Problema:

Implemente un sistema de caché distribuido utilizando un lenguaje de su elección.

Factores a considerar:

1. Almacenar y recuperar datos de manera eficiente.
2. Manejar problemas de consistencia distribuida.
3. Proporcionar mecanismos para la invalidación y caducidad del almacenamiento en caché.

Solución:

Considerando los factores necesitamos desglosar los requerimientos funcionales y no funcionales implícitos.

Factor: Almacenar y recuperar datos de manera eficiente.

Con este factor podemos extraer los siguientes requerimientos funcionales:

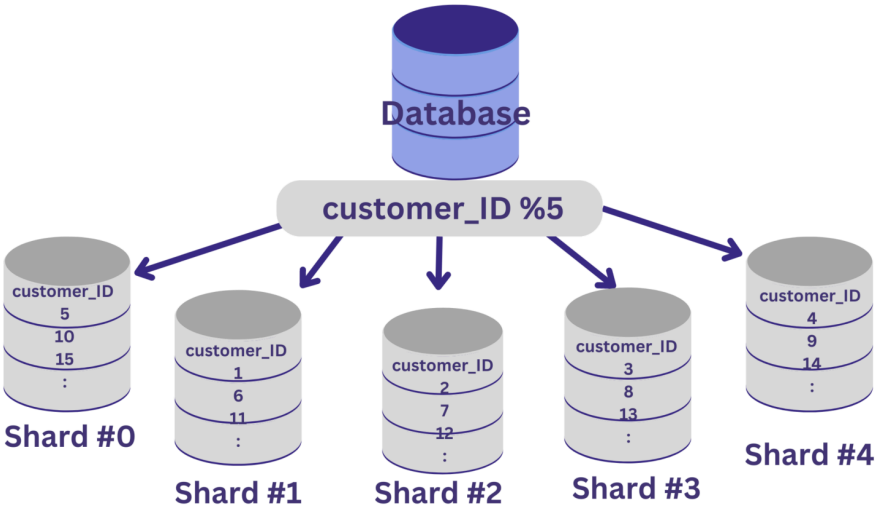
- Funcional para get(key)
- Funcional para put(key, value, expires_in?)

Factor: Manejar problemas de consistencia distribuida.

Aquí aparece uno de los retos más grandes del diseño: la consistencia. Si simplemente replicamos toda la información en todos los nodos, terminaríamos con N × M datos almacenados (donde N es la cantidad de claves y M la cantidad de servidores). Eso significa mucho espacio desperdiciado y, peor aún, la sincronización cuando tenemos escalas de datos muy grandes en cada nodo, lo cual es caro y poco eficiente.

Por eso, el enfoque de “replicar todo” queda descartado. La idea sería buscar un mecanismo que permita repartir las claves de manera balanceada, evitandonos redundancias innecesarias y reduciendo los dolores de cabeza de sincronización. entonces surgen de aquí distintas estrategias posibles para enfrentar el problema.

Sharding:

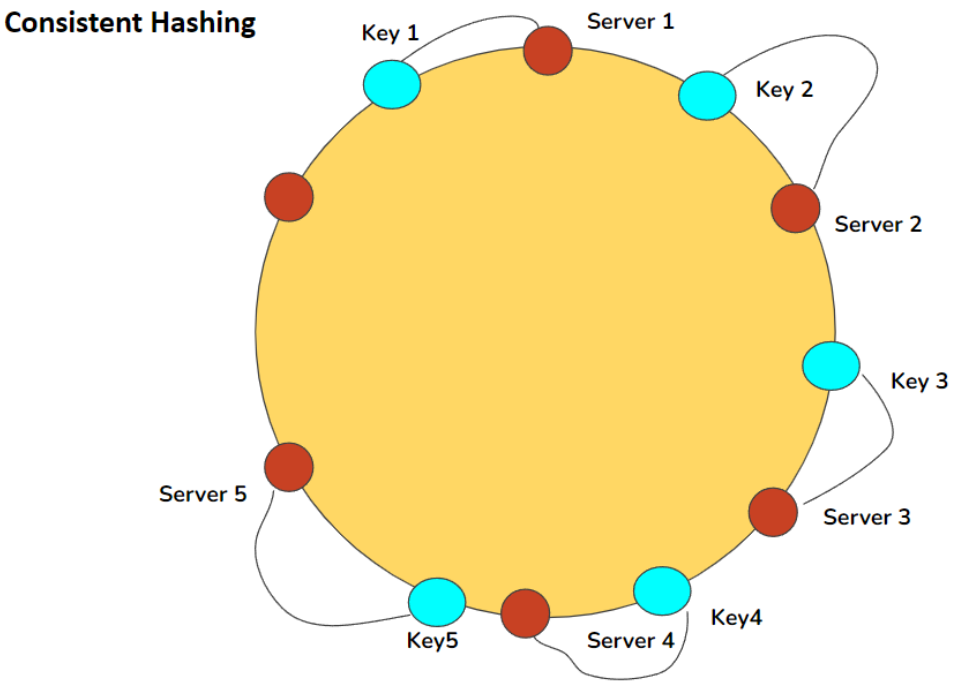


Una de las primeras alternativas sería usar **sharding**, consiste en dividir los datos en particiones y asignar cada una a un nodo específico. La idea es simple: en lugar de que todos los servidores guarden todo, cada uno guardaría solo “su pedazo”.

El problema está en la definición de esas particiones. Si usamos algo muy estricto (ejm, dividir por rangos de clave), terminamos con un número fijo de divisiones que no escalan nada bien cuando agregamos o quitamos nodos. Además, puede generar desbalance: algunos nodos cargan mucho más trabajo que otros si la distribución de claves no es pareja.

Consistent Hash:

Con este enfoque dividiremos el rango de los hashes consistentemente sin importar la cantidad de nodos que entren. Tendríamos que tener en cuenta la replicación de los datos de cada servidor, podríamos implementar rangos para que los hashes se distribuyan más uniformemente(Nodos virtuales) y así podríamos generar “réplicas” a cada servidor para solucionar el problema de la consistencia distribuida.



En comparación con el sharding, la ventaja del consistent hashing es que no estamos atados a un número fijo de particiones. Esto lo hace mucho más práctico en escenarios reales donde la infraestructura cambia con el tiempo. Es por eso que para este caso opté por consistent hashing: me da escalabilidad, flexibilidad y evita la necesidad de redefinir las particiones de forma estática.

Factor: Proporcionar mecanismos para la invalidación y caducidad del almacenamiento en caché.

Para solucionar este problema tengo en mente las siguientes posibilidades:

1. Verificación al get(key): fácil de implementar, pero la problemática principal es que podríamos tener valores en nuestro caché que ya expiraron ocupando espacio de memoria.
2. Subproceso + eliminación lineal: fácil de implementar, sin embargo, nos encontraríamos con una complejidad de O(N) cada vez que ejecutemos.
3. Subproceso + eliminación cuando expire(Cola de prioridades): Para esta solución necesitamos reorganizar una pila cada vez que insertemos, lo que nos daría una complejidad de O(log n) en un buen caso.
4. Subproceso + Timing Wheel: Un timing wheel agrupa expiraciones por “ranuras de tiempo” (buckets); el reloj avanza en “ticks” donde worker solo revisa el bucket(es decir las posibles claves que expirarán) del tick actual.

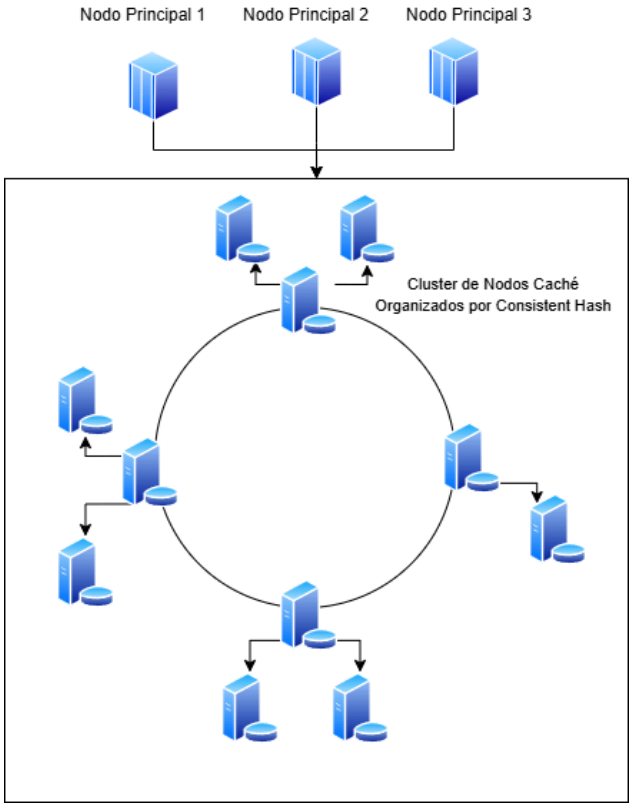
Cumpliremos con el factor haciendo una combinación del primer y cuarto punto

Consideraciones

1. El programa se iría a un memory leak si no controlamos la memoria, por lo que deberíamos poder tener un límite y seguir funcionando. Para solucionar esto tenemos varias opciones como:
    a. **LRU(Least Recently Used):** elimina primero los elementos que llevan más tiempo sin usarse.
    b. **LFU(Least Frequently Used):** en lugar de mirar el tiempo, se enfoca en la frecuencia: descarta las claves menos consultadas. Puede ser más justo en algunos escenarios, pero necesita contadores y es más costoso de mantener.
2. **Evitar las carreras:** como todo estaremos en un entorno concurrente debemos encontrar una manera de no modificar al mismo tiempo los datos y hacer bloqueos a nivel de hilo para modificar. Para eso es clave usar estructuras de datos seguras para concurrencia o locks ligeros, y también pensar en versiones de los objetos (ejemplo: descartar solo si la versión coincide).

Arquitectura Propuesta

Se propone un servicio de orquestación y balanceo que conoce a todos los servidores de caché. El balanceo de claves se hará con consistent hashing (con réplicas virtuales para balancear mejor). Los servidores se pueden desplegar como nodos maestros de partición (dueños del rango del id anillo) y como réplicas de esos maestros..



En el cual tendremos un anillo de nodos caché organizados de tal manera que si añadimos otro nodo maestro muestra construir la red circular. Todo esto lo lograremos con el consistent hash asignándoles ids a nuestras conexiones. Por lo que nuestros componentes quedarían:

1. **Nodo Principal:** En nuestra arquitectura este nodo construirá el consistent hash circular de los nodos caché a partir de los ids de estos y su tipo, lo que buscamos lograr es que este nodo sea un balanceador de carga y sincronizador, pero no que contenga ni maneje los datos.
2. **Nodo Principal Caché:** Será un Nodo que definirá el hash principal en la estructura circular, pero luego de eso se comportará como una instancia más en nuestro array de réplicas.
3. **Nodo Replica Caché:** Servirá para añadir una copia al nodo principal de caché y también. Si no existen nodos principales la réplica simplemente no se añadirá ya que no contendría un “Hash fuerte”. Sin embargo, en una implementación un poco más completa por la forma de nuestra arquitectura podríamos asignar la réplica como maestro.

Balanceo y Sincronización

La estructura de la red se organiza como un mapa de grupos de nodos. Cada grupo corresponde a un shard o partición y contiene un nodo principal (leader) junto con sus réplicas. De esta forma, al resolver una clave con consistent hashing, obtenemos directamente el grupo responsable de manejarla.

- Hash: principal_node_id
    - SocketNodoPrincipal
    - SocketReplica1
    - SocketReplica1
- Hash: principal_node_2
    - SocketNodoPrincipal
    - SocketReplica1

Dentro de cada grupo:

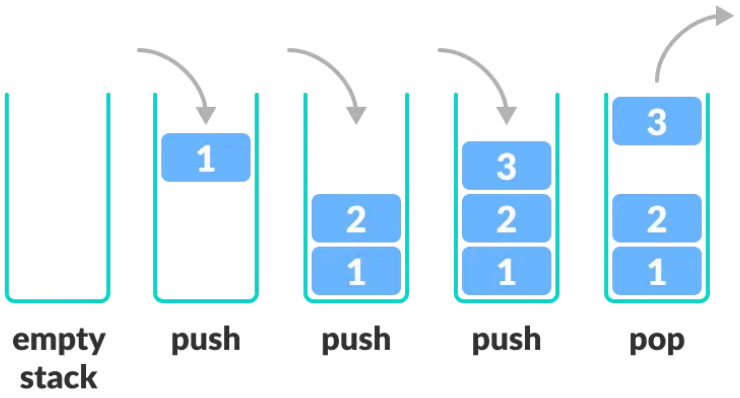
El nodo balanceador recibe las operaciones del cliente (PUT o GET) y, a partir del anillo de consistent hashing, determina el grupo de nodos que debe atender la petición. Una vez identificado el nodo maestro y sus réplicas, la petición se envía a todos los sockets del grupo en paralelo.

En un PUT, esto asegura que el valor quede almacenado al mismo tiempo en el maestro y en sus réplicas, manteniendo sincronizadas todas las copias del rango de claves. En un

GET, la petición también se envía a todos, de modo que se puede responder con el primero que conteste, o bien con el maestro si se quiere asegurar la versión más reciente.

Si alguno de los nodos (ya sea maestro o réplica) falla, simplemente se elimina de la lista de nodos disponibles y la sincronización continúa con los que siguen activos. La estructura no depende de un orden específico, lo importante es que todos los nodos de un shard reciban las operaciones en paralelo para garantizar consistencia y tolerancia a fallos.

En resumen estamos usando una combinación entre consistent hasher y un sharding "inteligente" para la una implementación de caché distribuido.



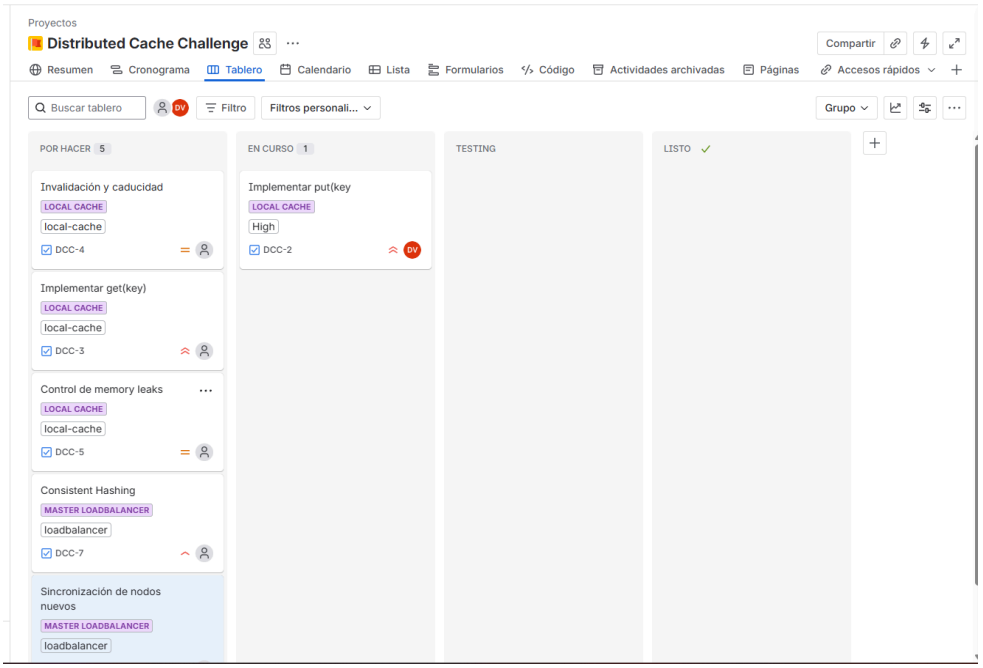
Bosquejo de Tareas a Realizar

Definiendo un sistema sencillo de pesos de 1 a 5, donde 1 es facil y 5 muy difícil, tenemos las siguientes tareas:

1. Local Cache:
    a. Implementar método put(key, value, expires_in?) (1, highest)
    b. Implementar método get(key) (1, highest)
    c. Implementar control de Memory Leaks (4, Low)
    d. Invalidación y caducidad de caché (4, Medium)
2. Master LoadBalancer:
    a. Implementar Consistent Hashing para Nodos (5, Highest)
    b. Sincronización de Nodos nuevos (5, Highest)
3. Arquitectura limpia
    a. Implementar clean architecture o arquitectura hexagonal para las apps principales.
    b. Implementar test para los casos de uso y servicios que pertenezcan al dominio.
4. Cliente:
    a. Crear un cliente que se conecte a los nodos principales y pueda obtener el caché de cualquiera de ellos.

5. Integración Continua
    a. Implementar un github action con el fin de priorizar la integridad del código en cada cambio.

Por lo que el tablero utilizando la metodología Kanban quedaría de la siguiente manera:



Stack tecnológico

TCP

El sistema se comunica sobre **TCP** porque es rápido, confiable y con bajo overhead frente a HTTP. Nos asegura entrega ordenada y retransmisión en caso de pérdida. El reto es que debemos implementar manualmente el protocolo de aplicación: definir mensajes, manejar el framing y controlar timeouts y reconexiones.

Rust

Elegí **Rust** como lenguaje principal. Si bien tengo experiencia en otros lenguajes (JavaScript, TypeScript, Go, etc.), Rust me pareció el más adecuado por varias razones:

- **Performance cercano a C/C++**, lo que es clave en un servicio de caché que debe responder en milisegundos.

- **Control explícito de concurrencia y asincronía**: con `async/await` y el modelo de ownership, puedo manejar miles de conexiones simultáneas sin bloquear hilos y al mismo tiempo evitar los *race conditions*.

- **Seguridad en memoria sin garbage collector**: el compilador de Rust asegura que no haya *data races* ni accesos inválidos, algo crítico en un sistema que estará corriendo de forma continua en producción.

- **Flexibilidad para diseño bajo nivel**: puedo decidir cuándo usar threads, cuándo usar `Arc`, `Mutex` o canales (`mpsc`) para comunicación entre tareas, según lo que más convenga.

Arquitectura (Clean + Hexagonal)

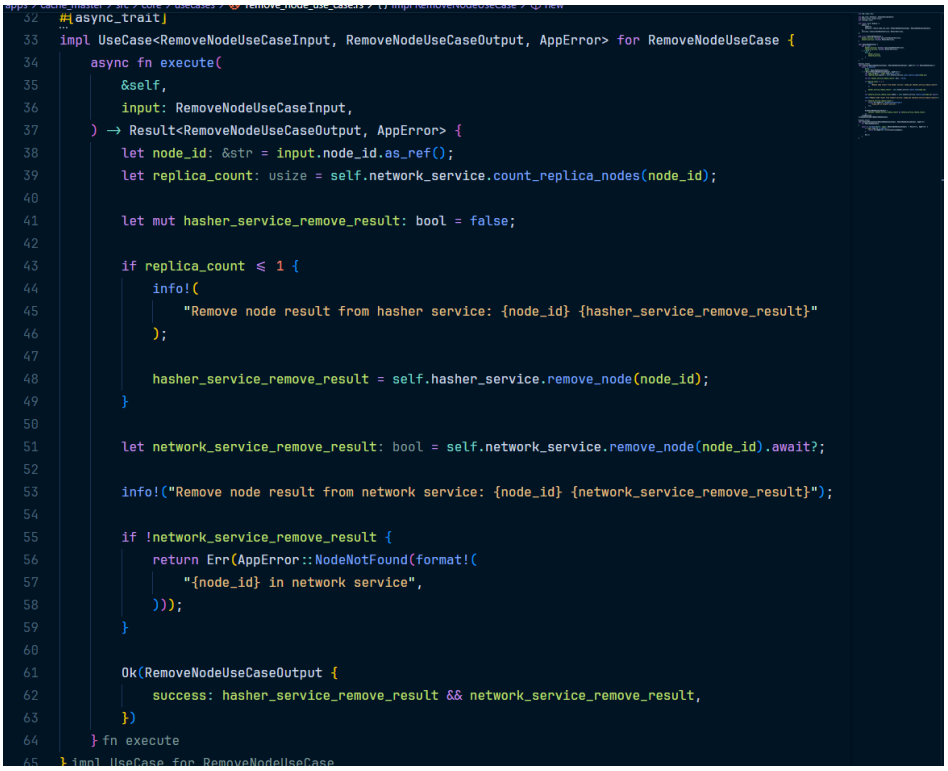
Implementé una mezcla de Clean Architecture y Hexagonal Architecture. Organicé el código en models, use cases, services y adapters, lo que permite:

- Separar la lógica de negocio del detalle técnico.
- Sustituir adaptadores sin modificar la lógica central.
- Mantener el sistema extensible y más fácil de probar.

En resumen, **TCP** me da un canal ligero para comunicaciones, y **Rust** me da la combinación de velocidad y seguridad necesaria para construir el sistema de caché. Junto con mi arquitectura custom puedo hacer el código más entendible y fácil de mantener.

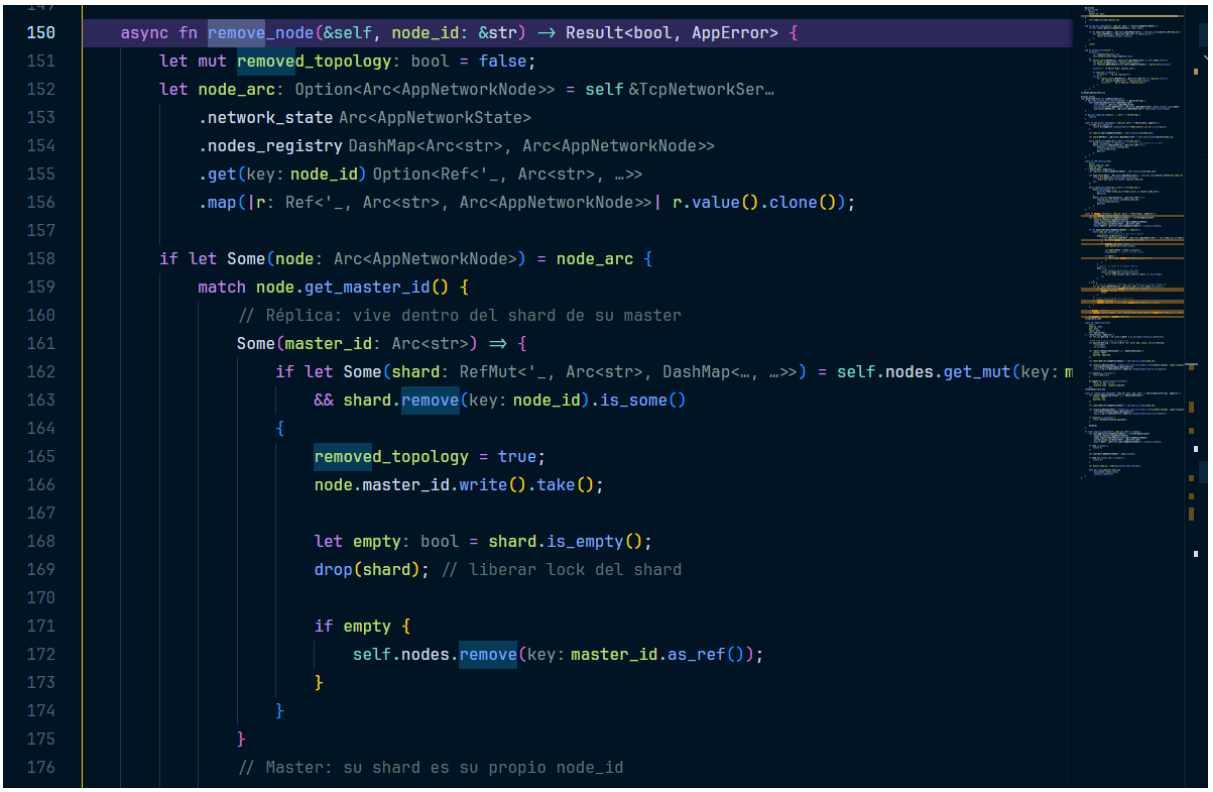
Anotaciones:

Si bien rust es un lenguaje muy genial y rápido, a veces es muy difícil no hacer código poco entendible(como caer en el boilerplate). Por lo que será más fácil leer la clean architecture implementada desde la carpeta core ejm: apps/cache_master/core (https://github.com/davp00/J-Challenge/tree/main/apps/cache_master/src/core). EJM: la lógica para eliminar un nodo.



```rust
#[async_trait]
impl UseCase<RemoveNodeUseCaseInput, RemoveNodeUseCaseOutput, AppError> for RemoveNodeUseCase {
    async fn execute(
        &self,
        input: RemoveNodeUseCaseInput,
    ) -> Result<RemoveNodeUseCaseOutput, AppError> {
        let node_id: &str = input.node_id.as_ref();
        let replica_count: usize = self.network_service.count_replica_nodes(node_id);

        let mut hasher_service_remove_result: bool = false;

        if replica_count <= 1 {
            info!(
                "Remove node result from hasher service: {node_id} {hasher_service_remove_result}"
            );

            hasher_service_remove_result = self.hasher_service.remove_node(node_id);
        }

        let network_service_remove_result: bool = self.network_service.remove_node(node_id).await?;

        info!("Remove node result from network service: {node_id} {network_service_remove_result}");

        if !network_service_remove_result {
            return Err(AppError::NodeNotFound(format!(
                "{node_id} in network service",
            )));
        }

        Ok(RemoveNodeUseCaseOutput {
            success: hasher_service_remove_result && network_service_remove_result,
        })
    } fn execute
} impl UseCase for RemoveNodeUseCase
```

Su implementación ya en la infraestructura se vuelve un poco más compleja por la cantidad de adaptadores y detalles técnicos que hay que manejar (sockets, concurrencia, sincronización, etc.), lo que hace aún más valioso tener una separación clara entre la lógica de negocio y la infraestructura.

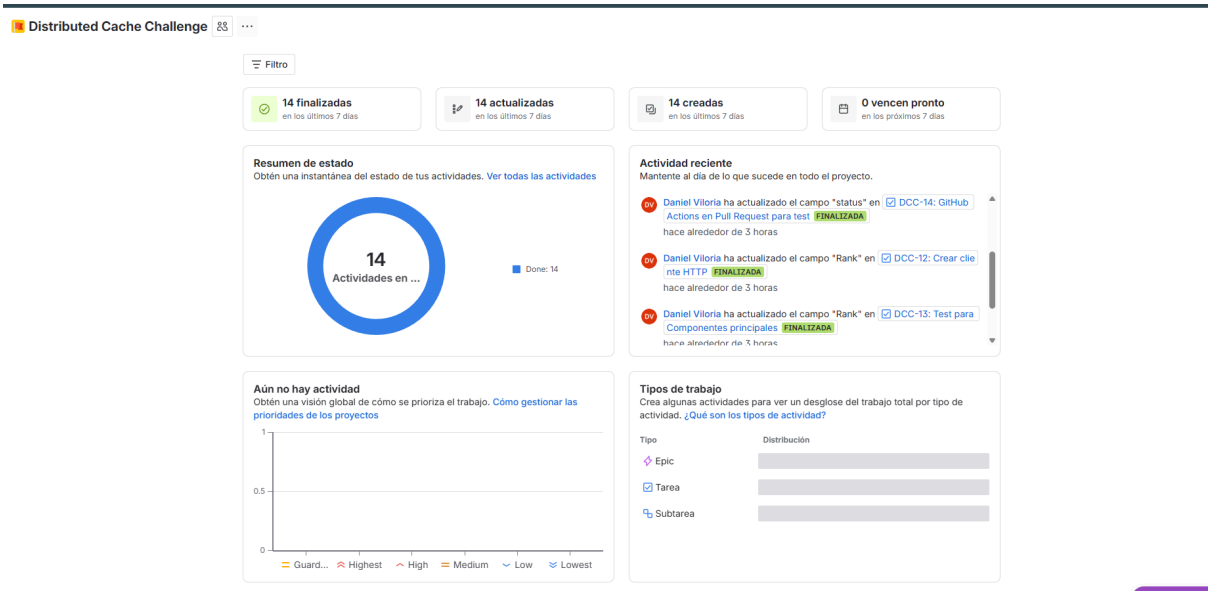


```rust
async fn remove_node(&self, node_id: &str) -> Result<bool, AppError> {
    let mut removed_topology: bool = false;
    let node_arc: Option<Arc<AppNetworkNode>> = self &TcpNetworkSer…
        .network_state Arc<AppNetworkState>
        .nodes_registry DashMap<Arc<str>, Arc<AppNetworkNode>>
        .get(key: node_id) Option<Ref<'_, Arc<str>, …>>
        .map(|r: Ref<'_, Arc<str>, Arc<AppNetworkNode>>| r.value().clone());

    if let Some(node: Arc<AppNetworkNode>) = node_arc {
        match node.get_master_id() {
            // Réplica: vive dentro del shard de su master
            Some(master_id: Arc<str>) => {
                if let Some(shard: RefMut<'_, Arc<str>, DashMap<…, …>>) = self.nodes.get_mut(key: m
                    && shard.remove(key: node_id).is_some()
                {
                    removed_topology = true;
                    node.master_id.write().take();

                    let empty: bool = shard.is_empty();
                    drop(shard); // liberar lock del shard

                    if empty {
                        self.nodes.remove(key: master_id.as_ref());
                    }
                }
            }
            // Master: su shard es su propio node_id
```

Conclusiones

Este proyecto fue un reto bastante grande debido a la cantidad de factores y algoritmos que había que tener en cuenta: desde el manejo de consistencia distribuida, hasta la caducidad de los datos y la orquestación de nodos en un anillo de consistent hashing. La combinación de Rust, TCP y una arquitectura limpia (mezcla de Clean + Hexagonal) permitió mantener orden y claridad dentro de la complejidad.

Nuestro tablero de kanban ha quedado así:



Oportunidades de mejora

- Aunque TCP fue una buena elección por su simplicidad y velocidad, la implementación de **parseo de mensajes** podría ser más robusta, con un framing más claro y tolerancia a errores.
- Posibilidad de **promover un nodo réplica a maestro** de manera más automatizada y segura.
- Mejorar la **gestión de nodos caídos**, incluyendo sincronización incremental cuando un nodo vuelve a la red.
- Optimizar la **estrategia de invalidación** para soportar escenarios más complejos (ej. invalidaciones masivas o por patrones).
- Extender la arquitectura para **métricas y monitoreo nativo** (latencias, aciertos/fallos de caché, consumo de memoria).
- Evaluar algoritmos alternativos de **eviction** (además de LRU/LFU) para cargas específicas.
- **Mejorar la asignación de las réplicas**, puesto que debemos esperar a que el primer nodo master inicie.

- DAVP