UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela de Ciencias de la Computación

Sistema Distribuido para Entrenamiento de IAs

con Algoritmo de Consenso RAFT

CC4P1 – Programación Concurrente y Distribuida

Final 2025-I

Integrantes

Apellidos y Nombres	Código
1. **Espinoza Valera, Cinver Alem**	**20220277F**
2. **Mercado Taype, Ariana Aracely**	**20221540B**
3. **Cadillo Tarazona, Jharvy Jonas**	**20210184E**
4. **Centeno Leon, Martin Alonso**	**20210161E**
5. **Calapuja Apaza, Luis Alberto**	**20150377G**

Lima, Perú
2025

Índice

1. Introducción **2**

2. Arquitectura del Sistema **2**
2.1. Visión General 2
2.2. Estructura de Directorios 3

3. Componentes del Sistema **4**
3.1. Cliente Python 4
3.2. Servidor Node.js 4
3.3. Core Java 4

4. Algoritmo de Consenso RAFT **5**
4.1. Estados del Nodo 5
4.2. Mensajes RPC 5
4.3. Proceso de Elección 5

5. Protocolo de Comunicación **6**
5.1. Formato de Mensajes 6
5.2. Configuración de Puertos 6

6. Flujo de Operaciones **6**
6.1. Fase 1: Entrenamiento 6
6.2. Fase 2: Predicción (Testeo) 7

7. Concurrencia y Paralelismo **7**
7.1. Multi-Threading en Java 7
7.2. Concurrencia en Node.js 7

8. Ejecución y Pruebas **7**
8.1. Comandos de Ejecución 7
8.2. Pruebas de Estrés 7

9. Conclusiones **8**

1. Introducción

El presente informe documenta el desarrollo de un **Sistema Distribuido para el Entrenamiento y Consumo de Modelos de Inteligencia Artificial**, implementado como proyecto final del curso de Programación Concurrente y Distribuida.

Objetivo del Sistema

Diseñar e implementar un sistema distribuido que permita el entrenamiento y consumo de modelos de IA de manera **paralela**, **distribuida** y **concurrente**, garantizando la gestión adecuada de los modelos mediante el algoritmo de consenso **RAFT**.

El sistema se desarrolló utilizando una arquitectura híbrida con tres lenguajes de programación:

- **Python**: Cliente Desktop con interfaz gráfica y scripts de pruebas
- **Node.js**: Servidores (Workers) con algoritmo RAFT y monitoreo HTTP
- **Java**: Motor de IA con redes neuronales y multi-threading

2. Arquitectura del Sistema

2.1 Visión General

El sistema implementa una arquitectura de tres capas que separa responsabilidades y cumple con el requisito de múltiples lenguajes de programación.

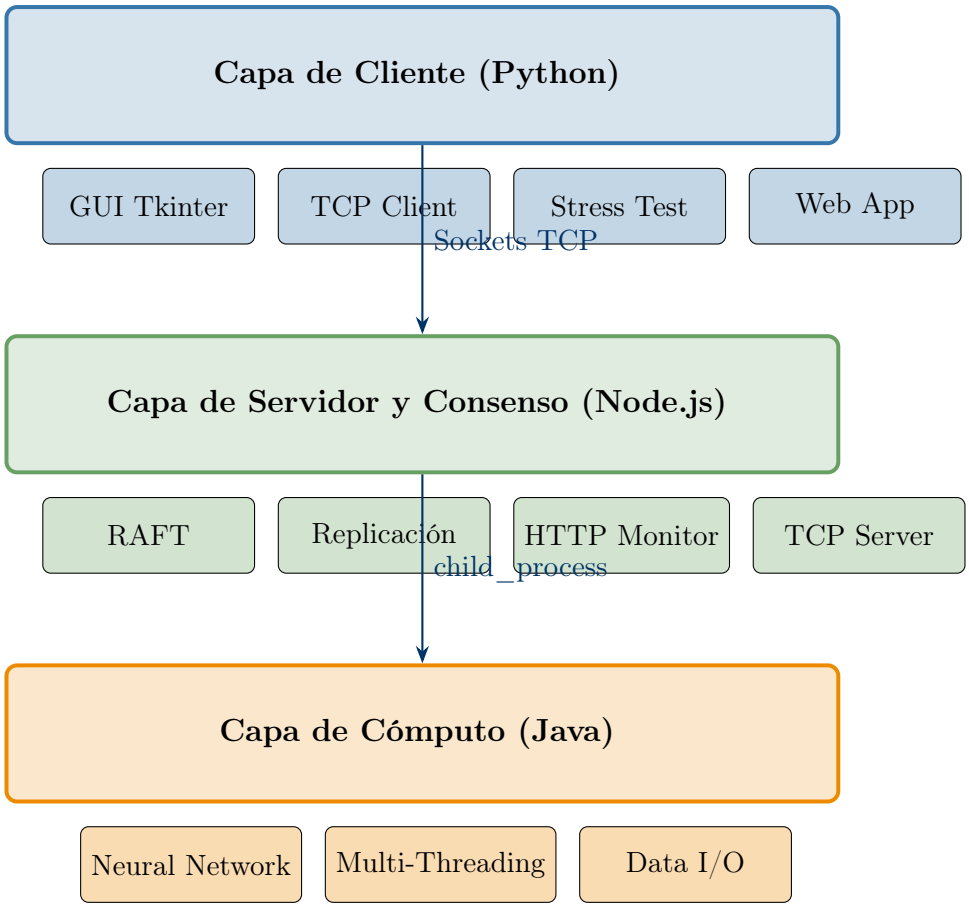

Figura 1: Arquitectura de tres capas del sistema

2.2 Estructura de Directorios

Organización del Proyecto

```
proyecto/
 src/
    client/          # Python: GUI, TCP Client, Web App
    server/          # Node.js: RAFT, TCP, HTTP
       node_worker/
           raft/    # Algoritmo de consenso
           tcp/     # Servidores TCP
           http/    # Monitor web
           java/    # Integración con Core
    core/            # Java: Red Neuronal
        src/
            nn/      # Clases de red neuronal
            math/    # Operaciones matriciales
            data/    # Carga de datos
            concurrent/ # Multi-threading
 Makefile
 README.md
```

3. Componentes del Sistema

3.1 Cliente Python

El cliente proporciona dos interfaces para interactuar con el sistema:

pythonblue!20 **Componente**	**Archivo**	**Función**
GUI Desktop	`gui_app.py`	Interfaz Tkinter para entrenamiento y predicción
Cliente Web	`web_app.py`	Interfaz Flask accesible por navegador
TCP Client	`tcp_client.py`	Comunicación con el clúster RAFT
Stress Test	`stress_test.py`	Pruebas de carga con 1000+ peticiones

Cuadro 1: Componentes del cliente Python

El cliente implementa **reconexión automática** cuando el líder RAFT cambia, rotando entre los nodos disponibles.

3.2 Servidor Node.js

Cada nodo del clúster ejecuta los siguientes módulos:

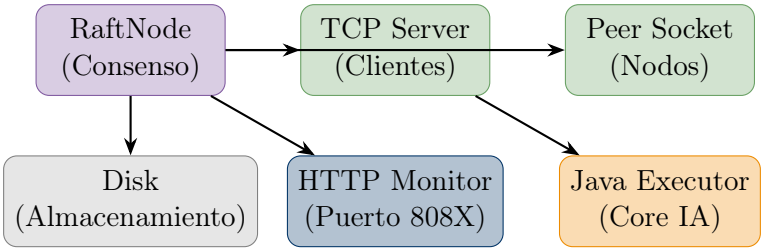

Figura 2: Módulos de un Worker Node.js

3.3 Core Java

El motor de IA implementa una red neuronal multicapa desde cero, sin frameworks externos:

- **Matrix.java**: Operaciones matriciales (multiplicación, transpuesta, Hadamard)
- **ActivationFunction.java**: Sigmoid, ReLU, Tanh, Softmax
- **NeuralNetwork.java**: Forward/Backpropagation, serialización
- **MultiThreadTrainer.java**: Entrenamiento paralelo usando `ExecutorService`
- **DataLoader.java**: Lectura de CSV y normalización de datos

Interfaz CLI del Core

```
java -jar core.jar train <input_path> <model_id>
java -jar core.jar predict <model_id> <input_data>
java -jar core.jar info
```

4. Algoritmo de Consenso RAFT

4.1 Estados del Nodo

El algoritmo RAFT implementado maneja tres estados:

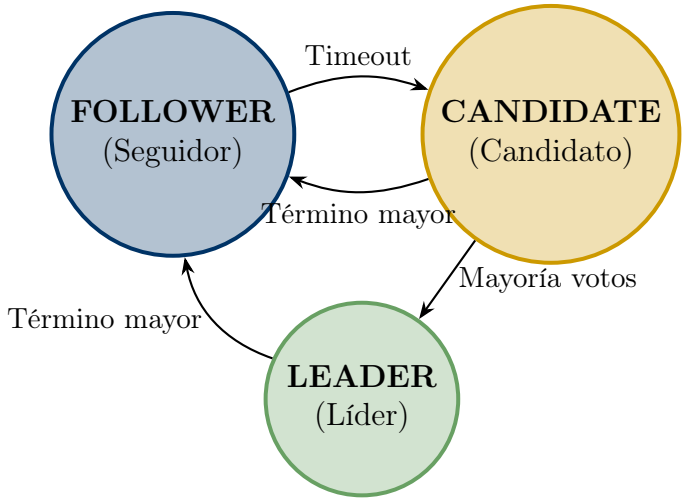

Figura 3: Máquina de estados RAFT

4.2 Mensajes RPC

raftpurple!20 **Tipo**	**Descripción**
`REQUEST_VOTE`	Solicitud de voto durante elección
`REQUEST_VOTE_RESPONSE`	Respuesta con voto otorgado o denegado
`APPEND_ENTRIES`	Heartbeats y replicación de logs
`APPEND_ENTRIES_RESPONSE`	Confirmación de replicación

Cuadro 2: Tipos de mensajes RAFT

4.3 Proceso de Elección

1. El nodo incrementa su término actual
2. Vota por sí mismo y envía `REQUEST_VOTE` a todos los peers
3. Espera respuestas hasta obtener mayoría (quórum)
4. Si obtiene mayoría, se convierte en líder
5. Si recibe heartbeat de líder válido, vuelve a follower

5. Protocolo de Comunicación

5.1 Formato de Mensajes

Todos los mensajes usan **JSON sobre TCP** con delimitador \n:

Listing 1: Petición de Entrenamiento

```
{
  "type": "TRAIN_REQUEST",
  "payload": {
    "data_content": "base64_encoded_csv",
    "model_name": "modelo_v1"
  }
}
```

Listing 2: Petición de Predicción

```
{
  "type": "PREDICT_REQUEST",
  "payload": {
    "model_id": "modelo_v1",
    "input_vector": [0.5, 0.1, 0.9]
  }
}
```

5.2 Configuración de Puertos

uniblue!20 **Nodo**	**Puerto Cliente**	**Puerto Peer**	**Puerto HTTP**
Node_A	5000	6000	8080
Node_B	5001	6001	8081
Node_C	5002	6002	8082

Cuadro 3: Configuración de puertos por nodo

6. Flujo de Operaciones

6.1 Fase 1: Entrenamiento

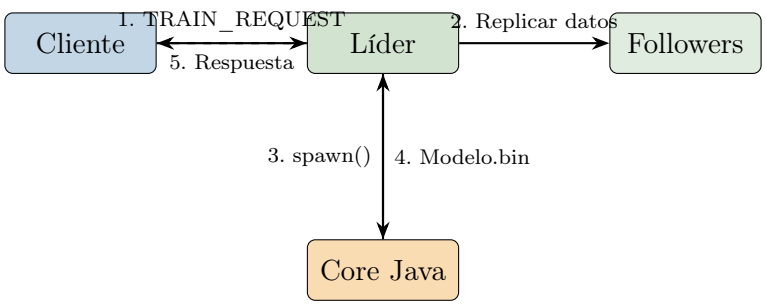

Figura 4: Flujo de entrenamiento distribuido

6.2 Fase 2: Predicción (Testeo)

El cliente envía una solicitud de predicción al líder, quien ejecuta el modelo entrenado en Java y retorna el resultado. El consenso RAFT garantiza que todos los nodos tienen acceso al modelo replicado.

7. Concurrencia y Paralelismo

7.1 Multi-Threading en Java

El componente `MultiThreadTrainer` divide el trabajo entre todos los núcleos disponibles:

Listing 3: Inicialización de threads

```
int numThreads = Runtime.getRuntime().availableProcessors();
ExecutorService executor = Executors.newFixedThreadPool(numThreads);
```

Técnica utilizada: *Paralelismo de datos* – cada thread procesa un batch diferente del dataset simultáneamente.

7.2 Concurrencia en Node.js

Node.js maneja múltiples conexiones TCP de forma asíncrona mediante su *event loop*, permitiendo:

- Conexiones simultáneas de múltiples clientes
- Comunicación con peers sin bloqueo
- Ejecución de Java como subproceso (`child_process.spawn`)

8. Ejecución y Pruebas

8.1 Comandos de Ejecución

Makefile

```
make core          # Compila el núcleo Java
make server-nodes  # Inicia el clúster de 3 nodos
make client        # Inicia la interfaz gráfica
make client-web    # Inicia el cliente web (puerto 5005)
make clean         # Limpia binarios
```

8.2 Pruebas de Estrés

El script `stress_test.py` envía 1000 peticiones aleatorias para validar:

- Consistencia del algoritmo RAFT bajo carga

- Manejo correcto de cambios de líder
- Rendimiento del sistema (requests/segundo)

9. Conclusiones

1. Se implementó exitosamente un sistema distribuido con **tres lenguajes de programación** (Python, Node.js, Java) comunicándose mediante sockets TCP.
2. El algoritmo **RAFT** garantiza la consistencia de datos entre nodos, manejando correctamente elecciones de líder y replicación de logs.
3. El motor de IA en Java aprovecha **todos los núcleos del procesador** mediante `ExecutorService` para entrenamiento paralelo.
4. La arquitectura modular permite escalar horizontalmente agregando más nodos al clúster.
5. El sistema cumple con las restricciones del proyecto: uso exclusivo de sockets nativos, sin frameworks de IA ni librerías de comunicación externas.

Repositorio del Proyecto

El código fuente completo está disponible en GitHub con documentación detallada de instalación y uso.