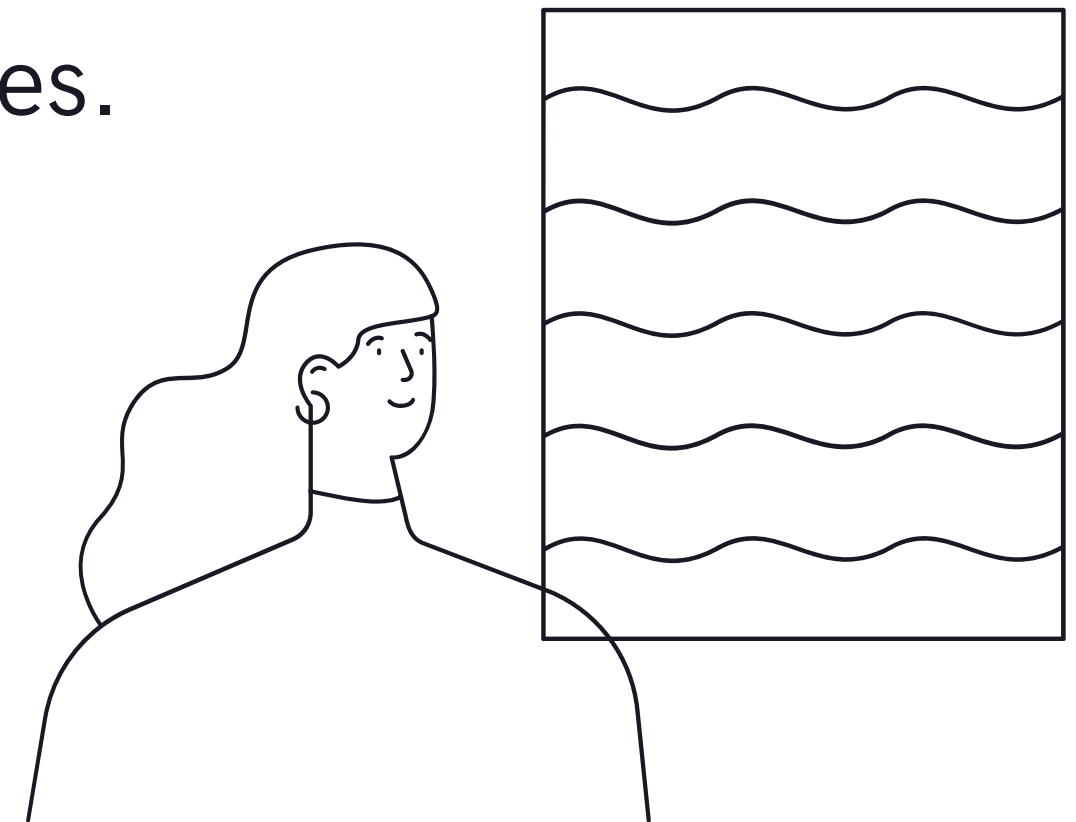


Cache Server Implementation in Rust

Aarya Aamish Bhatt
Aastha Bharill
Arjun Sankholkar
Ishan Nerlekar

WHAT IS A CACHE SERVER?

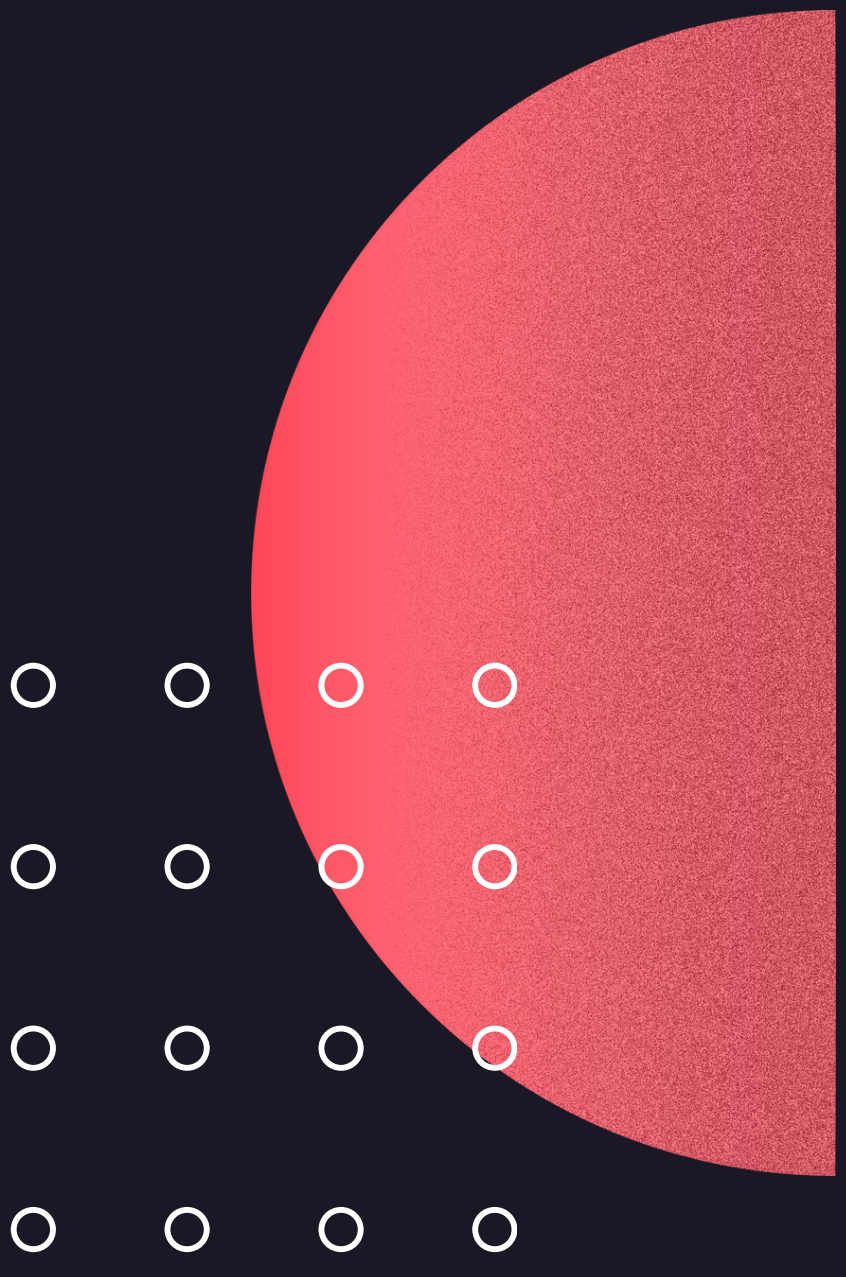
- Cache Server Overview:
 - Specialized server for data storage.
 - Stores frequently accessed data in memory.
- Accelerated Data Retrieval:
 - Speeds up data access.
 - Reduces reliance on slower data storage like databases.
- Enhanced Application Performance:
 - Boosts application responsiveness.
 - Provides rapid access to commonly used data.



FUNCTIONALITIES OF REDIS SERVER 03

1. **Reduced Latency:** Cached data is readily available, minimizing the time needed to fetch it from slower data sources.
2. **Lower Database Load:** Frequent queries are redirected to the cache, reducing the strain on the database.
3. **Improved Scalability:** Scaling by adding more cache servers is straightforward.
4. **Custom Expiry:** Cache data can be set to expire after a certain time, ensuring freshness.

Problem Statement

- 
1. Fast Data Retrieval: Cache servers like Redis and Memcached offer lightning-fast data access due to in-memory storage, ideal for applications needing swift access to frequently used data.
 2. Reduced Database Load: Storing frequently accessed data in a cache eases the burden on relational databases, resulting in better performance, quicker responses, and cost savings, especially in read-heavy workloads.
 3. Caching Complex Queries: Cache servers can store results of intricate queries, eliminating the need for repetitive calculations and saving valuable processing time.

RUST PRINCIPLES INCORPORATED

05

Ownership and Borrowing:

Arc (Atomic Reference Counting)

- `main_conns` and `store` are declared as `Arc<Mutex<HashMap<usize, Conn>>>` and `Arc<Mutex<Store>>` respectively. `Arc` provides shared ownership over its content, and `Mutex` helps manage concurrent mutable access to the shared data.

```
let main_conns = Arc::new(Mutex::new(HashMap::new()));  
let store = Arc::new(Mutex::new(Store::new()));
```

RUST PRINCIPLES INCORPORATED

06

References:

Function Parameters

- Functions like `event_data`, `handle_command`, and others take references (&) to arguments instead of taking ownership of them. This allows them to borrow data for the duration of the function call without taking full ownership. For example, `fn event_data(..., store: &Arc<Mutex<Store>>)`.

```
fn event_data(_id: usize, input: &mut Vec<u8>, store: &Arc<Mutex<Store>>) -> (Vec<u8>, bool) {  
    let mut output = Vec::new();  
    let mut close = false;  
    let mut i = 0;  
    let mut argss = Vec::new();
```

RUST PRINCIPLES INCORPORATED

07

Lifetimes:

The code uses lifetime annotations for references where necessary, indicating how long the referenced data is valid.

Lifetime annotations are typically used when defining functions that accept references with different lifetimes.

The 'static lifetime is used for string literals, which have a static lifetime and are always available for the entire program's duration.

RUST PRINCIPLES INCORPORATED

08

Shadowing:

In this code snippet, the variable `threads` is shadowed when it is redefined with a new value. It initially holds the parsed value of the command-line argument, and if the value is not present or invalid, it takes on the value returned by `num_cpus::get()`

```
let threads: usize = matches ArgMatches<'_>  
    .value_of(name: "threads") Option<&str>  
    .map(|t: &str| t.parse::<usize>().unwrap_or(default: 1)) Option<usize>  
    .unwrap_or_else(|| num_cpus::get());
```


RUST PRINCIPLES INCORPORATED

09

Traits:

The code uses various Rust traits implicitly, such as `std::io::Read` and `std::io::Write` for reading from and writing to `TcpStream`.

Traits like `std::fmt::Debug` and `std::fmt::Display` are implemented for debugging and formatting purposes, although they are not explicitly shown in the provided code.

RUST PRINCIPLES INCORPORATED

10

Dangling Pointers:

In the `event_data` function, this code handles the removal of processed data from the input vector. It ensures that if `i` is within the valid bounds of the vector, the data is cleared and, if needed, transferred to a new vector (`remain`). This approach avoids leaving any dangling pointers to the processed data.

```
if i > 0 {  
    if i < input.len() {  
        let mut remain: Vec<u8> = Vec::new();  
        remain.extend_from_slice(&input[i..input.len()]);  
        input.clear();  
        input.extend(iter: remain)  
    } else {  
        input.clear()  
    }  
}
```

RUST V/S C++

Memory Safety.

- Rust: Rust is known for its strong memory safety guarantees. It provides features like ownership, borrowing, and lifetimes that help prevent common programming errors such as null pointer dereferences, buffer overflows, and data races. This makes Rust a safe choice for building high-performance systems with reduced risk of memory-related bugs.
- C++: C++ allows for manual memory management, which means developers have more control over memory allocation and deallocation. However, this flexibility comes with a higher risk of memory-related bugs, like memory leaks and dangling pointers. Developers need to be diligent in managing memory properly.

RUST V/S C++

Concurrency and Parallelism

- Rust: Rust has a powerful concurrency model that makes it easier to write safe and concurrent code. It provides features like ownership and borrowing, which allow multiple threads to access and modify data without data races. Rust's ownership system enforces thread safety at compile time.
- C++: C++ provides a rich set of concurrency features, but it's up to the developer to ensure thread safety. This can be more error-prone and challenging, especially in large codebases.

RUST V/S C++

Ecosystem:

- Rust: Rust has a growing ecosystem of libraries and tools for various purposes, including building network services like cache servers. Libraries like tokio and actix provide high-level abstractions for asynchronous I/O and networking.
- C++: C++ has a mature ecosystem with many libraries and frameworks for building network services. Popular choices include Boost. Asio for asynchronous I/O and various web frameworks.

Commands

- SET key value
- GET key
- DEL key
- KEYS pattern
- FLUSHDB
- QUIT
- PING