# Scalable FIFO Channels for Programming via Communicating Sequential Processes

Nikita Koval
IST Austria & JetBrains

Dan Alistarh
IST Austria

Roman Elizarov
JetBrains

**Abstract**

Traditional concurrent programming involves manipulating shared mutable state. Alternatives to this programming style are communicating sequential processes (CSP) and actor models, which share data via explicit communication. These models have been known for almost half a century, and have recently had started to gain significant traction among modern programming languages. The common abstraction for communication between several processes is the *channel*. Although channels are similar to producer-consumer data structures, they have different semantics and support additional operations, such as the `select` expression. Despite their growing popularity, most known implementations of channels use lock-based data structures and can be rather inefficient.

In this paper, we present the first efficient lock-free algorithm for implementing a communication channel for CSP programming. We provide implementations and experimental results in the Kotlin and Go programming languages. Our new algorithm outperforms existing implementations on many workloads, while providing non-blocking progress guarantee. Our design can serve as an example of how to construct general communication data structures for CSP and actor models.

## 1 Introduction

Programming via communicating sequential processes (CSP) was introduced by Hoare [18] almost half a century ago, and has had significant research and practical impact [10]. In particular, many modern programming languages, such as Go [8], Kotlin [4], Scala [25], and Rust [9] provide support for this programming paradigm, as an alternative, or complement, to synchronization via shared memory.

Very roughly, programs in CSP can be seen as a parallel composition of sequential processes, communicating with each other via synchronous message-passing.[1] The CSP paradigm is built around *channels*, which provide the basic communication and synchronization mechanisms between the computational processes.

To efficiently support CSP programming on modern multi-threaded multiprocessors, it is critical to be able to implement *fast, scalable* channels, supporting the CSP semantics. Naively, concurrent channels can be seen as a classic instance of first-in-first-out (FIFO) queues, which have been extensively studied in shared-memory programming [17]. Yet, CSP channels typically require additional, non-trivial semantics, which are not easy to pigeonhole in the classic producer-consumer data structure definitions, and their usage scenarios can vary broadly. One such example is commonly-used

[1] CSP is broadly similar to the actor model [12], with key distinctions in terms of the basic assumptions regarding process identities and synchronization.

the `select` expression, by which a process can register operations in a set of channels, returning on the first operation that succeeds.

Different programming language implementations fill this implementation gap by "brewing" their own FIFO channel implementations, either independently, or adapting ideas present in the concurrency literature. Most implementations, including Go [8] and Rust [9], rely on *lock-based* designs. In particular, they rely on careful combinations of fine-grained and coarse-grained synchronization to implement the complex channel semantics. One notable exception is the Kotlin coroutines library [4], which implements a complex producer-consumer data structure, based on a doubly-linked list design [31]. Unfortunately, this implementation is known to allow for live-lock in certain corner cases [5]. Another one exception is `SynchronousQueue` in Java [27], which is based on Michael-Scott queue [23]; however, it does not support the `select` expression.

**Contribution.** In this work, we revisit the question of implementing an efficient channel with CSP semantics. We provide the first linearizable, lock-free implementation of a rendezvous channel supporting `send`, `receive`, and `select` operations, that is fast in both contended and uncontended scenarios, and can be extended to implement CSP semantics as specified by various programming languages.

Our design builds on ideas from previous work on lock-free FIFO queues, e.g. [23, 11, 27]. Our base data structure is similar to the Michael-Scott queue, but where nodes are *segments*, which can accommodate multiple waiting operations of either `send` or `receive` type.

We non-trivially extend this blueprint to support extended semantics in a wide range of contended and uncontended scenarios. Specifically, we add efficient support for `select` operations, which can wait on operation in a *set* of channels, via a customized implementation of descriptors [14]. A novel feature of our mechanism is that it permits physical removal of elements from the middle of a queue-like structure in $O(1)$ amortized operations, which is required for an implementation of the `select` expression. The proposed algorithm is quite general, in that it can be adapted to other synchronization primitives for CSP programming, such as buffered channels, mutexes, and semaphores.

We validate our algorithm with efficient implementations of our channel in Go and Kotlin. Our algorithm can provide comparable performance relative to the lock-based Go implementation in a range of scenarios, and outperforms it by up to 2× in terms of average time per operation. Our Kotlin implementation significantly outperforms the existing library implementation [4], especially the `select` expression. It speeds up the `send` and `receive` operations by up to 2× for the most part, and by a couple of magnitude in certain scenarios; the `select` expression is faster by up to 10×.

## 2 Channel Semantics

The rendezvous channel is the main abstraction for message passing protocols used in both CSP and actor programming models. Intuitively, in this abstraction, there are two types of processes, producers and consumers, which perform a rendezvous handshake as a part of their protocol. This section describes the channel semantics and its API, which is shown in Listing 0. In the following, we assume that threads are either *producers*, which perform `send` requests, or *consumers*, which perform `receive` requests.

**Send and receive semantics.** In order for a producer to "send" an element, it has to perform a rendezvous handshake with a consumer, to which it passes this element. The consumer semantics

```
class Channel {
  fun send(element: Any)
  fun receive(): Any
}

fun select(alternatives: SelectAlt[])
```

```
class SelectAlt(
  val channel: Channel,
  val element: Any?, // null for receive
  val action:  fun(Any?)
)
```

Listing 0: Rendezvous channel API

are symmetric. At the same time, we assume *first-in-first-out (FIFO)* guarantees, meaning that all requests in a channel should be processed in the order of arrival.

At the operational level, the rendezvous channel is essentially a queue of waiting processes, where each request atomically checks if the queue has processes of the opposite type, and either removes the first one and resumes it, or adds itself to the queue and suspends. Practical systems offer an efficient way to `suspend` a process waiting for a request, and `resume` it after that. This mechanism is described in Section 4.

**The `select` expression.** Channels in the CSP programming model usually support selection among several alternates. This expression is usually called `select` and it makes possible to await multiple `send` or `receive` invocations on different channels, and select the first one which becomes available. At the same time, the chosen process should be removed from the other waiting queues. The classic `select` expression checks the alternatives in the enumeration order, while the unbiased version uses a random order [8, 4]. For simplicity, we assume an arbitrary order on alternatives, but the proposed algorithm does not rely on this restriction and can be easily modified to support any ordering strategy.

Programming languages take different approaches for defining the `select` expression. For example, Go supports it as a built-in feature [8], while Kotlin implements it in the Kotlin Coroutines library providing a domain specific language (DSL) for the `select` declaration [4]. For this paper we use a DSL-based API, which can be used for built-in expression implementation as well.

To describe alternatives, we use the `SelectAlt` class, which specifies a channel, an element to be sent (`null` is used for receiving), and an action to be executed with the received element (`null` is passed when sending) in case this alternative is selected. The selection algorithm can be implemented as an external `select` function taking an array of alternatives. Listing 1 shows an example usage of this API.

```
ch1 := Channel(), ch2 := Channel()
select(
  SelectAlt(ch1, 42,   { _ -> println("Sent 42") }),
  SelectAlt(ch2, null, { res -> println("Received $res") })
)
```

Listing 1: Example of the `select` expression usage. This code either sends "42" to the first channel or receives an element from the second one.

# 3 Related Work

## 3.1 Coroutines and Actors

Two basic parallel programming models use message passing for synchronization: actors [12] and coroutines [20]. The second one is also known as green threads and fibers. Roughly, an actor can

be represented as a coroutine associated with a channel, to which other actors can send messages, and from which this actor can receive them. In contrast to actors, coroutines use channels directly and can perform the `select` expression on them.

Several modern programming languages and libraries use one of these models. For example, Go [8], Kotlin [4], Clojure [7], Rust [9], and project Loom for Java [6] use coroutines, while Erlang [3] and Akka [2] use the actor model; all of them have their own channel implementations. Almost all solutions we are aware of use *locks* in order to support a waiting queue, and perform the `select` expression using fine-grained synchronization on these locks. These approaches are fundamentally blocking.

To our knowledge, the only *non-blocking* channel implementation is a part of the Kotlin coroutines library [4], which is *lock-free*. Here, the waiting queue is implemented using a modified instance of the doubly-linked list designed by Sundell and Tsigas [31], and Harris descriptors [14] to ensure the atomicity. The resulting implementation is extremely complex, and shows significant overhead, as we show in the experimental section. Moreover, it is known that Kotlin's lock-free algorithm for a correctness bug, and can get into a livelock [5]. This makes our solution for the `select` expression the first lock-free implementation to support such semantics.

### 3.2 Producer-Consumer Data Structures

**Fair synchronous queues.** Among classic data structures, *synchronous queues* are probably closest to channels in terms of semantics. They support `send` and `receive` operations, which wait for a rendezvous. The main difference is that they do not support the `select` expression. Hanson suggested an algorithm based on three semaphores [13], which was improved in Java 5 using a global lock and `wait-notify` mechanism in order to make a rendezvous.

Java 6 implements a lock-free algorithm suggested by Scherer, Lea, and Scott [27]. Their solution is based on Michael-Scott lock-free queue [23]; we use a similar technique for maintaining the waiting queue.

Reference [19] presents an improved general scheme for implementing non-blocking dual containers. Their approach leverages the LCRQ non-blocking queue design [24] in a clever way, to implement a rendezvous mechanism. While this approach is quite interesting and efficient in practice, it requires access to double-width atomic primitives, which are only supported by Intel CPUs. This makes the code non-portable: for instance, it would prevent implementation in both Java/Kotlin and Go. Therefore, we cannot apply it in our setting.

**Unfair synchronous queues.** Scherer, Lea, and Scott also propose *unfair* but scalable synchronous queue, which is based on a stack instead of a queue [27]. However, this approach still induces a sequential bottleneck on the stack. To work around a single point of synchronization, Afek, Korland, Natanzon, and Shavit introduced *elimination-diffraction trees* [11]. In this solution, each request goes through a binary tree, in which internal nodes are *balancer objects* [30], and leaves are synchronous queues.

Another approach to reducing contention is *flat combining* [15], which was applied to unfair synchronous queue problem by Hendler et al.[16].

**Elimination.** One more way to reduce contention is using elimination, which was firstly applied for stacks [29] by Shavit and Touitou. They observed that concurrent `push` and `pop` operations can be *eliminated* preserving atomicity by having the `push` operation pass directly to `pop`, without modifying the stack. Scherer, Lea, and Scott use elimination in their exchanger algorithm [26],

```
class Coroutine {
  ...
  var result: Any?
}

fun curCor(): Coroutine

fun park()
fun unpark(c: Coroutine)

fun resume(c: Coroutine, res: Any?) {
  c.result = res
  unpark(c)
}

fun suspendAndGetResult(): Any? {
  c := curCor()
  park(c)
  return c.result
}
```

Listing 1: Coroutines management primitives

where there is only one type of request and so a rendezvous happens between any two threads that show up. In case of channels, not all pairings are allowed, so their approach is not applicable for our problem.

# 4 Preliminaries

**Coroutines management.** To implement `send` and `receive` operations, we need to have an ability to store the current coroutine somewhere and suspend it, after what an opposite operation can resume it. It is worth noting that some libraries use another approach and store continuation, which has the required information for resuming. Nevertheless, both approaches are equivalent for our purpose and should provide the described above functionality.

The algorithm presented in the paper uses definitions from Listing 4. The `Coroutine` class represents a coroutine and new fields can be inserted into it for synchronization. To get the current coroutine, we use the `curCor` function. The `park` and `unpark` methods are used for suspending and resuming the coroutine respectively. As well as in the native thread park mechanism, we assume that `park` returns without suspension if `unpark` has been called before. However, in case of suspension, `park` does not block the current thread, but schedules to another waiting coroutine.

Not all environments provide such contracts. For example, in Go language `gopark` (park) invocation should always *happen before* `goready` (unpark) invocation. This solution is sufficient for the lock-based algorithm, but can be also easily extended to support the required contract by providing an appropriate implementation of `park` on top of `gopark`.

In order to send an element to a suspended coroutine, we suggest adding a special field into `Coroutine` class, which stores the result. This way, in case of resuming a coroutine, we use `resume` method which stores the result in this field and unparks the coroutine. From the other side, to suspend and return this result we use `suspendAndGetResult`, which parks the current coroutine and returns the previously stored result right after it was unparked.

**Lock-freedom.** When discussing progress guarantees, we assume that the provided `park`, `unpark`, and `curCor` functions are lock-free. In case of `park`, this means that the current thread parks the current coroutine and schedules another one in a lock-free way.

Taking into account that the channel is a blocking data structure by design (sender waits for a receiver and vice versa), it is possible to guarantee lock-freedom for a part of the algorithm only. Similar to the dual data structures formalism [28], we split every operation into two parts. At first, it atomically checks for an opposite coroutine in the waiting queue and either removes it or adds the current one. Secondly, in case of adding to the waiting queue (or adding to all waiting queues

in `select` statement), it parks the current coroutine and waits for an `unpark` invocation. Thus, we guarantee lock-freedom for the first part only, which essentially does all the synchronization.

**Memory model and atomic primitives.** For simplicity of exposition, we assume a sequentially-consistent memory model, although our implementations work under the practical, weaker models. The presented algorithm requires only *compare-and-set* (CAS) primitive in addition to the standard read and write. It is denoted as **`CAS`**`(&p, old, new)`, and atomically checks that the value located by address `p` equals to `old` and changes it to `new`. It returns `false` if the check fails and `true` otherwise.

**Memory reclamation.** We assume that run-time environment supports garbage collection, which is true for Kotlin and Go. However, this assumption is mostly for simplicity of exposition. Reclamation techniques like hazard pointers [22] can be used in other environments.

# 5 Algorithm Description

**Overview.** Similarly to the rendezvous channel specification, our algorithm maintains a waiting queue, which is loosely based on the Michael-Scott [23] queue design. However, our waiting queue stores several waiters in each node, and supports a more complex channel contracts, including the `select` expression. This section describes the proposed algorithm iteratively. At first, the basic algorithm for `send` and `receive` operations is discussed; after that, we extend the algorithm in order to support the `select` expression.

## 5.1 Channel Structure Overview

Essentially, our algorithm implements a fair synchronous queue, which stores several waiting processes in each node, and supports the `select` expression. The data structure corresponding to the proposed algorithm is shown in Listing 1.

The overall structure of the channel is represented as a Michael-Scott lock-free queue [23], using head and tail pointers to `Node` instances. However, instead of dynamically creating a new `Node` for each operation to be waited on, our `Node` has a fixed-size `waiters` array of `NODE_SIZE` structures of `Waiter` type. The `Waiter` structure represents either a waiting receiver (if its `el` field is a special marker element `RECEIVE_EL`) or a waiting sender (when `el` field is the element that is being sent which is neither the marker nor `null`), together with the reference to the corresponding coroutine in the `cor` field. Initially, all items in the `waiters` array are filled with `null` values (both `el` and `cor` fields). Every instance of `Node` has a unique integer `id` that is equal to zero for the first node. When a new node is added to the `next` pointer, the invariant is maintained that `node.next.id` = `node.id` + 1.

In contrast to the synchronous queue algorithm by Scherer et al. [27], our channel has global `enqIdx` and `deqIdx` which indicate the current position to enqueue a new waiter and to dequeue the oldest one correspondingly. These positions are monotonically increasing 64-bit integer counters, their value modulo `SEGMENT_SIZE` indicate an offset in `waiters` array, while the remainder modulo `SEGMENT_SIZE` corresponds to the `id` of the corresponding `Node`.

While updating these indices, we maintain the invariant that `deqIdx` $\leq$ `enqIdx`, and these indices are equal when the channel is empty. An additional invariant is that the first waiter slot in `waiters` array of a node is always occupied when this node is added to the queue. To maintain this invariant, the initial values of `deqIdx` and `enqIdx` start with one. In practical applications,

```
class Node {
  val id: Long // initialized on creation
  val waiters: Waiter[SEGMENT_SIZE]
  var next: Node? = null
}
struct Waiter {
  var cor: Coroutine? = null
  var el: Any? = null // element
}
```

```
class Channel {
  var enqIdx: Long = 1 // enqueue index
  var deqIdx: Long = 1 // dequeue index
  var head, tail: Node

  Channel() {
    head = tail = Node{ id: 0 }
  }
}
```

Listing 1: Data structures for the channel

64-bit counters are big enough as to never overflow. In effect, with a linked list of `Node` structures we are modelling an array-based queue of unbounded size, where each node is a *segment* of this array.

Similarly to the LCRQ algorithm [24], and unlike the synchronous queue [27] and Michael-Scott [23] algorithms, both `send` and `receive` operations are linearized on the writes of the `el` field to the corresponding `Waiter` slot in the node. Lock-freedom is guaranteed in a similar way to LCRQ.

### 5.2 The `send` and `receive` Operations

Send and receive operations follow almost identical algorithm steps. They both look for a potential rendezvous with a waiter of the opposite type (`send` rendezvous-es with `receive`, and vice versa) or add themselves as a new waiter. This complex operation has to be performed atomically with respect to other operations, maintaining the invariant that the queue contains waiters of one type only (either senders or receivers), or is empty. The only difference between send and receive is that on rendezvous `send` transfers its element to the waiting `receive`, while `receive` does the opposite. So, in the following explanation we consider algorithm for the `send` operation only.

**High-level overview.** The pseudo-code for `send` operation is presented in Listing 5.2. Without interference from other threads, the `send` algorithm proceeds as follows. First, it reads both `enqIdx` and `deqIdx` (in this order) and checks if the queue is empty, adding itself to the queue in this case. If the queue contains waiters, it reads the first element and checks if it has the opposite type (if it is a `send` operation — the opposite type is `receive`). If a rendezvous is possible, it removes the first waiter from the queue and makes the rendezvous, resuming the corresponding coroutine with the specified element, terminating after that. Otherwise, the queue contains waiters of the same `send` type and the current coroutine is added to the queue as a new waiter. The whole `send` operation is enclosed in an infinite loop to retry when interference from other threads is detected.

**Adding to the waiting queue.** The algorithm to add to the waiting queue is presented in function `addToQueue`. The `enqIdx` for this operation has been already read in the beginning of the algorithm, and references the slot to which the waiter information is going to be written to. At first, the algorithm reads the `tail` pointer of the waiting queue, which references the last `Node` in it. There are two cases here: if a slot is in this last node then `storeWaiter` function is used, otherwise a new node is created using the `addNewNode` function.

When storing the waiter information to the last node, we increment the global `enqIdx` first, then write the current coroutine, then write the element. The storing of the element is the linearization point of the operation.

```
fun send(el: Any) = while (true) {
  enqIdx := this.enqIdx;
  deqIdx := this.deqIdx
  // Are enqIdx and deqidx consistent?
  if (enqIdx < deqIdx) continue
  // Is the queue empty?
  if (deqIdx == enqIdx) {
    if (addToQueue(enqIdx, element)) {
      park()
      return
    } else continue
  }
  head := this.head // read head
  // Is the state consistent?
  if (deqIdx / SEGMENT_SIZE < head.id)
    // deqIdx is inconsistent
    // with head
    continue
  if (deqIdx / SEGMENT_SIZE > head.id) {
    // head is outdated,
    // move it forward
    CAS(&this.head, head, head.next)
    continue
  }
  // Read the first element
  idxInNode := deqIdx % SEGMENT_SIZE
  firstEl := readEl(head, idxInNode)
  if (firstEl == BROKEN) {
    // The slot is broken, skip it
    CAS(&this.deqIdx, deqIdx, deqIdx + 1)
    continue
  }
  if (firstEl == RECEIVE_EL) {
    // Try to make a rendezvous
    if (resumeWaiter(head, deqIdx, elem))
      return
  } else {
    // Try to add to the queue
    if (addToQueue(enqIdx, element)) {
      park()
      return
    }
  }
}

fun resumeWaiter(head: Node, i: Int,
                 el: Any): Bool {
  if (!CAS(&this.deqIdx, i, i + 1))
    return false
  w := head.waiters[i] // read slot
  head.waiters[i] = null // clear
  resume(w.cor, el)
  return true
}

fun addToQueue(i: Int, el: Any): Bool {
  tail := this.tail // read tail
  // Is the state consistent?
  if (tail.id > i / SEGMENT_SIZE)
    return false
  if (tail.id == i / SEGMENT_SIZE &&
    i % SEGMENT_SIZE == 0)
    CAS(&this.enqIdx, i, i + 1)
  // Either store to the tail
  // or create a new node
  if (i % SEGMENT_SIZE != 0)
    return storeWaiter(tail, enqIdx, el)
  else
    return addNewNode(tail, enqIdx, el)
}

fun addNewNode(tail: Node, i: Int,
               el: Any): Bool {
  while (true) {
    tailNext := tail.next
    if (tailNext != null) {
      // Help another thread
      CAS(&this.tail, tail, tailNext)
      CAS(&this.enqIdx, i, i + 1)
      return false
    }
    newTail := Node {id: tail.id + 1}
    newTail.waiters[0].el = el
    newTail.waiters[0].cor = curCor()
    if (CAS(&tail.next, null, newTail) {
      // Others can help us
      CAS(&this.tail, tail, newTail)
      CAS(&this.enqIdx, i, i + 1)
      return true
    } else continue
  }
}

fun storeWaiter(tail: Node, i: Int,
                el: Any): Bool {
  if (CAS(&this.enqIdx, i, i + 1))
    return false
  tail.waiters[enqIdx].cor = curCor()
  if (CAS(&tail.waiters[i].el, null, el))
    return true
  tail.waiters[i].cor = null
  return false
}

fun readEl(n: Node, i: Int): Any? {
  el := n.waiters[i].el
  if (el != null) return el
  if (CAS(&n.waiters[i].el, null, BROKEN)
    return
  else return n.waiters[i].el
}
```

Listing 1: Algorithm for `send`, without the `select` expression support

When a new node is created, our algorithm uses the same logic as the Michael-Scott queue: it creates a new node with the current coroutine and element as the first waiter, changes the `next` field of the current tail, and updates the `tail` field. The linearization point here is update of the `next` field, as in Michael-Scott queue. Subsequent updates of the `tail` only maintain queue consistency, and other concurrent operations can help updating it. In our algorithm, we also maintain the global `enqIdx` similarly to how the tail pointer is maintained in the Michael-Scott queue. We update the `enqIdx` after the `tail` is updated. In case of concurrent execution, other operations can help with this update as well.

**Rendezvous.** The algorithm for rendezvous is presented as function `resumeWaiter`. In this case, the first element is already read, and we only need to increment the `deqIdx` to remove it from the queue. The successful CAS of `deqIdx` linearizes this operation.

**Reading an element.** When a new waiter is added to the queue, `enqIdx` is incremented, which signals that the queue is non-empty. After that, another thread can try to remove a waiter from this slot in `waiters` array, while the element was not written there yet. We cannot wait for the writer to write an element in a lock-free algorithm, so if the thread reads the `null` element, it does a CAS from `null` to a special `BROKEN` marker to "poison" this slot (see `readEl` function). On the other hand, the writer attempts to CAS from `null` to the element, and aborts the operation on encountering a broken slot. This solution is similar to LCRQ [24]. Two threads can repeatedly interfere with each other, which would render this algorithm obstruction-free if the `waiters` array were unbounded. However, the `waiters` array is bounded, and will be ultimately be filled with broken slots, triggering the creation of a new node with the already stored waiter, which proceeds as in the Michael-Scott algorithm and guarantees lock-freedom.

**Concurrency.** In order to preserve consistency of the data structure, all modifications before linearization points are performed using CAS, restarting on failures. Modifications after linearization points also use CAS, but do not retry on failures, as other threads can detect such inconsistencies and help fix them.

To check that the state is consistent, after reading the `enqIdx` and `deqIdx` fields, we check that $\texttt{enqIdx} \geq \texttt{deqIdx}$. If this condition does not hold, it means that indices were updated by concurrent operations in between reading of `enqIdx` and `deqIdx`. In this case, we retry the operation immediately, to re-read a consistent pair of indices. However, a consistent pair of indices can still point to wrong slots by the time we come to reading or writing them, due to concurrent operations.

We first consider the case when the `send` operation decides to add itself as a waiter to the queue, invoking `addToQueue`. This invocation succeeds and returns `true` only if `enqIdx` is not updated concurrently. Because `enqIdx` read is the very first action in the `send` algorithm, concurrent operations could only have removed elements from the waiting queue, incrementing `deqIdx`. However, removing elements from the queue does not invalidate the decision to add a new waiter to the queue: a new waiter is added when the queue either contains waiters of the same type, or is empty.

In the case when `send` decides to make a rendezvous and invokes `resumeWaiter`, the first action of it is to increment `deqIdx` using CAS, to ensure it is not updated by concurrent operations. The first `resumeWaiter` to successfully perform this CAS claims this slot. This successful CAS is a linearization point for `send` operation in this case.

### 5.3 The `select` Expression

A high-level algorithm for the `select` expression is presented in Listing 2 and proceeds in several phases. Each `select` instance is internally represented by `SelectOp` class, which contains the current coroutine (`cor` field) and the current state of this `select` instance (`state` field). In the first *registration* phase the `select` instance is added to all the corresponding channels as a waiter, similarly to the plain `send` and `receive` operations. During this phase, it can make a rendezvous with another waiter, become selected, and jump to the *removing* phase. If the registration phase completes without rendezvous, then this `select` is in the waiting phase until another coroutine makes a rendezvous with it by performing an opposite operation. After that the *removing* phase starts, during which all the registered waiters for this `select` instance are removed from the corresponding channels to avoid memory leaks.

```
1  class SelectOp {
2    val id: Long // unique
3    val cor: Coroutine
4    var state: Any?
5  }
6
7  fun select(alternatives: SelectAlt[]) {
8    s := SelectOp {cor: curCor(); state: REG}
9    // Registration phase
10   regInfos := emptyList<RegInfo>()
11   for (alt in alternatives) {
12     regInfo := alt.regSelect(s, alt.element)
13     if (regInfo == null)
14       // This select is done, the corresponding
15       // channel is stored in the state field,
16       // the result -- in the coroutine.
17       break
18     regInfos.add(regInfo)
19   }
20   // Waiting phase
21   result := suspendAndGetResult() // does not suspend
22                                   // if was selected
23   // Removing phase
24   for (regInfo in regInfos)
25     regInfo.segment.clean(regInfo.index)
26   // Invoke the of the selected alternative
27   alternatives.find{alt -> alt.channel == channel}
28               .action(result)
29 }
30 class RegInfo { val node: Node; val index: Int }
```

Listing 2: High-level algorithm for `select`

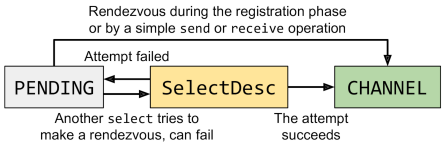


Figure 1: Life-cycle of the `state` in `select` instance (see `SelectOp`).

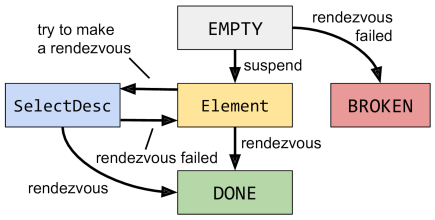


Figure 2: Life-cycle of the `el` field in waiter slot. Broken slot is represented as a waiter with a special `BROKEN` token in `el` field.

**Registration.** In the *registration* phase, the `select` instance is registered in each channel sequentially, using `regSelect` function in Listing 2. It uses a similar algorithm as for the simple `send` and `receive` operations, but instead of adding the current coroutine to the waiting queue, the reference to the `SelectOp` object is stored. If it makes a rendezvous with an opposite operation, we should change the state of this `SelectOp` object from `PENDING` to the corresponding channel atomically, via `CAS`, as shown on Figure 1. This `CAS` can fail if another coroutine has already made

a rendezvous with this `select`, due to a possibility that this `select` instance is already stored as a waiter in other channels. Due to this fact, we also cannot linearize on `deqIdx` increment: this increment claims the slot, but the `select` instance can fail on doing a rendezvous if it is already selected. Therefore, we change the `el` waiter field life-cycle (see Figure 2) and linearize on performing a successful `CAS` from `Element` state to `DONE`. This change allows updating `deqIdx` lazily. However, with this change we have to update two fields atomically: the `state` of this `select` instance and the corresponding `el` field; this successful update is a linearization point of performed by this `select` rendezvous. For this, we use descriptors (`SelectDesc` state in Figure 2) similarly to the Harris lock-free multi-word CAS [14]. Like in the Harris algorithm, concurrent operations that encounter a descriptor in the `el` field help to complete the operation it describes.

**Rendezvous with `select`.** Simple `send` and `receive` operations should atomically change the `select` instance state from `PENDING` to the corresponding channel, what is a linearization point of successful rendezvous with `select`.

A rendezvous between two `select`-s is more complicated, it requires updating both their states from `PENDING` to the corresponding channels, as well as the update in `el` field. Like in the registration phase we use descriptors, and update states to a `SelectDesc` at the beginning of the possible rendezvous (see Figure 1), processing the descriptor after that. It is known by the Harris paper that we have to always set descriptors in the same order to avoid livelocks. For this, we introduce an unique `id` field in `SelectOp`, and order `select` instances using it.

**The removing phase.** During the *removing* phase, we clean the corresponding waiter fields and remove a node if it is full of processed waiter cells; the number of cleaned waiters is maintained via an atomic counter, separately for each node. When this counter reaches `SEGMENT_SIZE`, we consider the node to be cleaned and logically removed from the queue. We physically remove the node from the Michael-Scott queue using `remove` function presented in Listing 2.

To perform removing in constant time, we add a new `prev` field into `Node`, which references the previous node and is initialized to the current tail when it is added to the queue. That helps us to remove nodes from the middle of the queue; however, we forbid removing head and tail. If the node to be removed is head, it is going to be removed after the constant number of increments due to the `head` moving forward in the Michael-Scott algorithm. At the same time, if the tail is fully cleaned, it is not considered as logically removed; it is going to be removed from the queue right after a new tail node is added.

When `remove` operations do not interfere, we first get the previous and next nodes, and then change their `next` (for the previous node) and `prev` (for the next one) links to each other. Our construction guarantees that neither of these `prev` and `next` links are not `null` when the node is neither head nor tail.

However, remove operations on neighbour nodes can interfere with each other and head or tail updates. In order to ensure correctness, we update `prev` field to the closest node of lower id that has not yet been cleaned, and the `next` field to the closest non-cleaned node of larger id. This way, concurrent operations cannot break linked list invariants and effectively help each other to move the `prev` and `next` references after logical removal, moving them to the left and to the right respectively (methods `movePrevToLeft` and `moveNextToRight` in Listing 2, they update `prev` and `next` pointers if the passed node has lower or greater id respectively), to physically remove all cleaned nodes from the list.

We also need to ensure that these previous and next nodes are not logically removed, so we check this invariant after the re-linking and help with removing these previous and next nodes if

```
fun remove(n: Node) {
  next := n.next;
  prev := n.prev
  // check if this segment is not tail
  if (next == null)
    return
  // check if this segment is not head
  if (prev == null)
    return
  // Link next and prev
  prev.moveNextToRight(next)
  next.movePrevToLeft(prev)
  // Help other threads
  if (prev.cleaned == SEGMENT_SIZE)
    prev.remove()
  if (next.cleaned == SEGMENT_SIZE)
    next.remove()
}
```

Listing 2: Removing empty node from the queue

needed.

## 6 Evaluation

We implemented the proposed rendezvous channel algorithm in Kotlin and Go [21]. As a comparison point, we use the optimized implementations provided by the languages; for Kotlin, we also implement and compare against the fair synchronous queue algorithm by Scherer et al. [27].

Go synchronizes channel operations via a coarse lock, and implements a fine-grained locking algorithm for the `select` expression. In Kotlin, all channel operations are lock-free, and use a concurrent doubly linked list, alongside with a descriptor for each operation, which is stored into the list head field and therefore forces other threads to help with the operation first. This way, all operations on a given channel are executing almost sequentially. The fair synchronous queue presented of Scherer et al. [27] is based on the classic Michael-Scott queue algorithm [23]. It is lock-free, but does not support the `select` expression.

**Benchmarks.** Our initial set of experiments consider a single channel to which coroutines apply a series of `send` and `receive` operations. To increase the parallelism level, we increase the maximum number of threads for the coroutines scheduler. We use the following three benchmarks to evaluate the performance:

- *Multiple-producer single-consumer:* This scenario simulates a channel associated with an actor, and shows the potential of using the proposed algorithms in actor-like scenarios. We have the same number of coroutines as the number of threads.
- *Multiple-producer multiple-consumer:* This is a standard benchmark for queue-like data structures. We again have the same number of coroutines as the number of threads.
- *Multiple-producer multiple-consumer with a thousand coroutines.* In CSP programming, it is often the case that one has significantly more coroutines than the number of cores ("oversubscription"). We therefore examine this scenario as well.

To benchmark the `select` expression, we use the same benchmarks, but where all operations inside the `select` expression receive from an empty coroutine-local channel at the same time. This benchmark simulates checking if the coroutine should be cancelled or not by trying to receive a special token from a specific additional channel. This is a widely used pattern in producer-consumer scenarios [1].

**Methodology.** To avoid artificial *long run scenarios* [23], we simulate some amount of work between operations. Specifically, we have threads consume 100 CPU cycles in an non-contended

local loop, which decreases the contention on the channel. Result trends are similar for higher values of this "backoff" term, but tend to have high variance, induced by contention, for much smaller values. We measure the time it takes to send $10^6$ elements over each channel implementation, averaged over 10 runs. This time is then divided by the number of operations, to obtain the results shown. In our algorithm, we have chosen a `NODE_SIZE` size of 32, based on some minimal tuning.

**Platform.** We used a server with 4 Intel Xeon Gold 6150 (Skylake) sockets, each socket has 18 2.70 GHz cores, each of which multiplexes 2 hardware threads, for a total of 144 hardware threads.

### 6.1 Experimental Results

Figure 3 shows the experimental results on different benchmarks. We compare our algorithm with Go (top) and Kotlin coroutines (bottom). In addition, we compare with the fair synchronous queue of Scherer et al. [27], implemented in Kotlin, the results of which are presented on the bottom-side graphs as well. We split the analysis of the results into two parts, considering the performance of plain `send` and `receive` operations, and the `select` expression.

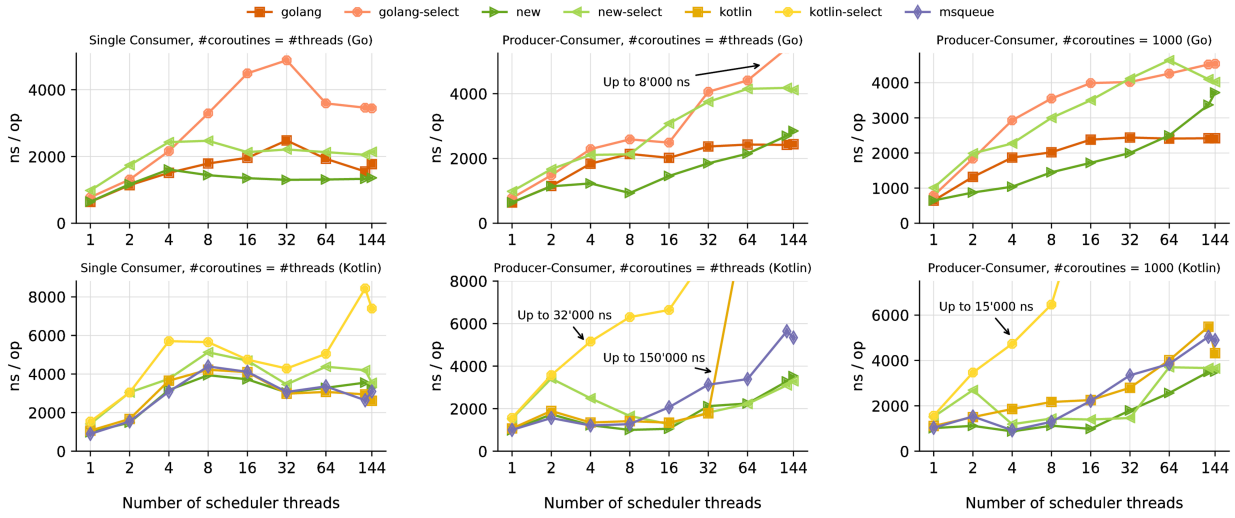


Figure 3: Performance of the proposed channel algorithm compared against Go and Kotlin rendezvous channels, and the FIFO synchronous queue algorithm by Scherer et al. The results of the `select` expression are shown on the same plots, but with the suffix *select* in line titles and using dimmed colors.

**`Send` and `receive` performance.** Our algorithm outperforms the Go implementation in all benchmarks and shows similar performance in the single thread case. A little loss of performance at more than 64 scheduler threads in multi-producer multi-consumer benchmarks is explained by the fact that our algorithm is more complicated to ensure lock-freedom, and therefore suffers in terms of cache performance under high contention.

Kotlin Coroutines implementation of `send` and `receive` works similarly in the single-consumer scenario and is outperformed by our algorithm in all other benchmarks, especially at high thread count. This happens because Kotlin uses a considerably more complex doubly-linked list with descriptors under the hood. Our benchmarks do not show the garbage collection overhead, which should also be decreased significantly with our algorithm.

We found that the baseline Kotlin implementation performed particularly badly for large number of coroutines (see Figure 3, bottom middle). We believe these bad results are due to the recursive helping mechanism employed by this implementation.

Our algorithm for `send` and `receive` improves on the fair synchronous queue by Scherer et al., and shows superior results for all benchmarks. One main difference comes from the fact that we are using a node for several items, which decreases the number of allocations and possible cache misses.

**The `select` expression performance.** Go's implementation uses a lock-based algorithm. In our setup, compared with simple `send` and `receive` operations, the 'select' operation needs to acquire an extra lock for another channel without contention. In contrast, our algorithm needs to create a descriptor for each such operation, and perform an additional `CAS` operations to update the `SelectOp.state` field. Our algorithm also requires a concurrent version of `park/unpark` primitives, which also does an additional `CAS` and degrades the performance. This explains a bit higher cost of our algorithm in low-contended scenarios ($\leq 4$ threads). However, because of no other difference compared with the plain 'send' and 'receive' operations, our algorithm shows the same performance trend with increasing the number of threads, and outperforms Go's implementation by up to $2\times$.

Our algorithm outperforms the lock-free Kotlin baseline implementation in all scenarios. It does so significantly at large thread counts and shows a bit better results on smaller thread counts. Similarly to the simple `send` and `receive` operations analysis, we believe, that so bad Kotlin's implementation behavior on large number of threads is a consequence of a lot of helping.

# 7 Discussion and Future Work

We have presented the first lock-free implementation of a channel supporting complete CSP semantics. Our design is built on several good ideas introduced in the context of lock-free ordered data structures, and introduces some new techniques to handle CSP semantics, in particular, the `select` expression and removing from the middle of a queue-like structure as a part of it. Our implementations [21] in Kotlin and Go outperform the existing baselines and show much better scalability, especially for the `select` expression. We also believe that it is possible to achieve better performance since our implementations are not as good optimized as Go and Kotlin Coroutines ones. In future work, we aim to study further optimizations for our algorithm in the high-contention case, and extend support for additional semantics, such as operation cancellation and channel closing.

# References


[1] Go Concurrency Patterns: Pipelines and cancellation - The Go Blog. `https://blog.golang.org/pipelines`, 2014.

[2] Akka. `https://akka.io/`, 2018.

[3] Erlang Programming Language. `http://www.erlang.org/`, 2018.

[4] Kotlin Coroutines. `https://github.com/Kotlin/kotlin-coroutines`, 2018.

[5] Livelock bug in the Kotling Coroutine Implementation. `https://github.com/Kotlin/kotlinx.coroutines/issues/504`, 2018.

[6] OpenJDK: Loom. http://openjdk.java.net/projects/loom/, 2018.

[7] The Clojure Programming Language. https://clojure.org/, 2018.

[8] The Go Programming Language. https://golang.org/, 2018.

[9] The Rust Programming Language. https://www.rust-lang.org/, 2018.

[10] Ali E Abdallah. *Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers*, volume 3525. Springer Science & Business Media, 2005.

[11] Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *European Conference on Parallel Processing*, pages 151–162. Springer, 2010.

[12] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[13] David R Hanson. *C interfaces and implementations: techniques for creating reusable software*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[14] Timothy L Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*, pages 265–279. Springer, 2002.

[15] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

[16] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable flat-combining based synchronous queues. In *International Symposium on Distributed Computing*, pages 79–93. Springer, 2010.

[17] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[18] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[19] Joseph Izraelevitz and Michael L. Scott. Generality and speed in nonblocking dual containers. *ACM Trans. Parallel Comput.*, 3(4):22:1–22:37, March 2017.

[20] Gilles Kahn and David MacQueen. Coroutines and networks of parallel processes. 1976.

[21] Nikita Koval, Dan Alistarh, and Roman Elizarov. Channel implementations in go and kotlin.

[22] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel & Distributed Systems*, (6):491–504, 2004.

[23] 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*, pages 267–275. ACM, 1996.

[24] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices*, volume 48, pages 103–112. ACM, 2013.

[25] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2007.

[26] William N Scherer III, Doug Lea, and Michael L Scott. A scalable elimination-based exchange channel. *SCOOL 05*, page 83, 2005.

[27] William N Scherer III, Doug Lea, and Michael L Scott. Scalable synchronous queues. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 147–156. ACM, 2006.

[28] WN Scherer III and ML Scott. Nonblocking concurrent objects with condition synchronization. In *Proc. of the 18th Intl. Symp. on Distributed Computing*, 2004.

[29] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks: preliminary version. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 54–63. ACM, 1995.

[30] Nir Shavit and Asaph Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

[31] Håkan Sundell and Philippas Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In *International Conference On Principles Of Distributed Systems*, pages 240–255. Springer, 2004.