Concurrency and Async Practices in Real-World Rust: How are lock-free and atomic-based concurrency techniques used in Rust crates?

[1] David Potskhishvili [2] Andreea Costea, Ruben Backx

[1]Student, TU Delft [2]Responsible Professor, Supervisor TU Delft

Introduction

Rust is a modern programming language. Through its ownership model and type system, Rust prevents data races and many memory-safety errors at compile time, while offering multiple concurrency mechanisms such as shared-memory synchronization, message passing, asynchronous programming, data parallelism, and low-level atomics. Prior work studied Rust concurrency bugs [1, 2] and adaptations of C++ concurrency patterns to Rust [3]. This project empirically analyzes atomic-based techniques in open-source Rust crates, focusing on used primitives and the concurrency patterns built on top of them. The main research question of this paper is:

How are lock-free and atomic-based concurrency techniques used in Rust crates?

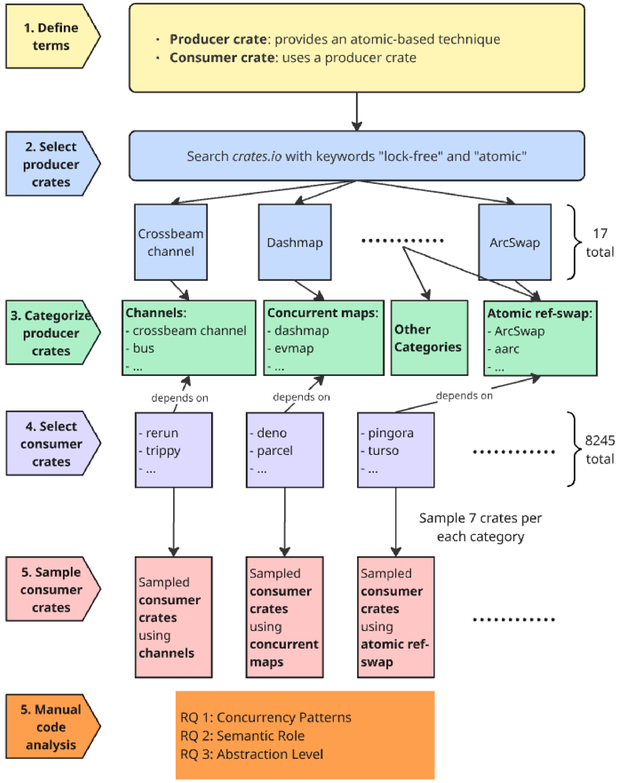
To structure the research, the main question is divided into the following sub-questions:

RQ 1. What concurrency patterns built on atomic primitives occur in open-source Rust crates?

RQ 2. What semantic role do atomic-based techniques play in open-source Rust crates?

RQ 3. What is the abstraction level of atomic-based techniques in open-source Rust crates? How far is the code from the raw atomic primitives?

Methodology



Manual code analysis

RQ 1. We reviewed producer-crate documentation to identify functionality and API calls, then searched these APIs in sampled consumer crates. By inspecting which threads call the APIs and what data is shared or transferred, we identified the corresponding concurrency pattern.

RQ 2. We determined the semantic role from the identified pattern together with variable names, method names, and surrounding code in sampled consumer crates.

RQ 3. We determined the abstraction level from crates.io metadata, documentation, and, when needed, manual inspection of the producer crate's source code.

Final per category statistics

Table 1. Producer crate example, categories, and dependent consumer crates counts.

Example Crate	Category	#Dependent crates
`crossbeam-skiplist`	concurrent map	4,155
`crossbeam-channel`	channels	2,300
`arc-swap`	atomic reference swap	1,536
`atomic`	no-std atomics	465
`crossbeam-queue`	lock-free queue	408
`ringbuf`	ring buffer	220
`crossbeam-deque`	work-stealing deque	109

Results

RQ1 & RQ2. Concurrency patterns and semantic roles.

1. Channel-like communication: transferring data between threads

```rust
let (tx, rx) = crossbeam_channel::unbounded();
tx.send(WorkerResult::Entry(entry)); // thread 1 sends data through the channel
let worker_result = rx.recv();       // thread 2 receives data from the channel
```

2. Atomic reference swapping: replacing shared references

```rust
backends: ArcSwap<BTreeSet<Backend>>;
backends.load_full(); // reader thread atomically loads an Arc reference to the shared object
backends.store(Arc::new(new_backends)); // writer thread atomically stores a new Arc reference
 ↪  to the shared object
```

3. Shared storage access: shared storage between threads

```rust
workers: DashMap<Arc<str>, WorkerContext>;
let worker = match workers.entry(worker_name) { // Any thread can check the key
    Entry::Occupied(entry) => { ... } // Case when the key exists in the map
    Entry::Vacant(entry) => { ... } // Case when the key doesn't exist in the map
};
```

4. Work-stealing deque: scheduling work on demand

```rust
local.pop() // worker tries to get a task from its local queue
    .or_else(|| stealers.iter().find_map(|s| s.steal().success())) // otherwise, tries to
     ↪  steal a task from another worker
    .or_else(|| global.steal().success()); // otherwise, tries to get a task from the global
     ↪  queue
```

RQ3. Abstraction levels of atomic-based techniques.

1. Low-level pointer management: high-level API hides memory details

```rust
pub struct ArcSwapAny<T: RefCnt, S: Strategy<T> = DefaultStrategy> {
    ptr: AtomicPtr<T::Base>, // atomic pointer
    strategy: S, // strategy of dealing with pointer
}
```

2. Lock-free data structures: direct atomic coordination

```rust
struct Inner<T> {
    front: AtomicIsize, // front task index
    back: AtomicIsize,  // back task index
}
inner.front.compare_exchange(f, f.wrapping_add(1), ...); // try incrementing front index
```

3. Partially atomic implementations: atomics as internal detail

```rust
struct RawRwLock {
    state: AtomicUsize, // atomic state
}
// Thread tries to acquire the lock using an atomic compare_exchange operation
while state & ONE_WRITER == 0 {
    match state.compare_exchange_weak(state, state | ONE_WRITER, ...) { ... }
}
```

Summary of abstraction levels

Abstraction level	Example	#Crates
Low-level pointer management	`ArcSwap`	2
Lock-free based on atomic operations	`crossbeam-deque`	10
Partially implemented using atomics	`crossbeam-channel`	5

Interpretation of Results

Channel-like communication is broader than channels and also appears in other crates with different API abstractions.

Some crates extend Rust's concurrency toolbox rather than introducing a new concurrency pattern.

Atomic-based techniques manage shared resources. Concurrent maps, queues, and work-stealing deques let threads access data, submit work, or coordinate execution.

Semantic roles depend on application context. The same pattern can transfer results, store connections, map names to workers, or schedule tasks.

High-level APIs can hide low-level internals. Simple public APIs may rely internally on atomics, pointer management, memory reclamation, or platform-specific primitives.

Discussion

Ownership transfer. We assume that channel-like patterns appear often because they fit Rust's ownership model: values can be moved between threads without shared mutable access.

Safe API, low-level internals. Our results might suggest that Rust concurrency crates often hide atomics, raw pointers, memory reclamation, and `unsafe` code behind safer high-level APIs.

Workload-specific coordination. We hypothesize that atomic-based crates are useful when the workload requires small concurrent operations, such as transferring data, accessing shared storage, replacing references, or submitting work.

References

[1] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. *Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs*, 2020.

[2] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sanchez Munoz, Oli Scherer, and Qian Wang. *Miri: Practical Undefined Behavior Detection for Rust*, 2026.

[3] Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C.Jeffrey. *When Is Parallelism Fearless and Zero-Cost with Rust?*, 2024.