puntos)** Con el algoritmo Paxos, se aprueban todos los casos donde se tiene 2 o más nodos proponentes y al menos 1 nodo aceptante se cae temporalmente o permanentemente.
- **(0.5 puntos)** Con el algoritmo Raft, se aprueban todos los casos donde solo hubo un único nodo es líder en todo momento y los nodos nunca se caen.
- **(0.5 puntos)** Con el algoritmo Raft, se aprueban todos los casos donde solo hubo un único nodo como líder en todo momento y al menos uno de los nodos que no son líder se cae temporalmente o permanentemente.
- **(1.0 punto)** Con el algoritmo Raft, se aprueban todos los casos donde los nodos se pueden caer temporalmente o permanentemente, pero el liderazgo solo lo logran 2 nodos distintos durante todo el *test*.
- **(1.0 punto)** Con el algoritmo Raft, se aprueban todos los casos donde los nodos se pueden caer y el liderazgo lo logran 3 o más nodos distintos durante todo el *test*.

Asignación de puntaje

Sea L la cantidad de `Logs` realizados en un *test* y D la cantidad de datos finales en la base de datos del mismo *test*. Se define C como $L + D$ que corresponde a la cantidad de consultas que se evaluarán el archivo de resultado.

Cada *test* podrá recibir uno de los siguientes valores:

- **2 puntos**: Las C consultas tienen el valor esperado.
- **1 punto**: Al menos el 60 % de las C consultas tienen el valor esperado.
- **0 puntos**: No se alcanza el porcentaje de consultas correctas en el *test*.

Para los casos donde $L == 0$ (no hay *logs)*, la única consulta a verificar es que exista el *string* correspondiente: `"No hubo logs"`, es decir, L será 1. Del mismo modo, para los casos que $D == 0$ (base de datos vacía al final de la simulación), la única consulta a verificar es que exista el *string* correspondiente: `"No hay datos"`, es decir, D será 1.

Es fundamental que la sección de `LOG` del archivo de salida contenga exactamente una línea por cada `log`, en el mismo orden en que fueron presentados en el *test*, y sin incluir líneas en blanco. La corrección se realizará comparando línea por línea con un archivo de referencia, por lo que cualquier desviación en el formato, contenido o posición —aunque la respuesta sea conceptualmente correcta— será considerada incorrecta. Asegúrense de respetar estrictamente el formato y el orden indicados para que sus respuestas puedan ser evaluadas correctamente.

Luego, para la sección de `BASE DE DATOS` del archivo de salida, el orden de las variables queda a decisión del programador. Para revisar esa sección, se hará la intersección de `set` entre la base de datos generada por la tarea y la base de datos esperada para determinar la cantidad de datos correctos. En caso que la base de datos del estudiante contenga más datos de la esperada, se descontará 1 consulta correcta por cada dato adicional.

Finalmente, cada ítem estará compuesto por un conjunto arbitrario de *tests*. Sea N la cantidad total de *tests*, se calculará el nivel de logro del ítem, el cual se utilizará para ponderar su puntaje máximo y así obtener el puntaje final correspondiente. La fórmula será la siguiente:

$$puntaje\_item_x = round\left(\frac{total\_puntaje\_tests\_logrado_x}{N \times 2} \times puntaje\_maximo\_item_x,\ 2\right)$$

Descuentos

La evaluación está sujeta a ciertas reglas formales. El incumplimiento de alguna de ellas puede implicar descuentos automáticos o a criterio del equipo docente, según se indica a continuación:

- **PEP8 – Largo de línea**

  Cada línea de los archivos `.py`, sin importar la naturaleza de la línea, debe tener un **máximo de 100 caracteres**, incluyendo espacios y tabulaciones. Si al menos **una línea** supera los 100 caracteres (es decir, tiene 101 o más), se descontarán 0.2 puntos de la nota final, incluso si la línea excedente contiene solo espacios, tabulaciones o comentarios.

- **Modularización**

  Cada archivo `.py` puede contener un **máximo de 400 líneas**, incluyendo código, comentarios y líneas en blanco. Si al menos **un archivo supera este límite** (401 líneas o más), se descontarán 0.2 puntos de la nota final, sin importar el contenido de la línea extra.

- **Formato de Entrega**

  Si el estudiante entrega una solución funcional, pero esta no respeta alguna regla del enunciado o del curso, el descuento será determinado por el equipo docente, y podrá variar entre 0.1 puntos y hasta 6.0 puntos, dependiendo de la gravedad del error.

  Por ejemplo, si un estudiante utiliza una librería externa que no estaba permitida y debido a ello, el código no se ejecuta correctamente en el computador del equipo docente, la penalización será de 6.0 puntos.

  En cambio, si el estudiante incluye un `import` de una librería no permitida que impide la ejecución del código en los computadores del equipo docente, pero el código no hace uso real de dicha librería y solamente eliminando el `import` permite que el código funcione. El equipo docente puede evaluar la posibilidad de eliminar dicho `import` para ejecutar la tarea, pero se aplicaría un descuento según lo definido anteriormente.

7. Entregables

Se espera que el entregable sea un archivo ZIP que contenga como mínimo el archivo **main.py**. En caso de crear archivos Python adicionales para modularizar la tarea, estos también deben estar incluidos en el ZIP.

El archivo `.zip` entregado también debe incluir un archivo `README.md` que contenga:

- Todas las **citas** de material externo utilizado.
- Una explicación clara y completa del uso de cualquier **herramienta generadora de código** (por ejemplo: asistentes de inteligencia artificial, generadores automáticos, etc.).

Si el archivo `README.md` está ausente o no incluye alguna de las referencias necesarias, se asumirá que **todo el contenido entregado sin referencias es de autoría exclusiva del estudiante**.

Advertencia: Si se detecta uso de material externo o herramientas generativas sin que estén debidamente citadas en el `README.md`, se considerará una **falta a la integridad académica**, y se aplicarán las sanciones correspondientes según las políticas establecidas en el programa del curso.

En caso de trabajar en parejas, solo un integrante debe entregar el trabajo. En caso que ambos entreguen, se revisará cualquiera de los 2 entregas sin poder apelar a dicha decisión.

Finalmente, no se aceptarán entregas en formatos distintos al indicado (por ejemplo, archivos `.ipynb`, `.pyc`, etc.). Si no se entrega el archivo ZIP o se entrega en un formato diferente, la entrega no será revisada y se asignará la nota mínima.

8. Dudas

Cualquier duda que tengas sobre esta evaluación, prefiere publicarla en el **Syllabus** correspondiente a esta evaluación. También, siente la libertad de responder dudas de tus pares si crees que conoces la respuesta. En caso de tener dudas que impliquen mostrar tu solución o partes de ella, no utilices este medio de consulta. Para estos casos, envía un correo al cuerpo docente o muestra tu solución solo en reunión personal (remota o presencial) cuando te reúnas con algún miembro del cuerpo docente.

9. Política de atraso

Existe la posibilidad de entregar esta evaluación con hasta **2 días de atraso** a partir de la fecha de entrega definida en el enunciado. En la eventualidad de entregar pasada la fecha de entrega, se aplicará una **reducción** a la nota máxima que podrás obtener en esta evaluación.

Si el trabajo se realiza en parejas, se corregirá aquella entregada con atraso y la reducción de nota aplicará a ambos miembros.

De haber atraso, **la nota máxima a obtener** se reduce en **1 punto (10 décimas)** por cada día de atraso. Cada día de atraso se determina como el techo de días de atraso. Por lo tanto, en caso de entregas atrasadas, la nota final de la tarea se calcula mediante la siguiente fórmula:

$$\text{mín}(7 - (0.5 \times \texttt{días\_atraso}),\ \texttt{nota\_obtenida}) - \texttt{descuento\_total})$$

Por otro lado, entregas con más de 2 días (48 hrs) de atraso no serán recibidas y serán evaluadas con la **calificación mínima (1.0)**.

10. Integridad académica y código de honor

Recuerde que esta evaluación está alineada al Código de Honor de la Universidad y se espera un comportamiento íntegro del estudiantado al momento de desarrollar. Dado lo anterior, se **enfatiza** que la solución de esta evaluación debe ser producto de un máximo 2 estudiantes que trabajaron en conjunto.

Finalmente, respecto al uso de herramientas generadoras de código. Estas están permitidas siempre y cuando se referencie correctamente cuándo es utilizado. Junto a lo anterior, se espera que **toda respuesta entregada por una herramienta generadora de código o de IA pase por un proceso de análisis crítico y modificación antes de ser incluida en una evaluación**. En caso de entregar una tarea donde se identifique que solo hubo un proceso de copiar la respuesta dada por una herramienta generadora de código o de IA, esto no será aceptado y será considerado como una falta a la integridad académica.