Rust vs C++ Concurrency

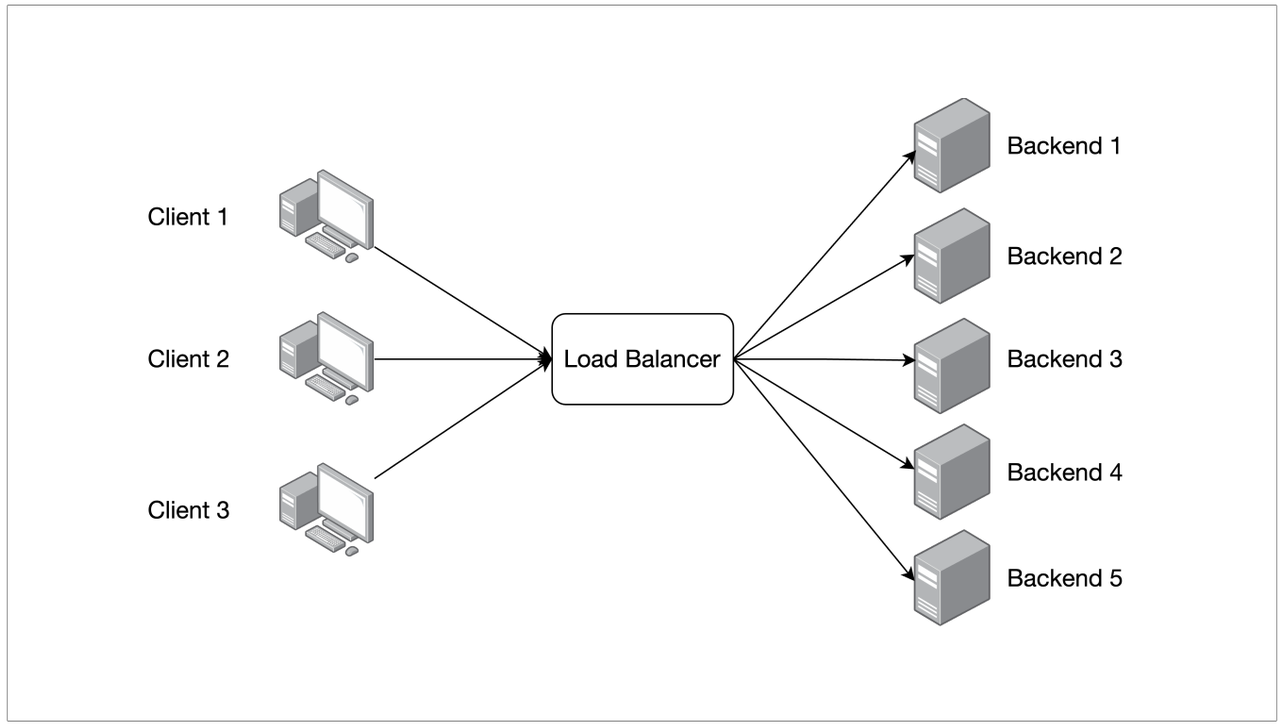
Zühlke Group - 10.09.24 - Samuel Gauthier

Why Concurrency Matters

- Needed for modern software performance (scaling, efficiency).
- Crucial for handling multiple tasks at once, like network requests.

Languages:

- C++: Powerful but complex concurrency mechanisms.
- Rust: Designed for memory safety and fearless concurrency.



What is a Load Balancer?

- Distributes incoming requests across multiple servers.
- Essential for efficiency and avoiding server overload.
- Accepting requests, dispatching to threads, managing connections.

Demo

C++ Concurrency

- `std::thread`, `std::jthread`
- `std::mutex` & `std::lock`
- `std::counting_semaphore`
- `co_await`, `co_return`

Main Libraries Used:
std::thread:
- Basic threading model.

std::jthread:
- Joins when out of scope

std::mutex, std::unique_lock:
- Locking mechanisms for shared resources.

std::counting_semaphore
co_await, co_return, std::future, std::promise:
- Asynchronous execution (usually you use a framework where the promises are already implemented, makes life easier)

Advantages:
- Flexible and customizable.
- Fine-grained control for performance tuning.

Challenges:
- Risk of race conditions and deadlocks.
- Manual management of threads, memory, and synchronization.

Show code in Load Balancer
Show small snippets

Rust Concurrency

- `std::thread`
- `std::sync::mpsc`
- `std::sync::Mutex, RwLock`
- `std::sync::Arc`
- `async, await`

Concurrency Tools:
std::thread:
- Safe threads with ownership guarantees.

std::sync::mpsc:
- Channels for message passing between threads. (**m**ulti **p**roducer, **s**ingle **c**onsumer)

std::sync::Mutex, RwLock:
- Manage shared state safely.

async/await:
- Efficient asynchronous operations.

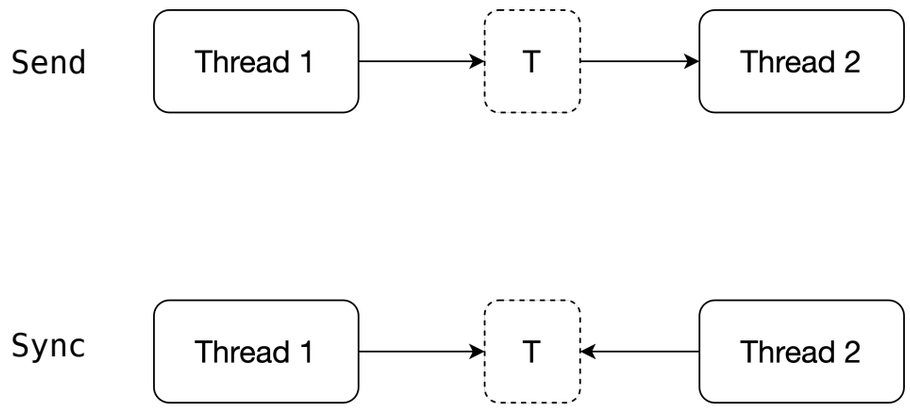
std::sync::Arc: Atomically Reference Counted
- Shared ownership of a type T allocated on the heap
- Does not allow mutable access to T
- Makes sure that all references are gone before dropping contained value
- Thread safe

Advantages:
- Ownership and borrowing system ensure memory safety.
- Data races are prevented at compile-time.
- “Fearless Concurrency”: The compiler enforces safety.

Challenges:

- Learning curve (ownership and lifetimes).
- More rigid compared to C++, but safer.

Sync and Send Traits



Send is for moving ownership of values across threads.
Sync is for sharing references to values across threads.

- (Allmost) all types are Sync + Send (except raw pointers, UnsafeCell and Rc)
- UnsafeCell allows you to get a mutable reference to data, &mut T
- It’s up to you to not do anything dangerous
- Rc is Reference Count
- Does not enforce anything about concurrent modification

Show Load balancer code
Show simpler snippet

Conclusion

- Safe Rust avoids data race and most common concurrency issues
- More flexibility with C++ but also more DYI and know-what-you-do
- Concurrency is indeed hard to get right

References

(1) A Comparison of Concurrency in Rust and C (https://ehnree.github.io/documents/papers/rustvsc.pdf)

(2) Comparing Rust's and C++'s Concurrency Library (https://blog.m-ou.se/rust-cpp-concurrency/)

(3) Exploring Concurrency Pitfalls: Rust vs. C++ and Go (https://www.youtube.com/watch?v=goughuZfpnc)

(4) Concurrency chapter of Rustonomicon (https://doc.rust-lang.org/nomicon/concurrency.html)

(5) The Rust Standard Library (https://doc.rust-lang.org/)