Concurrency and Async Practices in Real-World Rust

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

David Potskhishvili
Supervisor(s): Andreea Costea, Ruben Backx
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: David Potskhishvili
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Ruben Backx, Przemyslaw Pawelczak

Abstract

Rust provides strong safety guarantees and supports several forms of concurrency, including atomic-based techniques. In this paper, we study how such techniques are used in crates that provide and use atomic-based functionality. We observed that these techniques mainly support communication between threads, shared access to data, and coordination of work. We also observed that they appear at different abstraction levels, from low-level memory management to higher-level data structures. These results help explain how atomic-based concurrency is used in real-world Rust code.

1 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. Previous research has examined the types of concurrency-related bugs that can still occur in Rust [1, 2], as well as how C++ concurrency patterns are adapted into idiomatic Rust [3]. This project aims to empirically study *how concurrency is used in real-world Rust applications* by analyzing open-source crates. In this paper, we focus specifically on atomic-based techniques and their applications in open-source Rust projects.

The main research question of this paper is:

How are 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?

To answer these questions, we select atomic-based crates from `crates.io`, collect their dependent crates, and manually analyze sampled usages. We observe that atomic-based techniques appear as channel-like communication patterns, extensions of Rust's standard concurrency functionality, concurrent data structures, and work-stealing schedulers. We further observe that these techniques are used to transfer data between threads, provide shared storage, organize work scheduling, and replace shared references atomically. Finally, we classify their abstraction levels into low-level pointer management, lock-free implementations based on atomic operations, and implementations that use atomics only partially or in a platform-dependent way. The rest of the paper introduces the required background, describes the methodology, presents the analysis and results, discusses possible explanations, compares the results with related work, and concludes with future work.

2 Background

2.1 Rust ownership and concurrency

One of the main Rust's feature is **ownership** model [4]. Each value has exactly one owner, and once that owner goes out of scope (lifetime boundaries of the value), the memory owned by the value is freed. The correct handling of ownership in Rust is enforced by the compiler. Code snippet 1 demonstrates the basic idea. On line 2, we create the variable a, which owns the string `"hello"`. On line 3, ownership of the string is transferred from a to b. After this transfer, a can no longer be accessed, since it no longer owns the value, as shown on line 5. The scope itself is bounded with curly brackets on lines 1 and 6.

```
1 fn main() { // Scope begins
2     let a = String::from("hello");
3     let b = a;
4     println!("{}", b);
5     // println!("{}", a); // ERROR: value borrowed after move
6 } // Scope ends
```

Code snippet 1: Ownership transfer example in Rust

Shared ownership between threads can be expressed using `Arc` [5], an atomically reference-counted pointer whose reference count denotes the number of references to a pointer. `Arc<T>` allows multiple threads to share the same value. However, `Arc` only provides shared ownership; it does not by itself make mutation of the underlying safe in multi-thread environment. In code snippet 2 we can observe on line 1 how variable *five* is `Arc` that stores integer. On line 3, we create a cloned `Arc` value, which increments the reference counter and transfers ownership of the clone to a new thread on line 4.

```
1 let five = Arc::new(5); // initializing Arc with integer value
2 for _ in 0..10 {
3     let five_cloned = Arc::clone(&five); // creating new Arc pointer
4     thread::spawn(move || { // ownership transfer to a new thread
5         println!("{five_cloned:?}");
6     });
7 }
```

Code snippet 2: `Arc` shared ownership example

2.2 Atomic-based techniques and lock-free paradigm

Atomic types provide primitive shared-memory communication between threads. Atomic types support atomic operations, which guarantee that each operation is performed fully by only one thread at a time. For example, Rust provides atomic types such as `AtomicBool`, `AtomicUsize`, and `AtomicPtr`, as well as atomic operations such as `load`, `store`, and `compare_exchange` [6]. These operations allow values to be read, written, and conditionally updated atomically (i.e. only by one thread).

The lock-free paradigm [7] is a concurrency approach where shared data is updated without mutual-exclusion locks while guaranteeing that at least one thread can make execution progress despite delays or failures of other threads.

In code snippet 3 we can see one of the possible implementations of a lock-free algorithm that atomically sets a maximum for unsigned integer.

```rust
fn atomic_set_max(x: &AtomicUsize, new_value: usize) {
    let mut current = x.load(); // read atomically current max value
    while new_value > current {
        // atomically and conditionally write a new value to x, otherwise try
        ↪ again
        current = x.compare_exchange(current, new_value).unwrap_or_else(|x| x);
    }
}
```

Code snippet 3: Atomic maximum set

2.3 Rust's ecosystem

One of the parts of Rust's ecosystem is the crate, Rust's unit of code packaging and reuse. A crate is a library or executable package that can be built, versioned, and shared through Cargo, Rust's official package manager. Example of crates include `tokio` [8].

3 Methodology

In this section, we describe how the research was conducted in order to answer the research question and its sub-questions. We outline how we determined the dataset of open-source Rust repositories to analyze and how the source-code analysis was performed. Figure 1 visualizes the full methodology pipeline described throughout this section and should be followed together with the text.

3.1 Selecting relevant open-source Rust repositories

We first define the main terms used throughout the rest of the paper.

A **producer crate** is a crate that provides developers with a specific atomic-based technique. For example, a producer crate may provide a lock-free queue data structure.

A **consumer crate** is a crate that uses a producer crate for some functionality. For example, a consumer crate may use a producer crate that provides a lock-free queue for event processing between several threads.

To determine the set of producer and consumer crates, we used crates.io [9], the default registry used by Cargo, Rust's official package manager. This makes crates.io a suitable source for this research.

3.1.1 Selecting producer crates

To determine producer crates, we searched crates.io using the keywords "lock-free" and "atomic". All producer crates with fewer than 50 dependent consumer crates were removed.

This threshold was chosen empirically: such producer crates accounted for less than five percent of all consumer crates, and ninety percent of them had zero dependent consumer crates, making them clear outliers. Additionally, producer crates that were tools for testing or building atomic-based techniques, rather than application-level concurrency abstractions, were removed because they are outside the scope of this research. Since discovery on crates.io relies on public package metadata, such as descriptions, keywords, README files, and repository links, crates that intentionally provide lock-free or atomic-based concurrency abstractions are likely to mention these terms. However, we acknowledge the limitation that this strategy does not guarantee complete coverage, since some relevant crates may use different terminology or rely on atomics internally without advertising this in their metadata.

The keyword search resulted in 17 producer crates. Each producer crate was categorized based on the author's explicit description of its functionality on crates.io. Table 1 shows example producer crate and its category.

3.1.2 Selecting consumer crates

Using the identified producer crates, we fetched all dependent consumer crates and merged duplicates, since one consumer crate may depend on several producer crates. This resulted in 8245 consumer crates in total. Each consumer crate was assigned to the categories by its corresponding producer crates. Table 1 shows the number of dependent consumer crates per category.

Table 1: Producer crate examples, 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

3.2 Consumer crates analysis

In this subsection, we describe how the consumer crates were analyzed.

For each category, we uniformly sampled 7 consumer crates and analyzed them manually. Since our research questions are qualitative and comparative, we sample the same number of consumer crates from each category and prioritize category coverage rather than frequency estimation.

For each research question, the analysis was conducted as follows:

> **RQ 1.** For each category, we reviewed the official documentation of each producer crate to identify its functionality, required imports, and API names, such as method or function calls. We then searched for these API names in the sampled consumer crates to find code fragments that use the producer crate. We examined which threads use the producer crate's API calls, what data is shared between threads, and what data is transferred according to Rust's ownership model. These criteria reveal the

communication mechanism between threads by showing how memory is handled and what role the producer crate's APIs play. This makes it possible to identify the corresponding concurrency pattern.

RQ 2. For each category, the semantic role was determined from variable and method names in the sampled consumer crates, together with the pattern identified during the analysis for RQ 1. The author's naming choices provide natural-language semantics that help explain how the code fits into the abstraction of an atomic-based pattern.

RQ 3. This research question was answered by analyzing the categories of the producer crates. Based on the author's description in the crates.io metadata, we extracted the abstraction level of each producer crate. If the author did not provide an explicit description, we manually analyzed the internal codebase of the producer crate to include supporting evidence. This process reveals the abstraction level by relying either on the author's explicit description or on manual code inspection.

4 Analysis & Results

In this section, we analyze each research question according to the methodology described above. We focus on the functionality of each producer-crate category to determine its abstraction level and examine which concurrency patterns and semantic roles can be observed in its consumer crates. Each subsection presents the analysis, corresponding results, interpretation, and answer for a specific research question.

4.1 RQ 1 & 2. Analysis of atomic-based concurrency patterns and semantic roles

In this subsection, we present our analysis of RQ 1 and RQ 2 according to the methodology defined above. We examine how consumer crates in each category correspond to a concurrency pattern and what semantic role they play.

4.1.1 Channels

By analyzing the documentation of crates in the channels category, we observed that the channel interface is usually split into a sending side and a receiving side, for example `Sender<T>` and `Receiver<T>`. The sender provides an operation such as `send`, while the receiver provides operations for receiving messages.

To better understand this category, we consider the simplified Code snippet 4. We can notice two threads involved: thread 1 (line 3) sends a worker result, while thread 2 (line 5) receives this result through the sending endpoint `tx` and the receiving endpoint `rx` respectively. This suggests us a communication pattern where one thread communicates with another by sending data through a channel. Therefore, in this example, the semantic role of a channel can be described as a bridge that allows threads to transfer worker results from a sender to a receiver.

```
let (tx, rx) = crossbeam_channel::unbounded();
// Thread 1 sends and transfers ownership of the worker result
tx.send(WorkerResult::Entry(entry));
// Thread 2 receives the worker result
let worker_result = rx.recv();
```

Code snippet 4: Channel-based ownership transfer example

4.1.2 Atomic Reference Swapping

By analyzing the documentation of crates in this category, we observed that atomic reference swapping is a way to share a reference between threads while also allowing that shared reference to be replaced atomically. The main API is a swap cell, such as `ArcSwap<T>`. It stores an object wrapped in `Arc<T>`. Reader threads use operations such as `load` or `load_full` to atomically load the current `Arc<T>`. Writer threads use operations such as `store` to atomically store a new `Arc<T>`.

To better understand this category, we consider the simplified Code snippet 5. We can notice how reader thread 1 (line 3) accesses a shared resource, a `BTreeSet` in this example, by atomically loading an `Arc` object. Meanwhile, thread 2 (line 6) can atomically store a new reference, which can later be used by other reader threads. This suggests us the following communication pattern: reader threads atomically access a referenced object to read the necessary data, while writer threads atomically update the reference when needed. We therefore infer the semantic role: the shared resource, in this example a set of backends, is accessed to observe the currently available backends and updated when the set of available backends changes.

```
backends: ArcSwap<BTreeSet<Backend>>; // Definition of atomic reference swapping
// Thread 1 atomically gets the current reference Arc<BTreeSet<Backend>>
backends.load_full();
// Thread 2 atomically stores a new reference.
// Current readers will not fail, since they continue reading from the old Arc.
backends.store(Arc::new(new_backends));
```

Code snippet 5: Atomic reference swapping example

4.1.3 Concurrent Map

By analyzing the documentation of crates in this category, we observed that a concurrent map is a key-value collection that permits access from multiple threads through map operations. Its interface usually resembles an ordinary map, with operations such as `get`, `insert`, `remove`, or `entry`. The important difference is that the map object itself can be shared between threads while these operations coordinate thread-safe access to it internally.

To better understand this category, we consider the simplified Code snippet 6. We can notice how any thread (line 3) can access the map and try to access a value by providing a key through the `entry` method. This suggests us a pattern where different threads access shared storage to retrieve or insert the data they need. In this example, we might infer that

the semantic role of the map is to associate a worker name with the corresponding worker context in a multi-thread environment.

```
workers: DashMap<Arc<str>, WorkerContext>; // Definition of the concurrent map
// Any thread can access the map and try to either insert or get a worker by name
let worker = match workers.entry(worker_name) {
    Entry::Occupied(entry) => entry.get().clone(),
    Entry::Vacant(entry) => {
        let ctx = WorkerContext::spawn_for(input);
        entry.insert(ctx.clone());
        ctx
    }
};
```

Code snippet 6: Concurrent map example

4.1.4 Work-stealing deque

By analyzing the documentation of crates in this category, we observed that concurrent work-stealing deques are data structures used in work-stealing schedulers. The setup consists of worker threads, which execute tasks and have their own local task queues, a global queue called an injector, which can receive tasks from non-worker threads, and stealers, which are references to worker queues that allow tasks to be stolen from other workers.

To better understand this category, we consider the simplified Code snippet 10. We can notice how each worker thread has a `local` task queue (line 2) and access to `stealers` (line 3) for other workers' queues. There is also a `global` queue (line 5) that accepts tasks from external threads.

To find a new task, we can observe that a worker thread first checks its `local` queue (line 14), then tries to steal a task from another worker (line 16), and finally checks whether tasks were pushed by external threads into the `global` queue (line 18).

This example suggests us the following pattern. There are two types of threads: workers and non-workers. Worker threads interact with each other by stealing tasks when needed through special stealing-queue APIs. Non-worker threads interact with worker threads through the injector queue API where they can provide new tasks. As a result, this communication forms a work-stealing task-scheduling pattern 2. We therefore infer that the semantic role of this pattern is to organize worker threads as a resource for executing application tasks on demand.

4.1.5 Concurrent queue

By analyzing the documentation of crates in this category, we observed that a concurrent queue is a queue that can be shared among threads. It provides two main APIs: push and pop which allows thread-safely add and remove elements from the queue respectively. To better understand this category, we consider the simplified Code snippet 7. We can observe that any thread (line 4) can push an object to the queue, while any thread (line 6) can also pop an object from it. This suggests the following concurrency pattern: different threads push and pop objects from a shared queue, which is used to process data in queue order. We

therefore might infer the following semantic role in this example: some threads push new connections into the queue, while other threads process these connections in queue order.

```
use crossbeam_queue::ArrayQueue; // Bounded concurrent queue data structure
hot: ArrayQueue<(ConnId, T)>; // Definition of the concurrent queue
// Any thread can push a tuple connection object and its id to the queue
hot.push((id, connection));
// Any thread can pop from the queue a tuple with connection
let connection = hot.pop();
```

Code snippet 7: Concurrent queue example

4.1.6 Ring buffer

By analyzing the documentation of crates in this category, we observed that a ring buffer is a fixed-capacity lock-free buffer where one thread can write data into the buffer and another thread can read data from it. It provides two main APIs: `push` and `pop`, which allow a thread to write data into the buffer and read data from the buffer, respectively.

To better understand this category, we consider the simplified Code snippet 8. We can observe two threads involved: a writer thread and a reader thread, which share the same buffer. This suggests the following concurrency pattern: one thread (line 6) writes data into the buffer, while another thread (line 8) reads data from the buffer and processes it. We therefore infer the following semantic role: the ring buffer acts as shared fixed-capacity storage between two threads that write and read information through it. In this specific example, the shared storage contains sound samples.

```
use ringbuf::{traits::{Producer, Consumer, Split}, HeapRb}; // import ring buffer
// Initialise a buffer and split it into writer and reader objects for respective
↪  threads
let rb = HeapRb::<f32>::new(capacity);
let (mut writer, mut reader) = rb.split();
// Writer thread writes to the buffer
writer.push_slice(input_samples);
// Reader thread reads from the buffer
let copied = reader.pop_slice(output_samples);
```

Code snippet 8: Ring buffer example

4.1.7 Non-standard atomics

By analyzing the documentation of crates in this category, we observed that non-standard atomics provide additional atomic types that are not part of Rust's standard-library atomics. For example, the `portable-atomic` crate supports 128-bit atomic integer types.

To better understand this category, we consider the Code snippet 9. We can observe the initialization of an `AtomicU128` value (line 2), after which any thread can atomically load or store a new value (lines 4 and 6 respectively). We therefore infer that the concurrency

pattern is nearly identical to the use of standard atomics in Rust. In this example, the semantic role of the atomic type is to store a hash value that can be updated atomically.

```
use portable_atomic::AtomicU128; // import portable-atomic crate
hash: AtomicU128; // initialization of 128-bit atomic integer
// Any thread can atomically get the value of the hash
hash.load();
// Any thread can atomically store a new hash value
hash.store(compute_hash128(value));
```

Code snippet 9: Non-standard atomic integer example

4.2 Results and answer for RQ 1

As the first result, we observe that several categories provide a channel-like communication pattern. For example, apart from the channels category, the concurrent queue and ring buffer categories provide `push` and `pop` operations, which allow one thread to push data into shared storage and another thread to later pop this data from it. Our interpretation is that channel-like communication is not limited to crates explicitly named "channels", but also appears in other crates with different API abstractions.

As the second result, we observe that some categories extend Rust's standard library in terms of concurrency functionality. For example, the atomic reference swapping category provides crates that extend the capabilities of Rust's standard `Arc` type. Another example is the non-standard atomics category, where some crates provide additional atomic types used by software developers in specific cases. Our interpretation is that some atomic-based crates act as extensions of Rust's standard concurrency toolbox rather than as completely new concurrency patterns.

As the third result, we observe that some categories provide functionality for managing shared storage or resources between threads. For example, the concurrent map category provides a concurrency pattern where different threads can retrieve values by keys. The work-stealing deque category provides a concurrency pattern where different threads can submit tasks that worker threads execute. Our interpretation is that atomic-based techniques can be used to manage shared resources that different threads access to perform their work.

We therefore answer the RQ 1 as follows: in open-source Rust crates, the following concurrency patterns can be observed:

- channel-like communication patterns, which allow threads to transfer data between each other;
- extension patterns, which extend Rust's standard atomics or the atomically reference-counted type `Arc`;
- concurrent data structures, which store data and internally coordinate thread-safe access;
- work-stealing schedulers, which allow threads to submit and execute tasks on demand.

4.3 Results and answer for RQ 2

As the first result, we observe that channel-like communication patterns are used to transfer different types of data between threads. Following the examples from our analysis section, they transfer either plain data, such as a worker result, or a specific object, such as a network connection object.

As the second result, we observe that categories such as atomic reference swapping and concurrent maps are used to provide an object or storage structure that can be accessed between threads. Following the examples from our analysis section, they store either a reference to a set of available resources or a mapping between a name and a worker object.

As the third result, we observe that the work-stealing deque category is used to perform application tasks on demand.

Our interpretation of these results is that atomic-based concurrency crates can be used in a wide range of applications in terms of semantical role, from concurrent data structures that store necessary data to more specific use cases, such as transferring data between threads through channels.

We therefore answer RQ 2 as follows: atomic-based techniques play the following semantic roles:

- transferring data or objects between threads;
- providing concurrent data structures shared between threads, such as maps, queues, and other storage structures;
- organizing work scheduling through specialized data structures, such as work-stealing deques;
- allowing shared references between threads to be replaced atomically.

4.4 RQ 3. Analysis of abstraction levels

In this subsection, we present our analysis of RQ 3 according to the methodology defined above. For each category and its producer crates, we examine their abstraction level by reviewing the documentation and, if necessary, the source code of the producer crate.

4.4.1 Channels

For this category, producer crates such as `crossfire`, `bus`, and `kovan-channel` describe their implementations as lock-free and based on atomic operations in their documentation [10, 11, 12]. The remaining crate, `crossbeam-channel`, does not provide explicit documentation details about its underlying implementation. However, in the crate's source code snippet 11, we observe `head` and `tail` variables (lines 3 and 4 respectively) which appear to store indices to underlying channel elements. These indices are updated using the `compare_exchange` operation (line 8) [13]. Therefore, we classify `crossbeam-channel` as a channel implementation whose coordination mechanism is at least partially based on atomic operations. We avoid claiming that the entire crate is purely lock-free, since this would require a more rigorous analysis of the code.

4.4.2 Atomic reference swapping

According to the documentation, crates in this category are implemented through low-level, pointer-based reference-counted memory management [14, 15, 16]. For example, `ArcSwap` uses the idea of so-called "hazard" pointers [17]. `ArcShift` stores values in heap-allocated blocks together with reference counters [15]. `aarc` uses an external crate that supports memory reclamation [16]. Thus, these crates use low-level pointer manipulation and reference-count management to expose the higher-level abstraction of atomic reference swapping.

4.4.3 Concurrent map

In this category, many producer crates have lock-free implementations based on atomic operations according to their documentation [18, 19, 20, 21]. These crates include `crossbeam-skiplist`, `papaya`, `concurrent-map`, and `concache`.

By contrast, crate `dashmap` uses a lock, that is, a synchronization primitive that allows only one thread to access shared data at a time, to coordinate internal multi-threaded access [22]. However, this lock is custom and based on atomic operations. In the crate's source code snippet 12, we can observe that the lock source code contains an atomic integer state (line 2), which is used by the lock to coordinate access between threads (lines 5 and 6) [23]. We therefore infer that the abstraction level of `dashmap` is relatively high, since its implementation is not primarily built directly on atomic operations.

The remaining crate, `evmap`, uses another lock-free crate to implement thread-safe access to its operations [24].

4.4.4 Work-stealing deque

In this category, we analyze the `crossbeam-deque` crate [25]. Based on its source code and comments, we infer that it is implemented as a lock-free work-stealing deque based on atomic operations [26]. In the crate's source code snippet 13 with the crate's source code, we can observe that the `Inner<T>` struct (line 2) contains two atomic indices (lines 3 and 4), which are used to coordinate the queue data structure containing available tasks. These indices are updated using the atomic `compare_exchange` operation (line 12) [26]. This suggests that, in terms of abstraction level, the crate can be classified as a lock-free implementation based on atomic operations.

4.4.5 Concurrent queue

In this category, the `bbqueue` crate directly states in its documentation that it is a lockless thread-safe queue [27].

For the `crossbeam-queue` crate, we observe two different implementations: a bounded concurrent queue and an unbounded concurrent queue [28]. For the bounded version, the source code explicitly states that the implementation is based on "Dmitry Vyukov's bounded MPMC queue" [29], which the author describes as "not lockfree in the official meaning", but implemented using atomic read-modify-write operations without mutexes [30]. We therefore infer that the bounded version is partially implemented using atomic operations.

The unbounded version does not explicitly describe its implementation details in the documentation. However, in the crate's source code snippet 14, we observe that the unbounded version, named `SegQueue`, has `head` and `tail` (lines 8 and 9 respectively) positions containing an atomic index and an atomic pointer [31]. We can also observe that the `tail` index can be updated (line 12) using the `compare_exchange` operation [31]. We therefore

infer that the unbounded version is also partially implemented using atomic operations. We avoid claiming unbounded version purely lock-free, since this would require a more rigorous analysis of the code.

4.4.6 Ring buffer

In this category, the documentation of all crates explicitly states that their implementations are lock-free and based on atomic operations [32, 33, 34]. Therefore, we do not inspect additional source-code details and classify each crate in this category as lock-free.

4.4.7 Non-standard atomics

In this category, we observed from the documentation that crate implementations significantly depend on the platform, such as the operating system, instruction architecture, and hardware support [35, 36, 37]. For example, the `portable-atomic` crate uses standard-library atomics when available, falls back to inline assembly when the platform supports atomic instructions, and otherwise uses global locks, that is, synchronization primitives that allow only one thread to access shared data at a time [35]. A similar pattern applies to the `atomic` crate [36]. The remaining crate, `atomic-wait`, uses different system calls depending on the platform to implement its functionality [37]. Therefore, we classify all crates in this category as mixed and highly platform-dependent, since they provide different implementations depending on the target platform.

4.5 Results and answer for RQ 3

As the first result, we observe that one abstraction level is low-level pointer management. For example, this abstraction level appears in the atomic reference swapping category. Following the analysis above, crates in this category use pointer-based reference-counted memory management to provide a higher-level abstractions. Our interpretation is that this category exposes a high-level API to the user, but its implementation is close to memory management details such as pointers, heap allocation, and reference counters.

As the second result, we observe that several categories are implemented as lock-free data structures based on atomic operations. This abstraction level appears in categories such as concurrent maps, work-stealing deques, concurrent queues, and ring buffers. Our interpretation is that these crates provide high-level data-structure APIs, while their implementation is still directly based on atomic coordination.

As the third result, we observe that some crates are only partially implemented using atomic operations. Following the analysis above, these crates either do not explicitly document the full implementation guarantees, use atomics together with other synchronization mechanisms, use another crate internally, or select different implementations depending on the target platform. Our interpretation is that atomic operations can appear as an internal implementation detail while the whole crate should not be classified as purely lock-free or directly atomic-based.

We therefore answer RQ 3 as follows: atomic-based techniques in the analyzed producer crates appear at three abstraction levels, shown in Table 2. The table reports one example crate for each abstraction level and the number of producer crates classified into that level.

Table 2: Abstraction levels of atomic-based techniques in analyzed producer crates.

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

5 Discussion

In this section, we discuss possible explanations for the results observed in the analysis. The goal of this discussion is not to prove additional claims, but to explain why certain atomic-based concurrency patterns and abstraction levels may appear in Rust crates.

One possible explanation for the popularity of channel-like communication patterns is their integration with Rust's ownership model. When a value is sent through a channel, ownership of this value can be transferred from the sending thread to the receiving thread. Therefore, threads do not necessarily need to share mutable access to the same object. We assume that this makes channel-like abstractions a natural way to structure communication between threads in Rust programs.

Another observation is that several crates expose safe public APIs while relying on low-level implementation mechanisms internally. For example, crates that implement atomic reference swapping, lock-free queues, work-stealing deques, or non-standard atomics may use atomic operations, raw pointers, memory reclamation, or platform-specific primitives inside their implementation. These mechanisms may require `unsafe` code. We assume that this reflects a common design approach in Rust concurrency libraries, where low-level details are hidden behind a safer higher-level abstraction.

Finally, we assume that many applications use atomic-based crates because threads often need to perform small and specific operations concurrently. Following the examples from our analysis, such operations include transferring ownership of an object, retrieving a value by key, replacing a shared reference, or submitting a task to workers. These operations are narrow in scope, but they still require correct coordination between threads.

6 Responsible Research

To support reproducibility, this project is accompanied by a reproducibility package [38]. The package contains the data used in the study, including the selected producer crates, their assigned categories, the corresponding consumer crates, and the scripts used to collect and process data from `crates.io`. It also explicitly lists all sampled consumer crates used in the manual analysis, so that the selection and classification process can be inspected and reproduced.

A limitation of this study is that the dataset was constructed from information available on `crates.io`. Therefore, the search depends on crate metadata such as names, descriptions, keywords, categories, README files, and repository links. Some relevant atomic-based crates may not explicitly describe themselves using terms such as *atomic* or *lock-free*, and therefore may be missed. Conversely, some crates may mention these terms even though atomic-based concurrency is not central to their functionality. As a result, the dataset should be interpreted as a systematic sample of discoverable atomic-based techniques on `crates.io`, rather than as a complete list of all such techniques in the Rust ecosystem.

Artificial intelligence tools were used only as writing and programming assistance. In particular, AI was used to help generate code for fetching metadata from `crates.io`, and to check the English grammar of the paper and suggest more idiomatic formulations of sentences written by the author. The final research design, classification decisions, analysis, and conclusions were made and verified by the author.

7 Related Work

In this section, we compare existing academic papers and their results with our results. We examine their similarities, their differences, and how they complement each other. Each paragraph reviews one paper and compares it with our findings.

Astrauskas et al. [39] analyzed how programmers use unsafe blocks in Rust and whether unsafe code is hidden behind safe abstractions. This is comparable to our abstraction-level results, where several crates expose high-level APIs while their implementations rely on atomic operations, pointer management, or platform-specific mechanisms. Their paper complements our results by providing additional insight into how our producer crates could be further divided into abstraction sublevels.

Abdi et al. [3] observed that, for patterns involving mutable access, programmers may need to use unnecessary synchronization or dynamic checks. This complements our observed result of extension concurrency pattern, where Rust's standard guarantees are not always expressive enough, so libraries extend Rust with higher-level abstractions over unsafe code, atomics, synchronization, or dynamic checks. Their results therefore provide indirect support for our findings.

Qin et al. [1] studied memory and thread-safety practices and issues in real-world Rust programs. Their work provides insights into bugs related to channels and reports that a lack of understanding of Rust's lifetime rules is a common cause of blocking bugs. This result complements our finding that channel-like communication is one of the observed concurrency patterns. Channels are not only a common communication mechanism, but can also become a source of concurrency bugs when programmers misunderstand Rust's ownership, lifetime, or blocking behavior.

To summarize, existing work provides results that complement our findings. Together, these papers show that Rust concurrency is shaped not only by language-level guarantees, but also by abstractions, libraries, and programmer understanding.

8 Future Work & Conclusion

Several directions can be explored in future work. First, future research could study how lock-free data structures are implemented in Rust in more detail. This includes examining which operations require `unsafe` code blocks and how Rust's type system influences implementation choices. Second, more thorough research could be conducted on the workload motivations behind the use of atomic-based techniques in crates. It would be useful to identify benchmarks and workloads where different techniques perform better and explain why this happens. Third, atomic-based techniques in C++ and Rust could be compared. Investigating how similar concurrency patterns are used in C++ and Rust may provide useful insights.

To conclude, by analyzing code from different Rust crates, we identified a variety of concurrency patterns, semantic roles, and abstraction levels of atomic-based techniques in

Rust. We also discussed why certain patterns appear, how atomic-based techniques coexist with Rust's ownership model, and why they may be selected for use in the code. These results help explain how lock-free and atomic-based concurrency techniques are used in Rust crates.

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. DOI: 10.1145/3385412.3386036.

[2] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. "Miri: Practical Undefined Behavior Detection for Rust". In: *Proceedings of the ACM on Programming Languages* 10.POPL (2026), pp. 1383–1411. DOI: 10.1145/3776690.

[3] Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C. Jeffrey. "When Is Parallelism Fearless and Zero-Cost with Rust?" In: *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '24. Association for Computing Machinery, 2024, pp. 27–40. DOI: 10.1145/3626183.3659966.

[4] The Rust Project Developers. *Understanding Ownership. The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

[5] The Rust Project Developers. *std::sync::Arc documentation*. URL: https://doc.rust-lang.org/std/sync/struct.Arc.html.

[6] The Rust Project Developers. *std::sync::atomic documentation*. URL: https://doc.rust-lang.org/std/sync/atomic/.

[7] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '96. Association for Computing Machinery, 1996, pp. 267–275. DOI: 10.1145/248052.248106.

[8] Tokio Project Developers. *Tokio crate documentation*. URL: https://docs.rs/tokio/latest/tokio/.

[9] Rust Foundation. *crates.io*. URL: https://crates.io/.

[10] *crossfire: Rust crate documentation*. docs.rs. URL: https://docs.rs/crossfire/latest/crossfire/.

[11] *bus: Rust crate documentation*. docs.rs. URL: https://docs.rs/bus/latest/bus/.

[12] *kovan-channel: Rust crate documentation*. docs.rs. URL: https://docs.rs/kovan-channel/latest/kovan_channel/.

[13] *crossbeam-channel: list channel source code*. GitHub. URL: https://github.com/crossbeam-rs/crossbeam-channel/blob/master/src/flavors/list.rs.

[14] *arc-swap: Rust crate documentation*. docs.rs. URL: https://docs.rs/arc-swap/latest/arc_swap/.

[15] *ArcShift: repository README*. GitHub. URL: https://github.com/avl/arcshift.

[16] *aarc: Rust crate documentation*. docs.rs. URL: https://docs.rs/aarc/latest/aarc/.

[17] *arc-swap: internal implementation notes*. GitHub. URL: https://github.com/vorner/arc-swap/blob/master/src/docs/internal.rs.

[18] *crossbeam-skiplist: Rust crate documentation*. docs.rs. URL: https://docs.rs/crossbeam-skiplist/latest/crossbeam_skiplist/.

[19] *papaya: Rust crate documentation*. docs.rs. URL: https://docs.rs/papaya/latest/papaya/.

[20] *concurrent-map: Rust crate documentation*. docs.rs. URL: https://docs.rs/concurrent-map/latest/concurrent_map/.

[21] *concache: Rust crate documentation*. docs.rs. URL: https://docs.rs/concache/latest/concache/.

[22] *DashMap: Rust crate documentation*. docs.rs. URL: https://docs.rs/dashmap/latest/dashmap/struct.DashMap.html.

[23] *DashMap: lock implementation source code*. Source mirror. URL: https://wide.gitlabpages.inria.fr/data-wallet-prototype/src/dashmap/lock.rs.html.

[24] *evmap: repository README*. GitHub. URL: https://github.com/jonhoo/evmap.

[25] *crossbeam-deque: Rust crate documentation*. docs.rs. URL: https://docs.rs/crossbeam-deque/latest/crossbeam_deque/.

[26] *crossbeam-deque: deque source code*. GitHub. URL: https://github.com/crossbeam-rs/crossbeam/blob/master/crossbeam-deque/src/deque.rs.

[27] *bbqueue: Rust crate documentation*. docs.rs. URL: https://docs.rs/bbqueue/latest/bbqueue/.

[28] *crossbeam-queue: Rust crate documentation*. docs.rs. URL: https://docs.rs/crossbeam-queue/latest/crossbeam_queue/.

[29] *crossbeam-queue: bounded queue source code*. GitHub. URL: https://github.com/crossbeam-rs/crossbeam/blob/master/crossbeam-queue/src/array_queue.rs.

[30] Dmitry Vyukov. *Bounded MPMC queue*. URL: https://sites.google.com/site/1024cores/home/lock-free-algorithms/queues/bounded-mpmc-queue.

[31] *crossbeam-queue: unbounded queue source code*. GitHub. URL: https://github.com/crossbeam-rs/crossbeam/blob/master/crossbeam-queue/src/seg_queue.rs.

[32] *ringbuf: Rust crate documentation*. docs.rs. URL: https://docs.rs/ringbuf/latest/ringbuf/.

[33] *rtrb: Rust crate documentation*. docs.rs. URL: https://docs.rs/rtrb/latest/rtrb/.

[34] *thingbuf: Rust crate documentation*. docs.rs. URL: https://docs.rs/thingbuf/latest/thingbuf/.

[35] *portable-atomic: Rust crate documentation*. docs.rs. URL: https://docs.rs/portable-atomic/latest/portable_atomic/.

[36] *atomic: Rust crate documentation*. docs.rs. URL: https://docs.rs/atomic/latest/atomic/.

[37] *atomic-wait: Rust crate documentation*. docs.rs. URL: https://docs.rs/atomic-wait/latest/atomic_wait/.

[38] David Potskhishvili. *Research Project Reproducibility Package*. URL: https://github.com/DavidPotskhi/Research-Project-Reproducibility-Package.

[39] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. "How Do Programmers Use Unsafe Rust?" In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–27. DOI: 10.1145/3428204.

Appendix

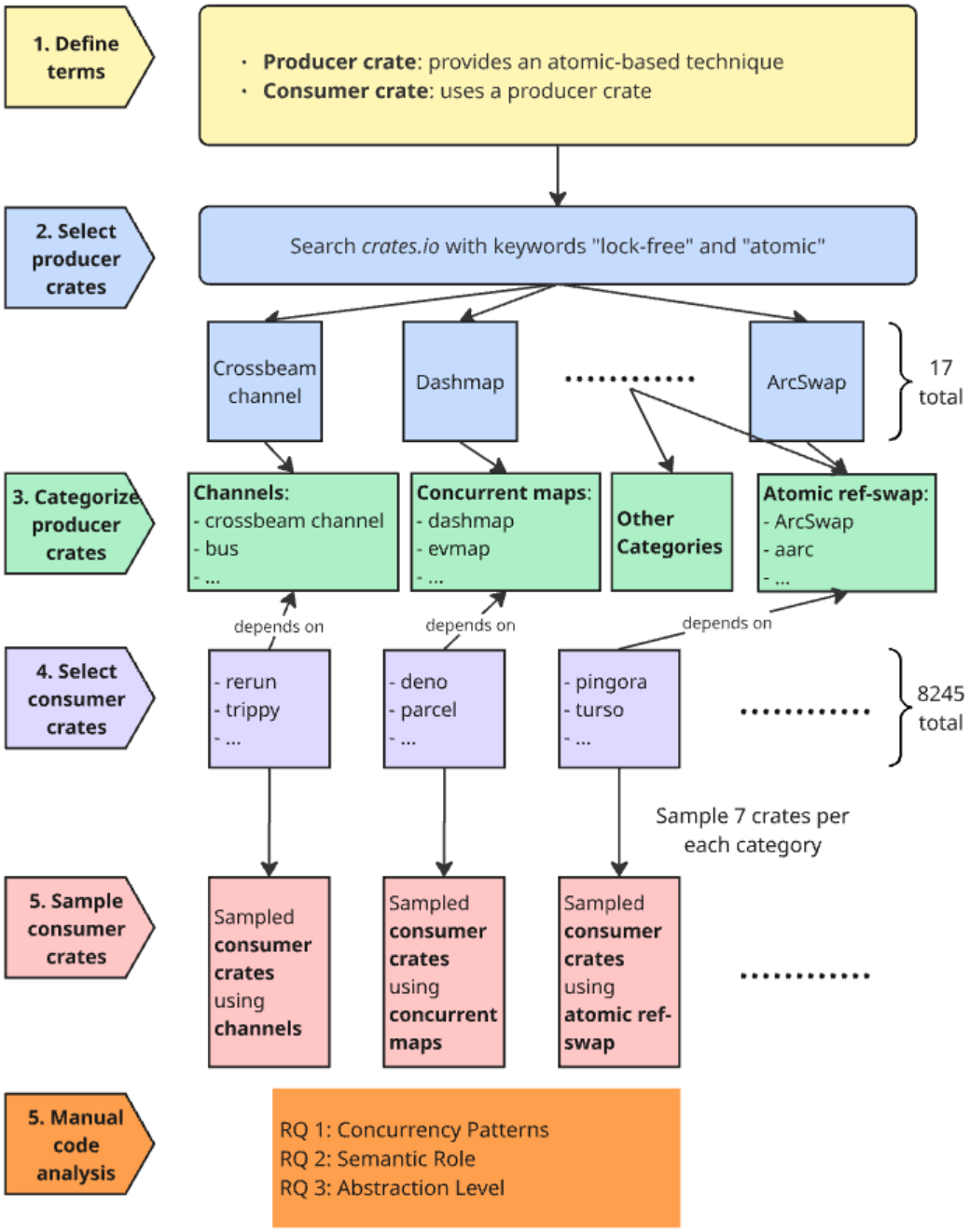


Figure 1: Methodology diagram

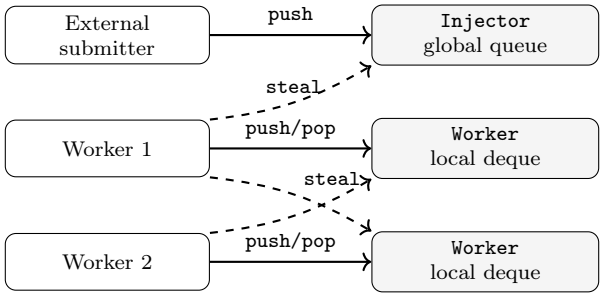
```rust
// Each worker thread has a local queue of tasks and references to stealers
local: Worker<Task>;
stealers: Vec<Stealer<Task>>;
// We have a global queue where non-worker threads can push their tasks
global: Injector<Task>;

// Work submitted from outside the worker pool enters the global queue
global.push(task);
// Work created by a running worker is pushed to its own local deque
local.push(task);

//Process of finding a new task for a worker
// 1. Prefer local work
local.pop()
// 2. If local work is empty, try to steal from other workers
for stealer in &stealers { match stealer.steal() { ... } }
// 3. Finally, take work submitted from outside the pool of workers
scheduler.global.steal() { ... }
```

Code snippet 10: Work-stealing deque example

Figure 2: Simplified work-stealing scheduler with two workers. Each worker first uses its own local deque, can steal from another worker's deque, and can also take externally submitted work from the shared `Injector`.




```rust
// Channel stores atomic indices to elements in the channel
pub(crate) struct Channel<T> {
    head: CachePadded<AtomicUsize>,
    tail: CachePadded<AtomicUsize>,
    ...
}
// One possible way to update an index in the channel
tail.compare_exchange_weak(tail, new_tail, ...);
```

Code snippet 11: Simplified `crossbeam-channel` source code.

```
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, ...) {
        ...
    }
}
```

Code snippet 12: Simplified `DashMap` lock source code.

```
// Internal queue data shared between the worker and stealers.
struct Inner<T> {
    front: AtomicIsize, // The front index.
    back: AtomicIsize,  // The back index.
    ...
}
// Pops a task from the queue.
pub fn pop(&self) -> Option<T> {
    let f = self.inner.front.load(Ordering::Relaxed); // Load the front index.
    ...
    // Try incrementing the front index.
    if inner.front.compare_exchange(f, f.wrapping_add(1), ...) {
        ...
    }
}
```

Code snippet 13: Simplified `crossbeam-deque` source code.

```
// A position in a queue
struct Position<T> {
    index: AtomicUsize,
    block: AtomicPtr<Block<T>>,
}
// An unbounded multi-producer multi-consumer queue
pub struct SegQueue<T> {
    head: CachePadded<Position<T>>,
    tail: CachePadded<Position<T>>,
}
// Try advancing the tail forward
match self.tail.index.compare_exchange_weak(tail, new_tail, ...) {
    ...
}
```

Code snippet 14: Simplified `crossbeam-queue` unbounded queue source code.