Sistema de Scrapping Web Distribuido

Proyecto de Sistemas Distribuidos

Implementación de un sistema distribuido para extracción automatizada de contenido web utilizando Docker Swarm

Estudiante: [Nombre del Estudiante]
Matrícula: [Número de Matrícula]
Profesor: [Nombre del Profesor]
Asignatura: Sistemas Distribuidos

Universidad [Nombre de la Universidad]

28 de noviembre de 2025

Índice

1 Introducción

1.1 Contexto del Proyecto

Los sistemas distribuidos han revolucionado la manera en que procesamos y analizamos grandes volúmenes de información en la era digital. En particular, el web scrapping o extracción automatizada de contenido web se ha convertido en una técnica fundamental para la recopilación de datos a gran escala, utilizada en campos que van desde el análisis de mercado hasta la investigación académica.

El presente proyecto implementa un **Sistema de Scrapping Web Distribuido** que aprovecha las ventajas de la computación distribuida para realizar extracción de contenido web de manera eficiente, escalable y tolerante a fallos. El sistema está diseñado bajo los principios fundamentales de los sistemas distribuidos: distribución de carga, escalabilidad horizontal, tolerancia a fallos y comunicación eficiente entre componentes.

1.2 Problemática Abordada

El scrapping web tradicional, ejecutado en un solo servidor, presenta limitaciones significativas:

- **Limitaciones de rendimiento**: Un solo nodo puede procesar un número limitado de páginas web simultáneamente
- **Falta de escalabilidad**: No es posible aumentar la capacidad de procesamiento dinámicamente
- **Punto único de falla**: El sistema completo se ve afectado si el servidor único falla
- **Ineficiencia en recursos**: No aprovecha completamente los recursos disponibles en múltiples máquinas
- **Limitaciones de red**: Una sola IP puede ser bloqueada por políticas anti-scrapping

1.3 Objetivos del Sistema

1.3.1. Objetivo General

Desarrollar un sistema distribuido que permita realizar scrapping web de manera eficiente, escalable y confiable, utilizando múltiples nodos de procesamiento coordinados a través de una arquitectura cliente-servidor distribuida.

1.3.2. Objetivos Específicos

1. **Distribución de carga**: Implementar un mecanismo de distribución de tareas de scrapping entre múltiples nodos trabajadores
2. **Escalabilidad dinámica**: Permitir la incorporación y desconexión de nodos scrapper sin interrumpir el funcionamiento del sistema
3. **Tolerancia a fallos**: Desarrollar mecanismos de detección y recuperación ante fallos de nodos individuales

4. **Coordinación distribuida**: Implementar un sistema de coordinación eficiente entre el servidor central y los nodos scrapper
5. **Monitoreo y gestión**: Proporcionar herramientas para el monitoreo del estado del sistema y gestión de recursos

1.4 Arquitectura General del Sistema

El sistema implementa una arquitectura distribuida basada en el patrón **Master-Worker** con las siguientes características:

- **Servidor Coordinador**: Nodo central que gestiona la cola de tareas y coordina a los nodos scrapper
- **Nodos Scrapper**: Trabajadores distribuidos que ejecutan las tareas de extracción de contenido
- **Gateway API**: Interfaz REST para interacción externa y monitoreo del sistema
- **Red Overlay**: Infraestructura de comunicación basada en Docker Swarm
- **Sistema de Descubrimiento**: Mecanismo automático para detección y registro de servicios

2 Componente Scrapper: Diseño e Implementación

2.1 Descripción General

El componente **Scrapper** constituye el núcleo operativo del sistema distribuido, funcionando como un nodo trabajador autónomo que se conecta dinámicamente al coordinador central para recibir y procesar tareas de extracción web. Cada instancia de scrapper opera de manera independiente, permitiendo que el sistema escale horizontalmente según las demandas de procesamiento.

2.2 Arquitectura del Scrapper

2.2.1. Diseño de Clases

El scrapper está implementado a través de la clase `ScrapperNode`, que encapsula toda la funcionalidad necesaria para:

- Autodescubrimiento de servicios coordinadores
- Gestión de conexiones de red persistentes
- Procesamiento asíncrono de tareas de scrapping
- Comunicación bidireccional con el coordinador
- Manejo de estados y recuperación ante fallos

2.2.2. Patrones de Diseño Implementados

1. **Patrón Observer**: Para el manejo de eventos de red y cambios de estado
2. **Patrón Producer-Consumer**: Para la gestión de colas de mensajes entre hilos
3. **Patrón State Machine**: Para el manejo de estados del scrapper (disponible, ocupado, desconectado)

2.3 Funcionalidades Principales

2.3.1. Autodescubrimiento de Servicios

El scrapper implementa un mecanismo de autodescubrimiento que permite detectar automáticamente coordinadores disponibles en la red:

```
def listen_for_broadcasts(self):
    """Escucha se ales de broadcast de los coordinadores"""
    broadcast_socket = socket.socket(socket.AF_INET, socket.
       SOCK_DGRAM)
    broadcast_socket.bind(('', self.broadcast_port))

    while not self.connected:
        data, addr = broadcast_socket.recvfrom(1024)
        message = json.loads(data.decode())

        if message.get('type') == 'coordinator_discovery':
            coordinator_host = message.get('coordinator_host')
            coordinator_port = message.get('coordinator_port')

            if self.connect_to_server():
                break
```

Listing 1: Mecanismo de Autodescubrimiento

Ventajas del autodescubrimiento:

- Eliminación de configuración manual de endpoints
- Detección automática de coordinadores en la red
- Adaptabilidad a cambios en la topología de red
- Simplificación del despliegue de nuevos nodos

2.3.2. Gestión de Conexiones

El sistema implementa un modelo de conexiones persistentes con gestión de hilos especializada:

```
def _send_worker(self):
    """Hilo que env a mensajes desde la cola"""
    while self.connected and not self.stop_event.is_set():
        try:
            message = self.message_queue.get(timeout=1.0)
            if message is None:
```

```
                break

            # Protocolo de env o con longitud prefijada
            length = len(message)
            self.socket.send(length.to_bytes(2, 'big'))
            self.socket.send(message)

        except queue.Empty:
            continue
        except Exception as e:
            logging.error(f"Error enviando mensaje: {e}")
            self.connected = False
            break
```

Listing 2: Gestión de Conexiones Persistentes

Características de la gestión de conexiones:

- **Protocolo binario optimizado**: Uso de longitud prefijada para eficiencia
- **Cola de mensajes thread-safe**: Comunicación asíncrona entre hilos
- **Reconexión automática**: Detección y recuperación de desconexiones
- **Timeouts configurables**: Prevención de bloqueos indefinidos

2.3.3. Procesamiento de Tareas de Scrapping

El núcleo del procesamiento de tareas integra el módulo especializado `scrapper.py`:

```
def execute_task(self, task_id, task_data):
    """Ejecuta la tarea asignada"""
    logging.info(f"Ejecutando tarea {task_id}: {task_data}")

    # Marcar como ocupado
    self.update_busy_status(True)

    try:
        url = task_data['url']

        # Realizar scrapping usando m dulo especializado
        scrape_result = get_html_from_url(url)

        # Preparar resultado optimizado
        result = {
            'url': scrape_result['url'],
            'html_length': len(scrape_result['html']),
            'links_count': len(scrape_result['links']),
            'links': scrape_result['links'][:10],  # Primeros 10
               enlaces
            'status': 'success'
        }

    except Exception as e:
```

```
        result = {
            'status': 'error',
            'error': str(e)
        }
    finally:
        # Marcar como disponible
        self.update_busy_status(False)
```

Listing 3: Ejecución de Tareas de Scrapping

Optimizaciones implementadas:

- **Gestión de estado automática**: Actualización automática de disponibilidad
- **Resultados optimizados**: Transmisión de metadata en lugar de contenido completo
- **Manejo robusto de errores**: Captura y reporte de fallos sin afectar el sistema
- **Integración modular**: Uso del módulo `scrapper.py` sin modificaciones

2.3.4. Sistema de Heartbeat y Monitoreo

Implementación de un sistema de latidos para mantener la conectividad:

```
def send_heartbeat(self):
    """Env a se al peri dica al servidor"""
    while self.connected:
        try:
            heartbeat_msg = {
                'type': 'heartbeat',
                'client_id': self.client_id,
                'time_now': datetime.now().isoformat()
            }

            self._enqueue_message(heartbeat_msg)
            time.sleep(60)  # Heartbeat cada 60 segundos

        except Exception:
            self.connected = False
```

Listing 4: Sistema de Heartbeat

2.4 Ventajas del Diseño Distribuido del Scrapper

2.4.1. Escalabilidad Horizontal

- Capacidad de agregar nodos scrapper dinámicamente
- Distribución automática de carga entre nodos disponibles
- Adaptación automática a variaciones en la demanda

2.4.2. Tolerancia a Fallos

- Detección automática de fallos de nodo
- Redistribución de tareas ante fallos
- Reconexión automática de nodos recuperados

2.4.3. Eficiencia de Recursos

- Procesamiento paralelo de múltiples URLs
- Utilización optimizada de recursos de red
- Balanceado automático de carga de trabajo

2.4.4. Flexibilidad de Despliegue

- Despliegue en múltiples máquinas físicas
- Compatibilidad con contenedores Docker
- Integración con orquestadores como Docker Swarm

3 Conclusiones Parciales

El componente Scrapper representa una implementación robusta de un nodo trabajador distribuido que combina eficiencia, escalabilidad y tolerancia a fallos. Su diseño modular y uso de patrones de sistemas distribuidos establecidos proporciona una base sólida para el procesamiento distribuido de tareas de web scrapping.

Las características implementadas, como el autodescubrimiento de servicios, la gestión de conexiones persistentes y el sistema de heartbeat, demuestran la aplicación práctica de conceptos fundamentales de sistemas distribuidos en un contexto real de procesamiento de datos web.