Distributed data structures for future many-core architectures

Panagiota Fatourou
FORTH-ICS a& Univ. of Crete
faturu@ics.forth.gr

Nikolaos D. Kallimanis
FORTH-ICS
nkallima@ics.forth.gr

Eleni Kanellou
FORTH-ICS
kanellou@ics.forth.gr

Odysseas Makridakis
Univ. of Crete
odmakryd@csd.uoc.gr

Christi Symeonidou
FORTH-ICS
chsymeon@ics.forth.gr

FORTH-ICS TR 447, APRIL 2015

Abstract. We present general techniques for implementing distributed data structures, such as stacks, queues, deques, lists, and sets, on top of future many-core architectures with non cache-coherent or partially cache-coherent memory. With the goal of contributing towards what might become, in the future, the concurrency utilities package in Java collections for such architectures, we implemented a comprehensive collection of data structures, richer than that provided in java.util.concurrent, by considering different variants of these techniques.

To achieve scalability, we present a generic scheme which can be used to make all our implementations hierarchical. We also describe a large collection of techniques for further improving scalability in most implementations.

We have compiled a library of the proposed data structures and performed experiments on top of a non cache-coherent 512-core architecture which is built using 64 hardware prototyping boards. The experiments illustrate nice scalability characteristics for the proposed techniques and reveal the performance and scalability power of the hierarchical approach.
Contents

1 Introduction 1

2 Theoretical Framework 5
   2.1 Abstract Data Types ........................................... 5
   2.2 Abstract Description of Hardware ............................. 6
   2.3 Theoretical Model .............................................. 7

3 Distributed Data Structures 11
   3.1 Implementation Paradigms ....................................... 11

4 Directory-based Stacks, Queues, and Deques 15
   4.1 Distributed Hash Table .......................................... 15
   4.2 Directory-Based Stack .......................................... 18
      4.2.1 Algorithm Description ..................................... 18
      4.2.2 Proof of Correctness ...................................... 20
   4.3 Directory-Based Queue .......................................... 23
      4.3.1 Algorithm Description ..................................... 23
      4.3.2 Proof of Correctness ...................................... 25
      4.3.3 Queues with Special Functionality ................. 29
   4.4 Directory-Based Double-Ended Queue (Deque) ........... 30
      4.4.1 Algorithm Description ..................................... 30
      4.4.2 Proof of Correctness ...................................... 32
   4.5 Hierarchical approach, Elimination, and Combining .... 38

5 Token-based Stacks, Queues, and Deques 41
   5.1 Token-Based Stack .............................................. 42
      5.1.1 Algorithm Description ..................................... 42
      5.1.2 Proof of Correctness ...................................... 45
   5.2 Token-Based Queue .............................................. 47
      5.2.1 Algorithm Description ..................................... 48
      5.2.2 Proof of Correctness ...................................... 53
   5.3 Token-Based Double-Ended Queue (Deque) ............... 57
      5.3.1 Algorithm Description ..................................... 57
      5.3.2 Proof of Correctness ...................................... 66
   5.4 Hierarchical approach ......................................... 70
Chapter 1

Introduction

High productivity languages, like Java, aim at increasing the productivity of non-expert programmers by simplifying parallel programming. Nowadays, the dominant parallelism paradigm used by most high-level, high productivity languages is that of threads and cache-coherent shared memory among all cores. However, cache-coherence does not scale well with the number of cores [1]. So, future many-core architectures, which will offer hundreds or even thousands of cores, are not expected to support cache-coherence across all cores. They would rather feature multiple coherence islands, each comprised of a number of cores (or a number of processors with more than one core each), which will share a coherent view of a part of the memory, but no hardware cache-coherence will be provided among cores of different islands. Instead, the coherence islands will be interconnected using fast communication channels. In recent literature, we meet even more aggressive approaches with Intel having proposed two fully non cache-coherent architectures, the SCC [2] and the Runnemede [3]. Additionally, [4] presents the FORMIC board, a 512-core non cache-coherent prototype.

Such architectures impose additional effort in programming them, since they require to explicitly code all communication and synchronization using messages between processors. This is tedious and difficult, as the programmer needs to reason about load balancing, distributing data among processors, explicit communication and synchronization. Previous works [5, 6, 7, 8, 9] indicate the community’s interest in bridging the gap between non cache-coherent or distributed architectures, and high-productivity programming languages by implementing runtime environments, like the Java Virtual Machine (JVM), for such architectures, which maintain the shared-memory abstraction.

In the GreenVM project we indeed undertook the task of making a significant step forward in bridging this gap, by porting the Java Runtime Environment for such architectures.

The difficulty in parallelizing many applications comes from those parts of the computation that require communication and synchronization via
data structures. Thus, the design of effective concurrent data structures is crucial for many applications. On this avenue, Java’s concurrency utilities package (JSR 166) [10, 11] provides a powerful framework of high-performance threading utilities, including a wide collection of concurrent data structures [10, 12, 13, 14]. However, to the best of our knowledge, the package targets cache-coherent shared-memory architectures.

To run Java-written programs on a non cache-coherent architecture, Java’s VM must be ported to that architecture, and some fundamental communication and synchronization primitives (such as CAS, locks, and others) must be implemented. Normally, once this is done, it will be possible to execute applications that employ the concurrency utilities package without any code modification. However, the algorithms provided in the package have been chosen to perform well on shared memory architectures. So, they take no advantage of the communication and synchronization features of non cache-coherent architectures and do not cope with load balancing issues or with the distribution of data among processors. Thus, they are expected to be inefficient when executing through JVMs ported for such architectures, even if optimized implementations of locks, CAS, and other primitives are provided on top of the architecture. Therefore, there is an urgent need to develop novel data structures and algorithms, optimized for non cache-coherent architectures. This is exactly what we address in Workpackage 2 of GreenVM. This deliverable gives detailed descriptions of how we accomplish this task.

We study general techniques for implementing distributed data structures, such as stacks, queues, deques, lists, and sets, on top of many-core architectures with non or partially cache-coherent memory. These techniques exhibit different properties to address different workloads, and exploit locality and/or the communication characteristics of the machine. With the goal of contributing to what might become, in the future, the concurrency utilities package in Java collections for such architectures, we end up, by considering different variants of these techniques, with a comprehensive collection of data structures, richer than those provided in java.util.concurrent. Our collection, which is based on message-passing to achieve the best of performance, facilitates the execution of Java written code on non-cache coherent architectures without any modification in a highly efficient way.

To achieve scalability, that is maintaining good performance as the number of cores increases [15], we present a generic scheme, which can be used to make all our implementations hierarchical [16, 17, 18, 19, 20, 21, 22]. The hierarchical version of an implementation exploits the memory structure and the communication characteristics of the architecture to achieve better performance. Specifically, a core (not necessarily always the same) from each island, the island master, participates in the execution of the implemented distributed algorithm, whereas the rest submit their requests to this core. Depending on the implemented data structure, the island master employs
elimination [23], combining [18, 24, 19], batching and other techniques to achieve better scalability and performance. In the partially cache-coherent case, the cores of the same island may synchronize by employing a combining synchronization algorithm [18, 19]. To execute the implemented distributed algorithm, however, the island masters (which then are the combiners of different islands) must exploit the communication primitives provided for fast communication among different islands. In architectures with thousands of cores, we could employ a more advanced hierarchical structure of intermediate masters for better scalability.

For efficiency, in some of our implementations, we employ a highly scalable distributed hash table (DHT) which uses a simple standard technique [25, 26, 27] to distribute the data on different nodes. Based on it and by employing counting networks [28, 29], we can come up with fully decentralized, scalable implementations of queues and stacks. We also present implementations of (sorted and unsorted) lists, some of which support complex operations like range queries. To design our algorithms, we derive a theoretical framework which captures the communication characteristics of non cache-coherent architectures. This framework may be of independent interest. In this spirit, we further provide full theoretical proofs of correctness for the algorithms we present in this deliverable.

We have implemented the proposed data structures on top of the encore machine and have compiled a powerful data structure library that takes into account the architecture characteristics in order to use them to its advantage.

We have performed experiments on top of a non cache-coherent 512-core architecture, built using FORMIC boards [4]; FORMIC is a hardware prototyping board. The experiments illustrate nice scalability characteristics for some of the proposed techniques and reveal the performance and scalability power of the hierarchical approach.
Chapter 2

Theoretical Framework

2.1 Abstract Data Types

For the sake of completeness, the abstract data types that are referenced throughout this paper are briefly defined below.

- **Stack.** An ordered sequence of elements that supports the LIFO property, i.e. the sequence can only be modified by appending or removing an element from only one end of the sequence, referred to as the *top*. A stack $S$ provides the following operations: (i) $\text{Push}(x, S)$: adds element $x$ at the end of $S$ and returns $\text{true}$ (it returns $\text{false}$ if the stack is full); (ii) $\text{Pop}(S)$: removes the top element of $S$ and returns it if $S$ is non-empty, returns $\text{false}$ if $S$ is empty; (iii) $\text{Top}(S)$: returns the top element of $S$ if $S$ is non-empty, returns $\text{false}$ if $S$ is empty; (iv) $\text{MakeEmptyStack}()$: returns an empty stack; (v) $\text{IsEmptyStack}(S)$: if $S$ is non-empty, returns $\text{true}$, otherwise $\text{false}$.

- **Queue.** An ordered sequence of elements that supports the FIFO property, i.e. a new element can only be appended at one end of the sequence, namely the *tail*, and an element can only be removed from the other end of the sequence, namely the *head*. A queue $Q$ provides the following operations: (i) $\text{Enqueue}(x, Q)$: adds element $x$ to the tail of $Q$, returns $\text{true}$ (it returns $\text{false}$ if the queue is full); (ii) $\text{Dequeue}(Q)$: removes the element at the head of $Q$, returns $\text{true}$ if successful, $\text{false}$ if $Q$ is empty; (iii) $\text{Front}(Q)$: returns the element at the head of $Q$, $\text{false}$ if $Q$ is empty; (iv) $\text{MakeEmptyQueue}()$: returns an empty queue; (v) $\text{IsEmptyQueue}(Q)$: if $Q$ is non-empty, returns $\text{true}$, otherwise $\text{false}$.

- **Set.** A collection of elements, where each element takes a value from some domain $U$. A set does not contain duplicates of elements. We are mainly interested for a special case of sets, known as dictionaries. A dictionary contains key-value pairs. For every such pair $(k, v)$,
the field \( k \) takes a unique value from some domain \( U \). A dictionary \( D \) provides the following operations: (i) \( \text{Insert}(k, v, D) \): inserts \( ⟨k, v⟩ \) into \( D \), returns \text{true} if such a pair is not already contained in \( D \), returns \text{false} otherwise; (ii) \( \text{Delete}(k, D) \): removes \( ⟨k, v⟩ \) from \( D \) and returns \text{true} if such a pair was contained in there, returns \text{false} otherwise; (iii) \( \text{MakeEmptySet}() \): returns an empty dictionary \( D \); (iv) \( \text{LookUp}(k, D) \): returns \( v \) if a pair \( ⟨k, v⟩ \) is contained in \( D \), returns \text{false} otherwise; (v) \( \text{IsEmptySet}(D) \): returns \text{true} if \( D \) is empty, \text{false} otherwise.

2.2 Abstract Description of Hardware

Inspired by the characteristics of non cache-coherent architectures [3] and prototypes [4], we consider an architecture which features \( m \) islands (or clusters), each comprised of \( c \) cores (located in one or more processors). The main memory is split into modules, with each module associated to a distinct island (or core). A fast cache memory is located close to each core. No hardware cache-coherence is provided among cores of different islands: different copies of the same variable residing on caches of different islands may be inconsistent. The islands are interconnected with fast communication channels.

A process can send messages to other processes by invoking \text{send} and it can receive messages from other processes by invoking \text{receive}. The architecture may provide cache-coherence for the memory modules of an island to processes executing on the cores of the island, i.e. the cores of the same island may see the memory modules of the island as cache-coherent shared memory. If this is so, we say that the architecture is partially non cache-coherent; otherwise, it is fully non cache-coherent. The following means of communication between cores (of the same or different islands) are provided:

\text{Send/Receive mechanism}: Each core has its own mailbox which is a (hardware-implemented) FIFO queue. (If more than one processes are executed on the same core, they share the same hardware mailbox; in this case, the functionality of \text{send} and \text{receive} can be provided to each process through the use of software mailboxes.) A process executing on the core can send messages to other processes by invoking \text{send}, and it can receive messages from other processes by invoking \text{receive}. Messages are not lost and are delivered in FIFO order. An invocation of \text{receive} blocks until the requested message arrives. The first parameter of an invocation to \text{send} determines the core identifier to which the message is sent.

\text{Reads and Writes through DMA}: A Direct Memory Access (DMA) engine allows certain hardware subsystems to access the system’s memory without any interference with the CPU. We assume that each core can perform \( \text{DMA}(A, B, d) \) to copy a memory chunk of size \( d \) from memory address \( A \)
to memory address $B$ using a DMA (where $A$ and $B$ may be addresses in local or a remote memory module). We remark that DMA is not executed atomically. It rather consists of a sequence of atomic reads of smaller parts, e.g. one or a few words, (known as bursts) of the memory chunk to be transferred, and atomic writes of these small parts to the other memory module. DMA can be used for performance optimization. Once the size of the memory chunk to be transferred becomes larger (by a small multiplicative factor) than the maximum message size supported by the architecture, it is more efficient to realize the transfer using DMA (in comparison to sending messages). Specifically, we denote by $MMS$ the maximum size of a message supported by the architecture. (Usually, this size is equal to either a few words or a cache line). Consider that the chunk of data that a core wants to send has size equal to $B$. To send these data using messages, the core must send $MMS/B$ messages. Each message has a cost $CM$ to set it up. So, in total, to transfer the data using messages, the overhead paid is $(MMS/B) \times CM$. Setting up a DMA has a cost $CD$, which in most architectures is by a small constant factor larger than $CM$. However, $CD$ is paid only once for the entire transfer of the chunk of data since it is done with a single DMA. Additionally, sending and receiving $MMS/B$ messages requires that a core’s CPU will be involved $MMS/B$ times to send each message and another core’s CPU will be involved $MMS/B$ times to receive these messages. This cost will be greatly avoided when using a single DMA to transfer the entire chunk of data. Thus, it is beneficial in terms of performance, to use DMA for transferring data whenever the size of the data to be transferred is not too small.

The architecture may provide cache coherence for the memory modules of an island to the cores executing on the island, i.e. the cores of the same island may see the memory modules of the island as cache-coherent shared memory. If this is so, we say that the architecture is partially non cache-coherent; otherwise, it is fully non cache-coherent.

2.3 Theoretical Model

An implementation of a data structure (DS) stores its state in the memory modules and provides an algorithm, for each process, to implement each operation supported by the DS. We model the submission and delivery of messages sent by processes by including incoming and outgoing message buffers in the state of each process (as described in standard books [30] on distributed computing). We model each process as a state machine. We model a DMA request as a simple program which contains a sequence of interleaved burst reads from a memory module and burst writes to a memory module. A DMA engine executes sequences of DMA requests, so it can also be modeled as a simple state machine whose state includes a DMA buffer storing DMA requests that are to be executed. A configuration is a
vector describing the state of each process (including its message buffers),
the state of each DMA engine, the state of the caches (or the shared variables
in case shared memory is supported among the cores of each island), and
the states of the memory modules. In an initial configuration, each process
and DMA engine is in an initial state, the shared variables and the memory
modules are in initial states and all message and DMA buffers are empty.
An event can be either a step by some process, a step by a DMA engine, or
the delivery of a message; in one step, a process may either transmit exactly
one message to some process and at least one message to every other process,
access (read or write) exactly one shared variable, initiate a DMA transfer,
invoke an operation of the implemented DS. In a DMA step a burst is read
from or written to a memory module. All steps of a process should follow the
process’s algorithm. Similarly, all steps of a DMA engine should be steps of
the simple program that performs a DMA request submitted to this DMA
engine.

An execution is an alternating sequence of configurations and steps start-
ing with an initial configuration. The execution interval of an instance of
an operation op in an execution α is the subsequence of α starting with the
configuration preceding the invocation of this instance of op and ending with
the configurations that follows its response. A step is enabled at a configu-
ration C, if the process or DMA engine will execute this step next time it
will be scheduled. A finite execution α is fair if, for each process p and each
DMA engine e, no step by p or e is enabled at the final configuration C of α
and all messages sent in α have been delivered by C. An infinite execution
α is fair if the following hold:

- for each process p, either p takes infinitely many steps in α, or there
  are infinitely many configurations in α such that in each of them (1)
  no step by p is enabled, (2) for every prefix of α that ends at such a
  configuration, all messages sent by p have been delivered.
- for each DMA engine e, either e takes infinitely many steps in α, or
  there are infinitely many configurations in α such that in each of them
  no step by e is enabled (we remark that a DMA engine e always have
  an enabled step as long as its DMA buffer is not empty).

Correctness. For correctness, we consider linearizability [31]. This means
that, for every execution, one can assign a linearization point to each com-
pleted operation (and to some of the uncompleted operations) so that the
linearization point of each operation occurs after the operation starts and
before it ends, and the results of these operations are the same as if they
had been performed sequentially, in the order of their linearization points.

Progress. We aim at designing algorithms that always terminate, i.e. reach
a state where all messages sent have been delivered and no step is enabled.

Communication Complexity. Communication between the cores of the
same island is usually faster than that across islands. Thus, the commu-
nication complexity of an algorithm for a non cache-coherent architecture
is measured in two different levels, namely the intra-island communication and the inter-island communication. The *intra-island communication complexity* of an instance $inst$ of an operation $op$ in an execution $\alpha$ is the total number of messages sent by every core $c$ to cores residing on different islands from that of $c$ for executing $inst$ in $\alpha$. The intra-island communication complexity of $op$ in $\alpha$ is the maximum, over all instances $inst$ of $op$ in $\alpha$, of the intra-island communication complexity of $inst$ in $\alpha$. The intra-island communication complexity of $op$ in $\alpha$ is the maximum, over all instances $inst$ of $op$ in $\alpha$, of the intra-island communication complexity of $inst$ in $\alpha$. We remark that communication can be measured in a more fine-grained way in terms of bytes transferred instead of messages sent, as described in [32]. For simplicity, we focus on the higher abstraction of measuring just the number of messages as described in [30].

If the architecture is fully non cache-coherent, then the *inter-island communication complexity* is defined as follows. The inter-island communication complexity of an instance $inst$ of $op$ in $\alpha$ is the maximum, over all islands, of the total number of messages sent by every core $c$ of an island to cores residing on the same island as $c$ for executing $inst$ in $\alpha$.

If the architecture is partially non cache-coherent, then we measure the inter-island communication complexity in terms of cache misses following the cache-coherence (CC) shared-memory model (see e.g. [33, 34]). Specifically, in the (CC) shared memory model, accesses to shared variables are performed via cached copies of them; an access to a shared variable is a *cache miss* if the cached copy of this variable is invalid. In this case, a cache miss occurs and a valid copy of the variable should be fetched in the local cache first before it can be accessed. Once the cache miss is served and as long as the variable is not updated by processes that are being executed on other cores, future accesses to the variable by processes that are being executed on this core do not lead to further cache misses. In such a model, the inter-island communication complexity of an instance $inst$ of an operation $op$ is the maximum, over all islands, of the total number of cache-misses that the cores of the island experience to execute $inst$.

We remark that independently of whether the architecture is partially or fully non cache-coherent, the inter-island communication complexity of $op$ in $\alpha$ is the maximum, over all instances $inst$ of $op$ in $\alpha$, of the inter-island communication complexity of $inst$ in $\alpha$. Moreover, the inter-island communication complexity of $op$ for an implementation $I$ is the maximum, over all executions $\alpha$ produced by $I$, of the inter-island communication complexity of $op$ in $\alpha$.

The *DMA communication complexity* of an instance $inst$ of an operation $op$, is the total number of DMA requests initiated by every process to execute $inst$; in a more fine-grained model, we could instead measure the total number of bursts performed by these DMA requests. The DMA complexity of $op$ in $\alpha$ and the DMA communication complexity of $op$ in $I$ are defined
as for the other types of communication complexities.

**Time complexity.** Consider a fair execution $\alpha$ of an implementation $I$. A timed version of $\alpha$ is an enhanced version of $\alpha$ where each event has been associated to a non-negative real number, the time at which that event occurs. We define the delay of a message in a timed version of $\alpha$ to be the time that elapses between the computation event that sends the message and the event that delivers the message. We denote by $T_\alpha$ those timed versions of $\alpha$ for which the following conditions hold: (1) the times must start at 0, (2) must be strictly increasing for each individual process and the same must hold for each individual DMA engine, (3) must increase without bound if the execution is infinite, (4) the timestamps of two subsequent events by the same process (or the same DMA engine) must differ by at most 1, and (4) the delay of each message sent must be no more than one time unit. Let $T = \cup_{\forall \alpha}$ produced by $I\{T_\alpha\}$.

The *time* until some event $\rho$ is executed in an execution $\alpha$ is the supremum of the times that can be assigned to $\rho$ in all timed versions of $\alpha$ in $T_\alpha$. The *time* between two events in $\alpha$ is the supremum of the differences between the times in all timed versions of $\alpha$ in $T_\alpha$. The time complexity of an instance $\text{inst}$ of an operation $\text{op}$ in $\alpha$ is the time between the events of its invocation and its response. The time complexity of $\text{op}$ in $\alpha$ is the maximum, over all instances $\text{inst}$ of $\text{op}$ in $\alpha$, of the time complexity of $\text{inst}$ in $\alpha$. The time complexity of an operation $\text{op}$ for $I$ is the maximum, over all executions $\alpha$ produced by $I$, of the time complexity of $\text{op}$ in $\alpha$.

**Space Complexity.** The *space complexity* of $I$ is determined by the memory overhead introduced by $I$, and by the number and type of shared variables employed (in case of partially non cache-coherence).
Chapter 3

Distributed Data Structures

This section describes the distributed data structures we designed and implemented. We provide several designs and implementations for a comprehensive collection of data structures. These are Stacks, Queues, Deques, Lists, Hash Tables and Binary Search Trees.

3.1 Implementation Paradigms

In this work, we propose three major methods for designing and implementing efficient data structures. In all implementations, we appoint a number of $NS$ out of the processes as servers; the rest of the processes act as clients. The exact value of $NS$ can be tuned for best performance. For simplicity, we assume that it is a single process that runs on each core; our implementations work even if this is not so. The servers store parts of the distributed data structure in their local memory and act as synchronization managers for the operations they receive. We study three major methods for designing and implementing efficient data structures:

Centralized. This paradigm is included mainly for experimental purposes. The entire data structure is stored in the local memory of a statically designated server $s_c$. This central server also carries out the synchronization to the data structure. This approach could be the preferred choice for workloads where the status of the data structure does not become too big and contention is low. Even when contention is high, elimination [23] can make this approach scalable for stacks and deques (given that their states do not become too large).

Directory-based approach. The state of the data structure is stored in a highly-scalable distributed directory (which, in this paper, is implemented as a hash table), so it is distributed over the local memory modules of the $NS$ servers. To perform an operation, each client must first access a fetch&add object to get a sequence number which it uses as the key for the requested data. This object can be implemented using a designated server. The client then communicates with the appropriate server to complete its operation.
This approach suits better to workloads where the state of the data structure becomes too large to fit in the local memory of one server. In cases of high contention, we employ elimination [23], combining [24], batching, and we implement the fetch&add object using counting networks [28, 29] to make the approach scalable for stacks, queues and deques; we expect that this approach will be the best choice for such data structures in most cases.

**Token-based approach.** The storage of the data structure is distributed over the local memory of the NS servers in a way that subsequent elements of the data structure are stored in the same server to exploit locality. Each server synchronizes the access to the part of the data structure that is stored in its local memory. The proposed implementations are as simple as their centralized analogs if the state of the data structure does not evolve to be large but they distribute the state of the data structure to more servers otherwise. We expect that this approach could be the preferable choice in cases where the size of the state of the data structure is unknown at the beginning of the execution and cannot be easily predicted. More importantly, starting from this approach, we were able to build more advanced data structures, like (sorted or not) lists. Remarkably, a slightly modified version of our sorted list implementation supports advanced operations, like range-queries, for free without having to communicate with a different server to access each list element during traversals.

**The hierarchical approach.** To exploit the fast communication between the cores of the same island, one process from each island \( i \), the island master \( m_i \), gathers requests from clients located on island \( i \) and forwards them to the appropriate servers. To minimize the number of messages sent to servers, \( m_i \) batches several requests in one or more memory chunks. Then, \( m_i \) may choose to transfer these memory chunks to the servers using DMA. To further reduce the number of messages, a server could also batch the responses for requests initiated by clients of island \( i \) and send them to \( m_i \) which forwards them to the appropriate clients.

In non cache-coherent architectures, a client submits a request to \( m_i \) by sending it a message; \( m_i \) sets a timeout waiting for requests by different clients of its island to arrive. In partially cache-coherent architectures, an instance of a combining synchronization algorithm [18, 19] can be used in each island with all clients of the island participating to the protocol. A combining synchronization algorithm employs a list, which stores requests of active clients from the island. After announcing its request by placing a node in the list, a client tries to acquire a global lock. The client that manages to acquire the lock, called the combiner, serves, in addition to its own request, other active requests recorded in the list. Thus, at each point in time, the combiner plays the role of the island master. When the island master receives (a batch of) responses from a server, it records each of them in the appropriate element of the request list to inform active clients of the island about the completion of their requests. In the meantime, each such
client performs spinning (on its element) until either the response for its request has been fulfilled by the island master or the global lock has been released.

The simple one-level hierarchical scheme of island masters, described above, can easily be generalized to work for more layers of intermediate masters (in a tree-like fashion). The number of intermediate masters and the number of layers can be tuned for achieving the best performance.

For simplicity, the algorithms below are presented for non cache-coherent architectures. For partially cache-coherent architectures, it is only the clients which play the role of an island master that execute the client actions described below, at each point in time; the rest of the clients execute the combining synchronization protocol presented in [18, 19].

In our list implementations, we assume that elements have distinct keys. However, duplicate keys could easily be supported by maintaining a counter in each list element counting the number of times that the key of the element has been inserted in the list.
Chapter 4

Directory-based Stacks, Queues, and Deques

We start with an informal description of the directory-based technique presented in this section. The directory is a data structure that supports the operations \texttt{DirInsert}, \texttt{DirDelete}, and \texttt{DirSearch}. Although the directory can be implemented with several different ways, we employ a simple highly-efficient distributed hash table implementation (also met in [25, 26, 27]) where hash collisions are resolved by using hash chains, called \textit{buckets}. Each server stores a number of buckets. For simplicity, we consider a simple hash function which employs \texttt{mod} and works even if the key is a negative integer. The hash function returns an index which is used to find the server where a request must be sent, as well as the appropriate bucket at this server in which the element resides (or must be stored). Then (to apply the request), a message to this server is sent; the server locally processes the request and responds to the process that initiated it. One of the servers, denoted \( s_s \), acts as the \textit{synchronizer}. Its main job is to assign a unique sequence number \( k \) to each element \( e \) inserted in the data structure \( DS \); this number serves as the key of \( e \).

The hash table implementation we use as our directory is presented in Section 4.1, for completeness. Similar hash table designs have been presented (or discussed) in [25, 27, 26]. Section 4.2 presents the details of the directory-based distributed stack. The directory-based queue implementation appears in Section 4.3. Section 4.4 provides the token-based deque. We remark that our directory-based data structures would work even when using a different directory implementation.

4.1 Distributed Hash Table

A \textit{hash table} stores elements, each containing a key and a value (associated with the key). It is comprised of a table which contains pointers to buckets; each bucket can store several such elements. The operations supported by a
hash table are *insert*, *search* and *delete*. Insert stores a new key-value pair in the hash table, if no element with this key already exists in it; Search looks for a given key in the hash table; Delete searches for an element in the hash table and removes it (if it exists).

Each server stores hash table elements to a local data structure. This structure can be a smaller hash table (as in Algorithm 1), or any other data structure (array, list, tree, etc.) To perform an operation (INSERT, SEARCH or DELETE), a client $c$ finds the appropriate server to submit its request by hashing the key value of interest. This hash value identifies the id of the server that manages the partition of the hash table where the element must be stored. Then, it sends a message to this server, which performs the operation and sends back the result to $c$.

A server $s$ processes all incoming messages sequentially. Each message $s$ receives has four fields: (1) the $op$ field that denotes the type of the operation (INSERT, SEARCH or DELETE), (2) the $key$ field that contains the key, (3) the $data$ field, which has a value in case of an insert and is equal to $\perp$ otherwise, (3) and the $cid$ field which contains the id of the client that initiated the transmission of the message. Event-driven pseudocode for a server $s$ is described in Algorithm 1. Upon receiving a message (line 2), $s$ checks whether its type is INSERT (line 3), SEARCH (line 6) or DELETE (line 9). Depending on the type of the request, the server is going to invoke the appropriate function each time, and then send the function’s result back to the client.

**Algorithm 1** Events triggered in a hash table server.

```plaintext
1   HashTable buckets = ∅;
2   if (op == INSERT) {
3       status = insert(buckets, key, data);
4       send(cid, status);
5   } else if (op == SEARCH) {
6       status = search(buckets, key);
7       send(cid, status);
8   } else if (op == DELETE) {
9       status = delete(buckets, key);
10      send(cid, status);
11   }
```

If the request was for an insert (line 3), the server calls the `insert()` function. This function searches the local buckets for a previously inserted element with the same key. If such an element is found, `insert()` returns a negative acknowledgement (NACK), denoting that the new element is already in the hash table. If the key was not found, it stores it and returns
an acknowledgement (ACK). The response from `insert()` is returned to the client.

For the SEARCH and DELETE messages the action sequence is the same. If the server receives a SEARCH, it executes the function `search()` that searches for the key. If it is not found, `search()` returns NACK and the value of the pair, otherwise. If the server receives a DELETE, it is going to execute the function `delete()` that searches for the key in order to delete it. If the key is found, it deletes it and returns ACK. Otherwise, it returns NACK.

```
Algorithm 2 Insert, search and delete operations of a client of the hash table.
12 boolean DirInsert(int cid, Data data, int key) {
13    sid = hash_function(key);
14    send(sid, ⟨INSERT, data, key, cid⟩);
15    status = receive(sid);
16    return status;
17 }
18 boolean DirSearch(int cid, int key) {
19    sid = hash_function(key);
20    send(sid, ⟨SEARCH, ⊥, key, cid⟩);
21    status = receive(sid);
22    return status;
23 }
24 boolean DirDelete(int cid, int key) {
25    sid = hash_function(key);
26    send(sid, ⟨DELETE, ⊥, key, cid⟩);
27    status = receive(sid);
28    return status;
29 }
```

The `DirInsert()`, `DirSearch()` and `DirDelete()` functions called by the clients are described in Algorithm 2. These functions have all the same instructions, but they differ in the type of message that the clients send towards the servers, as seen on lines 14, 19, and 24, respectively. After receiving a response message from the server, all client functions return a boolean value depending on whether the operation was successful or not (lines 16, 21, and 26, respectively).
4.2 Directory-Based Stack

To implement a stack, the synchronizer $s_s$ maintains a variable $top$ which stores the key of the topmost element of the stack at each point in time. A client $c$ sends a PUSH (POP) request to $s_s$ to obtain a key $k$. When $s_s$ processes such a PUSH (POP) request, $top$ is incremented (decremented) and sent as $k$ to $c$. Then, $c$ uses $k$ as the input argument to DirInsert (DirDelete). Below, a detailed description of the algorithm is given.

4.2.1 Algorithm Description

To apply an operation $op$ a client sends a message to the synchronizer $s_s$. If $op$ is a push operation, $s_s$ uses $top$ variable to assign unique keys to the newly inserted data. Each time it receives a push request, $s_s$ sends the value stored in $top$ to the client after incrementing it by one. Once a client receives a key from $s_s$ for the push operation it has initiated, it inserts the new element in the directory by invoking DirInsert. Similarly, if $op$ is a pop operation, $s_s$ sends the value stored in $top$ to the client and decrements it by one. The client then invokes DirDelete repeatedly, until it successfully removes from the directory the element with the received key.

The synchronizer receives, processes, and responds to clients’ messages. Event-driven pseudocode for the synchronizer is described in Algorithm 3. The messages have an $op$ field that represents the operation to be performed (PUSH or POP), and a $cid$ field with the client’s identification number, needed for identifying the appropriate client to communicate with.

Algorithm 3 Events triggered in the synchronizer of the directory-based stack.

```
1 int top_key = -1;
2
3 a message ⟨op, cid⟩ is received:
4   if (op == PUSH) {
5       top_key ++;
6       send(cid, top_key);
7   } else if (op == POP) {
8       if (top_key == -1) {
9           send(cid, NACK);
10      } else {
11       send(cid, top_key);
12       top_key --;
13     }
14 }
```

When $s_s$ receives a message from the client it first checks its $op$ field. If the type of the message is PUSH (line 3) it first increments the value of $top_key$ by one and sends a message to the client containing it.
When a message of type POP arrives to $s$, it first checks whether the value of $top_key$ is $-1$ (line 7). If this is so, the stack is empty, so the synchronizer responds with a NACK (line 8). Otherwise, it sends a message with the value of $top_key$ to the client (line 10) and then decrements $top_key$ (line 11).

The code for the ClientPush() operation, is presented in Algorithm 4. ClientPush() has the addition of the directory insert that the client performs itself (line 16). The client is free to insert the element lazily, since it has obtained a unique key. This key was returned by the $s$ and since the augmentation of the keys is performed only by $s$, the client can be sure that the key it received is not held by other clients at the same time.

**Algorithm 4** Push operation for a client of the directory-based stack.

```c
12 void ClientPush(int cid, Data data) {
13     sid = get the synchronizer id;
14     send(sid, ⟨PUSH, cid⟩);
15     key = receive(sid);
16     status = DirInsert(key, data);
17     return status;
18 }
```

The ClientPop() function, presented in Algorithm 5 is analogous to the push operation: it sends a POP message to $s$ and waits for its response (line 21). Using the key that was received as argument, DirDelete() is repeatedly called (line 26). This is necessary since another client responsible for inserting the key may not have finished yet its insertion. In this case DirDelete returns $\bot$ (line 27). However, since the key was generated previously by $s$, it is certain that it will be eventually inserted into the directory service.

**Algorithm 5** Pop operation for a client of the directory-based stack.

```c
18 Data ClientPop(int cid) {
19     sid = get the synchronizer id;
20     send(sid, ⟨POP, cid⟩);
21     key = receive(sid);
22     if (key == NACK) {
23         status = ⊥
24     } else {
25         do {
26             status = DirDelete(key);
27         } while (status == ⊥);
28     }
29     return status;
30 }
```
4.2.2 Proof of Correctness

Let \( \alpha \) be an execution of the directory-based stack implementation. We assign linearization points to push and pop operations in \( \alpha \) as follows: The linearization point of a push operation \( op \) is placed in the configuration resulting from the execution of line 16 for \( op \) by the client that invoked it. If \( op \) is a pop operation and line 8 is executed for \( op \) by \( s_s \), then the linearization point is placed in the resulting configuration. If \( op \) is a pop operation for which line 10 is executed by \( s_s \), then we distinguish two cases. Let \( op' \) be that push operation, which inserts into the directory the element that \( op \) removes. If the linearization point of \( op' \) occurs before or at the execution of line 10 for \( op \), then \( op \) is linearized in the configuration resulting from the execution of this line. Otherwise, the linearization point of \( op \) is placed right after the linearization point of \( op' \).

Lemma 1. The linearization point of a push (pop) operation \( op \) is placed within its execution interval.

Proof. Inspection of the pseudocode easily shows that the claim holds for push operations, as the execution of the line after which the linearization point is placed, takes place after the invocation and before the response of the operation.

Assume now that \( op \) is a pop operation invoked by client \( c \) and assume that \( op \) removes an element with key \( k \) from the directory. Let \( op' \) be the push operation that inserts this element in the directory. Let \( C \) be the configuration in the first \texttt{do-while} loop iteration of lines 25 - 27, in which the execution of \texttt{DirDelete} does not return \texttt{⊥}. Let \( C' \) be the configuration resulting from the execution of \texttt{DirInsert} on line 16 by \( op' \), after which the element with key \( k \) is inserted in the directory by \( op' \). We consider two cases.

First, assume that \( C' \) precedes the execution of line 10 for \( op \) by \( s_s \). In this case, the linearization point of \( op \) is placed in the configuration resulting from the execution of line 10 for \( op \) by \( s_s \). Inspection of the pseudocode shows that this line is executed by \( s_s \) for \( op \) after \( s_s \) receives from \( c \) the message that is sent by executing line 20, i.e. after \texttt{ClientPop} is invoked. Further inspection shows that \( c \) blocks (line 21) until it receives from \( s_s \) the message sent on line 10. This means that \texttt{ClientPop}, and therefore, \( op \), does not respond before line 10 is executed. The above implies that the linearization point of \( op \) is included in its execution interval.

Assume next that \( C' \) follows the execution of line 10 for \( op \) by \( s_s \). Following the same argumentation as for the previous case, we have that the execution of that line occurs in the execution interval of \( op \). From the definitions of \( C \) and \( C' \), we further have that \( C' \) happens before \( C \), since the element that \( op' \) inserts in the directory by using \texttt{DirInsert}, is the element that \( op \) removes from the directory in \( C \). Recall that by the way that the linearization points are assigned, the linearization point of \( op' \) is placed in
$C'$. Since $C$ is included in the execution interval of $op$ and $C'$ occurs after the execution of line 10 and before $C$, and given that the linearization point of $op$ is in this case also placed in $C'$, it follows that the linearization point for $op$ is included in its execution interval.

Notice that the argument for the case where $op$ receives a response from $s_s$ because $s_s$ executes line 8, is analogous with the case where $C'$ precedes the execution of line 10 for $op$ by $s_s$.

Thus, the claim holds for all cases.

Notice that since only $s_s$ executes Algorithm 3, we have the following.

**Observation 2.** Instances of Algorithm 3 are executed sequentially, i.e. their execution does not overlap.

Further inspection of the pseudocode of Algorithm 3 indicates that the value of $top$ is incremented before an element is inserted into the directory and decremented before one is removed from the directory. This implies the following observation.

**Observation 3.** The value of $top$ is equal to $-1$ when as many elements have been inserted in the directory as have been removed.

Denote by $L$ the sequence of operations (which have been assigned linearization points) in the order determined by their linearization points. Let $C_i$ be the configuration in which the $i$-th operation $op_i$ of $L$ is linearized. Denote by $\alpha_i$, the prefix of $\alpha$ which ends with $C_i$ and let $L_i$ be the prefix of $L$ up until the operation that is linearized at $C_i$. We denote by $top_i$ the value of the local variable $top$ at configuration $C_i$; let $top_0 = 0$. Denote by $S_i$ the sequential stack that results if the operations of $L_i$ are applied sequentially to an initially empty stack. Denote by $d_i$ the number of elements in $S_i$. We associate a sequence number with each stack element such that the elements from the bottommost to the topmost are assigned $1, \ldots, d_i$, respectively. Denote by $sl_{d_i}$ the $d_i$-th element of $S_i$.

**Lemma 4.** For each integer $i > 0$, it holds that if $op_i$ is a pop operation, then it returns the value of the field data of $sl_{d_i - 1}$ if $S_{i-1} \neq \epsilon$, or $\perp$ if $S_{i-1} = \epsilon$.

**Proof.** We prove the claim by induction on $i$.

**Base case.** We prove the claim for $i = 1$. Recall that at $C_0$, since no operation has been linearized, the equivalent sequential stack is empty. Recall also that at $C_0$ it holds that $top = -1$. If $op_1$ is a push operation, the claim holds trivially. Let then $op_1$ be a pop operation. We consider two cases.

First, assume that $op_1$ is the first operation for which $s_s$ executes Algorithm 3. In that case, when $s_s$ checks the condition of the if clause of line 7 for $op_1$, it evaluates to true and NACK is sent to the client that invoked $op_1$. By inspection of the client pseudocode in Algorithm 5 (lines 22 - 23),
we see that when a pop operation receives NACK from \( s \), it returns \( \bot \) to the client. Thus, the claim holds.

Assume next that \( op_1 \) is not the first operation for which \( s \) executes Algorithm 3. Then, by Observation 2, at the point where the `if` condition of line 7 is evaluated by \( s \) for \( op_1 \), it will hold that \( top\_key > -1 \). Therefore, \( op_1 \) is not linearized at the execution of line 8. Thus, by the way linearization points are assigned, \( op_1 \) is linearized either at the execution of line 10 by \( s \) or at an even later configuration. By assumption, \( op_1 \) is the first operation to be linearized. This means that there is no linearization point for some push operation that is placed in a configuration preceding the execution of line 10 by \( s \) for \( op_1 \). Then, by definition, if \( op_1 \) is linearized at a configuration later than this, then it is linearized together with the push operation whose value \( op_1 \) returns. Then, however, \( op_1 \) is not the first operation to be linearized – a contradiction. Therefore, \( op_1 \) is linearized at the execution of line 8 by \( s \) and the claim holds.

**Hypothesis.** Fix any \( i, i > 0 \) and assume that the claim holds for all \( C_j, j \leq i \).

**Induction step.** We prove that the claim also holds at \( C_{i+1} \). If \( op_{i+1} \) is a push operation, the claim holds trivially. Let then \( op_{i+1} \) be a pop operation. We proceed by case analysis.

First, assume that \( op_{i+1} \) is linearized after the execution of line 8 by \( s \). This implies that in the configuration in which \( s \) evaluates the `if` condition of line 7, it evaluates to `true`. By Observation 3, this means that for each push operation that has been linearized up to that configuration, there has been a matching pop operation that has been linearized as well. It follows that \( S_i \) is empty and that the claim holds.

Next, assume that \( op_{i+1} \) is linearized in the configuration right after the execution of line 10 by \( s \). By definition, this means that \( op_{i+1} \) removes an element from the directory that has been inserted into the directory by a push operation \( op_j, j \leq i \), which has been linearized before the execution of this line, due to the way linearization points are assigned. We distinguish two cases.

First assume that \( op_i \) is a push operation and assume that \( k_i \) is the value of \( top\_key \) that it has received by \( s \), i.e., \( op_i \) inserts into the directory an element with key \( k_i \). Since \( op_i \) is linearized before the execution of line 10 by \( s \) for \( op_{i+1} \) and by Observation 2, we have that at the end of the execution of the instance of Algorithm 3 by \( s \) for \( op_i \), it holds that \( top\_key = k_i \). Inspection of Algorithm 3 shows that a pop operation that follows a push operation receives the same value of \( top\_key \) as the one that was sent to the push operation. Therefore, if no further instance of Algorithm 3 is executed for some other operation by \( s \) after it executes it for \( op_i \) and before it executes it for \( op_{i+1} \), then the claim follows straightforwardly. Assume now that between \( C_i \) and \( C_{i+1} \), more instances of Algorithm 3 are executed by \( s \) for other operations. Let \( op' \) be that out of those operations for which Algorithm 3 is executed last before \( C_{i+1} \) and assume that it is a push. Let \( k' \)
be the value of top_key at the end of this instance of Algorithm 3. Then, at $C_{i+1}$, $s_s$ sends $k'$ to the client that invoked $op_{i+1}$. Then this client attempts to remove from the directory an element with key $k'$. However, since there is no further operation linearized between $C_i$ and $C_{i+1}$, this element is not in the directory at $C_{i+1}$. Thus, the push operation that inserts in the directory the value which $op_{i+1}$ removes, is linearized after $C_{i+1}$ – a contradiction to the definition of linearization points. If $op'$ is a pop operation and it receives $k'$ as the value of top_key from $s_s$, then $op_{i+1}$ receives $k' - 1$ as value of top_key. Then, $op_{i+1}$ attempts to remove from the directory an element with key $k' - 1$. Let $op''$ be the push operation that inserts an element with this key. If $op''$ is linearized after $C_{i+1}$, once more we arrive at a contradiction. If $op''$ is linearized before $C_{i+1}$, then by the induction hypothesis, implies that each of the pop operations between $C_i$ and $C_{i+1}$ removes the top-most element of the sequential stack. Thus, at $C_{i+1}$, the element inserted by $op''$ is the top-most one and the claim holds.

Finally, assume that $op_{i+1}$ is linearized right after the linearization point of that push operation $op'$ whose value it removes from the directory. In this case, since no further operation is linearized between $op_{i+1}$ and $op'$, this means that the value inserted by $op'$ is indeed the top-most of $S_i$ when it is removed by $op_{i+1}$ and the claim holds.

From the above lemmas we have the following.

**Theorem 5.** The directory-based distributed stack implementation is linearizable.

### 4.3 Directory-Based Queue

The directory-based distributed queue implementation follows similar ideas as those of the directory-based stack implementation of Section 4.2. To implement a queue, $s_s$ maintains two counters, head and tail, which store the key associated with the first and the last, respectively, element in the queue. A client $c$ sends an enqueue (dequeue) request to $s_s$ to obtain a key $k$. Then, it uses $k$ as the input argument to $\text{DirInsert}(\text{DirDelete})$. When $s_s$ receives an enqueue (dequeue) request from $c$, it sends the value stored in tail (head) to $c$ and increments tail (head). In case of a dequeue request on an empty queue (i.e. if head = tail), $s_s$ sends NACK to $c$ without changing head.

#### 4.3.1 Algorithm Description

The synchronizer, described in Algorithm 6, receives, processes and responds to clients’ messages. The messages it may receive correspond to enqueue and dequeue requests. If the server receives an ENQ message (line 3), it sends to the client a message containing the current value of tail_key and increments
Algorithm 6 Events triggered in the synchronizer of the directory-based queue.

1 int head_key = 0, tail_key = 0;
2
3 a message \( \langle \text{op}, \text{cid} \rangle \) is received:
4 if (op == ENQ) {
5 send(cid, tail_key);
6 tail_key++;
7 } else if (op == DEQ) {
8 if (head_key == tail_key) {
9 send(cid, NACK);
10 } else {
11 send(cid, head_key);
12 head_key++;
13 }
14 }

When a DEQ message is received, \( s_s \) first checks if the values of head_key and tail_key are the same (line 7). If they are, the queue is empty, therefore \( s_s \) responds to the client with a NACK message (line 8). Otherwise, the directory still has elements stored, so the synchronizer sends the current value of head_key to \( c \) (line 10) and then increments head_key by one (line 11).

In order to perform an enqueue or dequeue operation, a client calls ClientEnqueue() or ClientDequeue(), respectively. ClientEnqueue(), the code of which is presented in Algorithm 7, performs similar steps as those presented in Algorithm 4 of the directory-based stack.

Algorithm 7 Enqueue operation for a client of the directory-based queue.

12 void ClientEnqueue(int cid, Data data) {
13 sid = get the server id;
14 send(sid, \( \langle \text{ENQ}, \text{cid} \rangle \));
15 tail_key = receive(sid);
16 DirInsert(tail_key, data);
17 }

ClientDequeue(), presented in Algorithm 8, works in a similar way as Algorithm 7. The client sends a DEQ message to \( s_s \). If \( s_s \) responds with NACK (line 21), the queue is empty and the client returns \( \bot \). If \( s_s \) responds with the value of head_key, the client uses this value as the key of the element to remove from the directory (line 24). DirDelete returns \( \bot \) if the insertion of the key to be deleted is still pending. When DirDelete() returns the data
associated with head_key, ClientDequeue() terminates.

**Algorithm 8** Dequeue operation for a client of the directory-based queue.

```c
17 Data ClientDequeue(int cid) {
18     sid = get the server id;
19     send((sid, ⟨DEQ, cid⟩));
20     head_key = receive(sid);
21     if(head_key == NACK)
22         return ⊥;
23     do {
24         status = DirDelete(head_key);
25     } while (status == ⊥);
26     return status;
27 }
```

### 4.3.2 Proof of Correctness

Let α be an execution of the directory-based queue implementation. We assign linearization points to enqueue and dequeue operations in α as follows: The linearization point of an enqueue operation op is placed in the configuration resulting from the execution of line 4 for op by ss. The linearization point of a dequeue operation op is placed in the configuration resulting from the execution of either line 8 or line 10 for op (whichever is executed) by ss.

**Lemma 6.** The linearization point of an enqueue (dequeue) operation op is placed within its execution interval.

**Proof.** Assume that op is an enqueue operation and let c be the client that invokes it. After the invocation of op, c sends a message to ss (line 15) and awaits a response from it. Recall that routine receive() (line 15) blocks until a message is received. The linearization point of op is placed at the configuration resulting from the execution of line 4 for op by ss. This line is executed after the request by c is received, i.e. after c invokes ClientEnqueue. Furthermore, it is executed before c receives the response by the server and thus, before ClientEnqueue returns. Therefore, the linearization point is included in the execution interval of enqueue.

The argumentation regarding dequeue operations is similar. □

Denote by L the sequence of operations which have been assigned linearization points in α in the order determined by their linearization points. Let $C_i$ be the configuration in which the $i$-th operation $op_i$ of $L$ is linearized; denote by $C_0$ the initial configuration. Denote by $α_i$, the prefix of $α$ which ends with $C_i$ and let $L_i$ be the prefix of $L$ up until the operation that is linearized at $C_i$. We denote by $head_i$ the value of the local variable head_key of ss at configuration $C_i$, and by $tail_i$ the value of the local variable tail_key
of $s_a$ at $C_i$. By the pseudocode, we have that the initial values of $\text{tail\_key}$ and $\text{head\_key}$ are 0; therefore, we consider that $\text{head}_0 = \text{tail}_0 = 0$.

Denote by $Q_i$ the sequential queue that results if the operations of $L_i$ are applied sequentially to an initially empty queue. Let the size of $Q_i$ (i.e., the number of elements contained in $Q_i$) at $C_i$ be $d_i$. Denote by $s\text{l}_j^i$ the $j$-th element of $Q_i$, $1 \leq j \leq d_i$. Each element of $Q_i$ is a pair of type $\langle \text{key, data} \rangle$ where for the $i-th$ enqueue operation, $\text{key} = i - 1$. Consider a sequence of elements $S$. If $e$ is the first element of $S$, we denote by $S \setminus e$ the suffix of $S$ that results by removing only element $e$ from the first position of $S$. If $e$ is an element not included in $S$, we denote by $S' = S \cdot e$ the sequence that results by appending element $e$ to the end of $S$.

Notice that since only $s_a$ executes Algorithm 6, we have the following.

**Observation 7.** Instances of Algorithm 6 are executed sequentially, i.e., their execution does not overlap.

By inspection of Algorithm 6, we have that for some instance of it, either lines 3-5, or lines 7-8, or lines 9-11 are executed. Then, by the way linearization points are assigned, and by Observation 7, we have the following.

**Observation 8.** Given two configurations $C_i$, $C_{i+1}$, $i \geq 0$, in $\alpha$, there is at most one step in the execution interval between $C_i$ and $C_{i+1}$ that modifies either $\text{head\_key}$ or $\text{tail\_key}$.

**Lemma 9.** For each integer $i \geq 1$, the following hold at $C_i$:

1. If $i > 1$ and $\text{op}_{i-1}$ is an enqueue operation, then $\text{tail}_i = \text{tail}_{i-1} + 1$ and $\text{head}_i = \text{head}_{i-1}$; if $i = 1$, then $\text{tail}_i = \text{tail}_{i-1}$.

2. If $i > 1$, $\text{head}_{i-1} \neq \text{tail}_{i-1}$ and $\text{op}_{i-1}$ is a dequeue operation, then $\text{head}_i = \text{head}_{i-1} + 1$ and $\text{tail}_i = \text{tail}_{i-1}$; if $i = 1$, then $\text{head}_i = \text{head}_{i-1}$.

**Proof.** Fix any $i \geq 1$. The linearization point of $\text{op}_i$ may be placed at the configuration resulting from the execution of line 4, line 8 or line 10, whichever is executed by $s_a$ for it. By inspection of the pseudocode, we have that in either case, the execution of neither of these lines, nor the ones preceding it in the instance of Algorithm 6 executed for $\text{op}_i$, modify $\text{tail\_key}$ or $\text{head\_key}$. Notice also that because of Observation 7 no process other than $s_a$ modifies neither $\text{tail\_key}$ nor $\text{head\_key}$ between $C_{i-1}$ and $C_i$.

We proceed by case analysis. First, consider the case where $i = 1$. Recall that $\text{tail}_0 = \text{head}_0 = 0$. Because of the preceding argument, $\text{tail}_1 = \text{tail}_0 = 0$ and $\text{head}_1 = \text{head}_0 = 0$. Thus, the claims hold.

Next, consider the case where $i > 1$. Let $\text{op}_{i-1}$ be an enqueue operation. By the pseudocode (line 5), $\text{tail\_key}$ is incremented after the linearization point of $\text{op}_{i-1}$, i.e., between configurations $C_{i-1}$ and $C_i$. Thus, $\text{tail}_i = \text{tail}_{i-1} + 1$. The value of $\text{head\_key}$ is not modified by enqueue operations (lines 3-5), therefore $\text{head}_i = \text{head}_{i-1}$.

Now let $\text{op}_{i-1}$ be a dequeue operation that is linearized at the execution of line 8. By inspection of the pseudocode (line 7), this occurs only in
case $head_{i-1} = tail_{i-1}$. By the pseudocode (lines 7-8) and by Observation 7, it follows that in this case $head_{key}$ is not modified in the execution interval between $C_{i-1}$ and $C_i$. Therefore, $head_i = head_{i-1}$. Since a dequeue operation does not modify $tail_{key}$, it also holds that $tail_i = tail_{i-1}$.

Finally, let $op_{i-1}$ be a dequeue operation that is linearized at the execution of line 10. By the pseudocode, line 11 and by Observation 7, $head_{key}$ is incremented by 1 after the linearization point of $op_{i-1}$, i.e. between configurations $C_{i-1}$ and $C_i$. Thus, $head_i = head_{i-1} + 1$. The value of $tail_{key}$ is not modified by dequeue operations (lines 7-11), therefore $tail_i = tail_{i-1}$.

We denote the $key$ field of the $\langle data, key \rangle$ pair that comprises some element $sl_{1}^j$, $0 < j \leq d_i$, of $Q_i$ by $sl_{1}^j.key$. By inspection of the pseudocode (lines 3-5), we see that, when $op_i$ is an enqueue operation, $tail_i$ is sent by $s_s$ to the client $c$ that invoked $op_i$. By further inspection of the pseudocode (lines 15-16), we see that $c$ uses $tail_i$ as the $key$ field of the element it enqueues. When $op_i$ is a dequeue operation, by inspection of the pseudocode (lines 7-8), we have that when $head_{key} = tail_{key}$, $s_s$ sends NACK to $c$, and that when $c$ receives NACK, it does not enqueue any element and instead, returns $\bot$(lines 21-22). When $head_{key} \neq tail_{key}$, $s_s$ sends $head_i$ to $c$ (lines 9-11) and $c$ uses $head_i$ as the $key$ field in order to determine which element to dequeue (lines 24-26).

**Observation 10.** If $op_i$ is an enqueue operation, it inserts a pair with $key = tail_{i}$ into the directory. If $op_i$ is a dequeue operation then, if $head_i \neq tail_i$, it removes a pair with $key = head_i$ from the directory; if $head_i = tail_i$, it does not remove any pair from the directory.

**Lemma 11.** At $C_i$, $i \geq 1$, the following hold:

1. If $op_i$ is an enqueue operation, then $tail_i = sl_{i}^{d_i}.key$.
2. If $op_i$ is a dequeue operation, then if $Q_{i-1} \neq \epsilon$, $head_i = sl_{i-1}^1.key$. If $Q_{i-1} = \epsilon$, then $head_i = tail_i$.

**Proof.** We prove the claims by induction.

**Base case.** We prove the claim for $i = 1$. Consider the case where $op_1$ is an enqueue operation. Then, $d_1 = 1$ and $Q_1$ contains only the pair $\langle 0, data \rangle$. By Observation 7, it is the first operation in $\alpha$ for which an instance of Algorithm 6 is executed by $s_s$. Therefore, by Lemma 9, $tail_1 = tail_0 = 0$. Thus, $tail_1 = sl_{1}^1.key$

Now consider the case where $op_1$ is a dequeue operation. By Observation 7, $op_1$ is the first operation in $\alpha$ for which an instance of Algorithm 6 is executed by $s_s$. Notice that then, $Q_1 = \epsilon$. Therefore, by Lemma 9, $head_1 = head_0 = 0$. By the same reasoning, $tail_1 = tail_0 = 0$. Thus, $head_1 = tail_1$, so Claim 2 holds.

**Hypothesis.** Fix any $i$, $i > 0$ and assume that the lemma holds at $C_i$.

**Induction step.** We prove that the claims also hold at $C_{i+1}$. Assume that $op_{i+1}$ is an enqueue operation. We examine two cases. First,
Consider that $op_i$ is an enqueue operation as well. By the induction hypothesis, $sl^{d_i}_i.key = tail_i$. By Lemma 9, we have that $tail_{i+1} = tail_i + 1$. By Observation 10, we have that the client $c$ that initiated $op_{i+1}$ inserts a pair with key $= tail_{i+1} = tail_i + 1$ into the directory. By definition, $sl^{d_{i+1}}_{i+1}.key = sl^{d_i}_i.key + 1$. Thus, $sl^{d_{i+1}}_{i+1}.key = tail_i + 1$, and Claim 1 holds.

Next, consider that $op_i$ is a dequeue operation. By Lemma 9, dequeue operations do not modify tail.key. This lemma further implies that $tail_{i+1} = tail_j + 1$, where $op_j$ is the last enqueue operation preceding $op_{i+1}$ in $L_{i+1}$. By definition and by Observation 10, $op_j$ enqueues a pair with key $= tail_j$ to $Q_j$. Furthermore, by definition of $op_j$, all other operations in $L_{i+1}$ that have a linearization point between that of $op_j$ and $op_{i+1}$, are dequeue operations. Therefore, no further element is appended to $Q$ between $C_j$ and $C_{i+1}$, i.e. $sl^{d_j}_j = sl^{d_i}_i$. Notice that $sl^{d_j}_j.key = tail_j$. By Observation 10, $c$ inserts a pair with key $= tail_{i+1}$ into the directory and by definition, $sl^{d_{i+1}}_{i+1}.key = sl^{d_i}_i.key + 1$. Thus, since $tail_{i+1} = tail_j + 1$, it follows that $sl^{d_{i+1}}_{i+1}.key = tail_j + 1 = sl^{d_j}_j.key + 1 = sl^{d_i}_i.key + 1$, and Claim 1 holds.

Now let $op_{i+1}$ be a dequeue operation. Again we examine two cases. First, consider that $op_i$ is a dequeue operation as well. By the induction hypothesis, $op_i$ dequeues an element with key $sl_{i-1}.key = head_i$. Since Claim 2 holds at $C_i$, $sl_i.key = head_i + 1$. By Lemma 9, $head_{i+1} = head_i + 1$. Thus, $op_{i+1}$ removes from the sequential queue the element with key equal to $head_i + 1$. Since Claims 1 and 2 hold at $C_i$ by the induction hypothesis, we have that this element is $sl_1$, i.e. Claim 2 also holds at $C_{i+1}$.

Next consider that $op_i$ is an enqueue operation. By Lemma 9, enqueue operations do not modify head.key. This lemma further implies that $head_{i+1} = head_j + 1$, where $op_j$ is the last dequeue operation preceding $op_{i+1}$ in $L_{i+1}$. By definition and by Observation 10, $op_j$ dequeues a pair with key $= tail_j$ from $Q_{j-1}$. Furthermore, by definition of $op_j$, all other operations in $L_{i+1}$ that have a linearization point between that of $op_j$ and $op_{i+1}$, are enqueue operations. Therefore, no further element is removed from $Q$ between $C_j$ and $C_{i+1}$, i.e. $sl^{d_j}_j = sl^{d_i}_i$. Notice that $sl^{d_j}_j.key = head_j$. By Observation 10, $c$ removes a pair with key $= head_{i+1}$ from the directory and by definition, $sl^{d_{i+1}}_{i+1}.key = sl^{d_i}_i.key + 1$. Thus, since $head_{i+1} = head_j + 1$, it follows that $sl^{d_{i+1}}_{i+1}.key = head_j + 1 = sl^{d_j}_j.key + 1 = sl^{d_i}_i.key + 1$, and Claim 2 holds.

By Lemma 9 and by inspection of the pseudocode, we have that at $C_i$, $i > 0$, the value of tail.key indicates the number of enqueue operations on $Q_i$ that have been linearized in $\alpha_i$, and the value of head.key indicates the number of successful dequeue operations (i.e. dequeue operations that do not return $\bot$) on $Q_i$ that have been linearized in $\alpha_i$. Thus, the following corollary holds.

**Corollary 12.** $Q_i = \epsilon$ if and only if $head_i = tail_i$. 

28
Lemma 13. If \( op_i \) is a dequeue operation, then it returns the value of the field data of \( sl_{i-1}^1 \) or \( \perp \) if \( Q_{i-1} = \epsilon \).

Proof Sketch. Consider the case where \( Q_{i-1} \neq \epsilon \). By definition of \( Q_i \), we have that \( Q_i = Q_{i-1} \setminus \{sl_{i-1}^1\} \). Let \( op_j \) be the enqueue operation that is linearized before \( op_i \) and inserts an element with key head\(_i\) to the queue. Notice by the pseudocode, line 24, that the parameter of \texttt{DirDelete} is head\(_i\). By the semantics of \texttt{DirDelete}, if at the point that the instance of \texttt{DirDelete} is executed in the do - while loop of lines 24-26 for \( op_i \), the instance of \texttt{DirInsert} of \( op_j \) has not yet returned, then \texttt{DirDelete} returns \( \langle \perp, - \rangle \).

By Lemma 11, and since head\(_i\) key is not modified by the execution of line 10 by the server, head\(_i\) is the key of the first pair \( sl_{i-1}^1 \) in \( Q_{i-1} \). Therefore, when \texttt{DirDelete} returns a status \( \neq \perp \), it holds that it returns the data field of \( sl_{i-1}^1 \), the first element in \( Q_{i-1} \), as the return value of \( op_i \), i.e. the claim holds.

Now consider the case where \( Q_{i-1} = \epsilon \). Since, by Corollary 12, when this is the case, head\(_i\) = tail\(_i\), NACK is sent to the client that invoked \( op_i \) and, by inspection of the pseudocode, \( op_i \) returns \( \perp \), i.e. the claim holds. \( \square \)

From the above lemmas we have the following:

Theorem 14. The directory-based queue implementation is linearizable.

4.3.3 Queues with Special Functionality

Synchronous queue. A synchronous queue \( Q_S \) is an implementation of the queue data type (see Section 2.1). Instead of storing elements, a synchronous queue matches instances of \texttt{Dequeue()} with instances of \texttt{Enqueue()} operations. Thus, if \( op_e \) is an instance of an \texttt{Enqueue}(\( x, Q_S \)) operation and \( op_d \) an instance of a \texttt{Dequeue}(\( Q_S \)) such that \( op_d \) returns the element \( x \) enqueued by \( op_e \), then a synchronous queue ensures that the execution intervals of \( op_e \) and \( op_d \) are overlapping.

In order to derive a distributed synchronous queue from the directory-based queue proposed here, \( s_s \) must respond to a dequeue request with the value of head and increment head, even if head = tail. Moreover, \( s_s \) must use a local queue to store active enqueue requests together with the keys it has assigned to them (notice that there can be no more such requests that the number of clients); \( s_s \) must send the key \( k \) for each such enqueue request to the client that initiated it, at the time that head becomes equal to \( k \). In this way, the execution interval of an enqueue operation for element \( e \) overlaps that of the dequeue operation which gets \( e \) as a response, as specified by the semantics of a synchronous queue.

Delay queue. A delay queue \( Q_D \) implements the queue abstract data type. Each element \( e \) of a delay queue is associated with a delay value \( t_e \) that represents the time that \( e \) must remain in the queue before it can be removed from it. Thus, an \texttt{Enqueue}(\( e, t_e, Q_D \)) inserts an element \( e \) with
time-out value $t_e$ to $Q_S$. $\text{Dequeue}(Q_D)$ returns the element $e$ residing at the head of $Q_D$ if $t_e$ has expired and blocks (or performs spinning) if this is not the case. Notice that this implementation can easily be provided by associating each element inserted in the directory with a time-out value. We also have to change the way that the directory works so that it takes into consideration the delay of each element before removing it.

4.4 Directory-Based Double-Ended Queue (Deque)

The implementation of the directory-based deque follows similar principles as the stack and queue implementations. In order to implement a deque, $s_s$ also maintains two counters, $\text{head}$ and $\text{tail}$, which store the key associated with the first and the last, respectively, element in the deque. However, in this case, counters $\text{head}$ and $\text{tail}$ may store negative integers and are incremented or decremented based on the operation to be performed.

4.4.1 Algorithm Description

Event-driven pseudocode for the synchronizer $s_s$ is presented in Algorithm 9; $s_s$ now performs a combination of actions presented for the synchronizers of the stack and the queue implementations (Algorithms 3 and 6).

The synchronizer $s_s$ has two counters, $\text{head}$ and $\text{tail}$ (line 1), that store the key associated with the first and the last, respectively, element in the deque. The $\text{head}$ is modified when operations targeting the front are received by $s_s$ and the $\text{tail}$ is modified when operations targeting the back are received by $s_s$. Because each endpoint of a deque behaves as a stack, the actions for enqueuing and dequeuing are similar as in Algorithm 3.

Upon a message receipt, if $s_s$ receives a request $\text{ENQ}_T$ (line 4) it increments $\text{tail}$ by one (line 5), and then sends the current value of $\text{tail}$ to the client (line 6). The client uses the value that $s_s$ sends to it, as the key for the data to insert in the directory. Likewise, if $s_s$ receives a request $\text{ENQ}_H$ (line 18), it sends the current value of $\text{head}$ to the client (line 19), and then decrements $\text{head}$ by one (line 20).

When a message of type $\text{DEQ}_T$ arrives (line 8), $s_s$ first checks whether the deque is empty (line 9). If this is so, $s_s$ sends a $\text{NACK}$ to the client (line 10). Otherwise, the synchronizer repeatedly calls $\text{DirDelete}(\text{tail})$ to remove the element corresponding to a key equal to the value of $\text{tail}$ from the directory (line 13), and then decrements $\text{tail}$ (line 15). Finally, $s_s$ sends the data to the client (line 16). The synchronizer performs similar actions for a $\text{DEQ}_H$ message, but instead of decrementing the $\text{tail}$, it increments the $\text{head}$ (line 26).

The code for the clients operations for enqueue, is presented in Algorithm 10. For enqueuing to the back of the deque, the client sends an $\text{ENQ}_T$ message to $s_s$ and blocks waiting for its response. When it receives the unique key from $s_s$, the client is free to insert the element lazily. For
Algorithm 9 Events triggered in the synchronizer of the directory-based deque.

```c
int head_key = 0, tail_key = 0;

a message \langle op, cid \rangle is received:
switch (op) {
  case ENQ.T:
    tail_key ++;
    send(cid, tail_key);
    break;
  case DEQ.T:
    if (tail_key == head_key) {
      send(cid, NACK);
    } else {
      do {
        status = DirDelete(tail_key);
      } while (status == ⊥);
      tail_key --;
      send(cid, status);
    }
    break;
  case ENQ.H:
    send(cid, head_key);
    head_key --;
    break;
  case DEQ.H:
    if (tail_key == head_key) {
      send(cid, NACK);
    } else {
      head_key ++;
      do {
        status = DirDelete(head_key);
      } while (status == ⊥);
      send(cid, status);
    }
    break;
}
```

enqueueing to the front of the deque, the client sends an ENQ.H message and performs the same actions as for enqueueing to the back.

The client code for dequeue to the front and dequeue to the back, is presented in Algorithm 11. For dequeueing to the back of the deque, the client sends an DEQ.T message to s and blocks waiting for its response. The synchronizer performs the dequeue itself and sends back the response. For dequeueing to the front of the deque, the client sends an DEQ.H message and performs the same actions as for enqueue to the back.
Algorithm 10 Enqueue operations for a client of the directory-based deque.

```c
void EnqueueTail(int cid, Data data) {
    sid = get the synchronizer id;
    send(sid, ⟨ENQ_T, cid⟩);
    key = receive(sid);
    DirInsert(key, data);
}

void EnqueueHead(int cid, Data data) {
    sid = get the synchronizer id;
    send(sid, ⟨ENQ_H, cid⟩);
    key = receive(sid);
    DirInsert(key, data);
}
```

Algorithm 11 Dequeue operation for a client of the directory-based deque.

```c
Data DequeueTail(int cid) {
    sid = get the synchronizer id;
    send(sid, ⟨DEQ_T, cid⟩);
    status = receive(sid);
    return status;
}

Data DequeueHead(int cid) {
    sid = get the synchronizer id;
    send(sid, ⟨DEQ_H, cid⟩);
    status = receive(sid);
    return status;
}
```

4.4.2 Proof of Correctness

Let $\alpha$ be an execution of the directory-based deque implementation. We assign linearization points to enqueue and dequeue operations in $\alpha$ as follows:

The linearization point of an enqueue back operation $op$ is placed in the configuration resulting from the execution of line 6 for $op$ by $s_s$. The linearization point of a dequeue back operation $op$ is placed in the configuration resulting from the execution of either line 10 or line 16 for $op$ (whichever is executed) by $s_s$. The linearization point of an enqueue front operation $op$ is placed in the configuration resulting from the execution of line 19 for $op$ by $s_s$. The linearization point of a dequeue front operation $op$ is placed in the configuration resulting from the execution of either line 24 or line 30 for $op$ (whichever is executed) by $s_s$.

Lemma 15. The linearization point of an enqueue (dequeue) operation $op$ executed by client $c$ is placed within its execution interval.
Proof. Assume that \( op \) is an enqueue front (back) operation and let \( c \) be the client that invokes it. After the invocation of \( op \), \( c \) sends a message to \( s_s \) (line 41) and awaits a response from it. Recall that routine \( \text{receive()} \) (line 42) blocks until a message is received. The linearization point of \( op \) is placed at the configuration resulting from the execution of line 19 for \( op \) by \( s_s \). This line is executed after the request by \( c \) is received, i.e. after \( c \) invokes \( \text{EnqueueHead} \) (\( \text{EnqueueTail} \)). Furthermore, it is executed before \( c \) receives the response by the server and thus, before \( \text{EnqueueHead} \) (\( \text{EnqueueTail} \)) returns. Therefore, the linearization point is included in the execution interval of enqueue front (back).

The argumentation regarding dequeue front (back) operations is similar.

Denote by \( L \) the sequence of operations which have been assigned linearization points in \( \alpha \) in the order determined by their linearization points. Let \( C_i \) be the configuration in which the \( i \)-th operation \( op_i \) of \( L \) is linearized; denote by \( C_0 \) the initial configuration. Denote by \( \alpha_i \), the prefix of \( \alpha \) which ends with \( C_i \) and let \( L_i \) be the prefix of \( L \) up until the operation that is linearized at \( C_i \). We denote by \( \text{head}_i \) the value of the local variable \( \text{head}_\text{key} \) of \( s_s \) at configuration \( C_i \), and by \( \text{tail}_i \) the value of the local variable \( \text{tail}_\text{key} \) of \( s_s \) at \( C_i \). By the pseudocode, we have that the initial values of \( \text{tail}_\text{key} \) and \( \text{head}_\text{key} \) are 0; therefore, we consider that \( \text{head}_0 = \text{tail}_0 = 0 \).

By analogous reasoning as the one followed in the case of the directory-based queue, inspection of the pseudocode leads to the following observations.

Observation 16. Instances of Algorithm 9 are executed sequentially, i.e. their execution does not overlap.

Observation 17. Given two configurations \( C_i, C_{i+1}, i \geq 0 \), in \( \alpha \), there is at most one step in the execution interval between \( C_i \) and \( C_{i+1} \) that modifies \( \text{tail}_\text{key} \).

Denote by \( D_i \) the sequential deque that results if the operations of \( L_i \) are applied sequentially to an initially empty queue. Let the size of \( D_i \) (i.e. the number of elements contained in \( D_i \)) at \( C_i \) be \( d_i \). Denote by \( sl^j_i \) the \( j \)-th element of \( D_i \), \( 1 \leq j \leq d_i \). Each element of \( D_i \) is a pair of type \( \langle \text{key}, \text{data} \rangle \) where the elements from the bottommost to the topmost are assigned integer keys as follows: Let \( f_i \) be the key of element \( sl^1_i \) and \( l_i \) be the key of element \( sl^{d_i}_i \) in some configuration \( C_i \). We denote the \text{key} field of the \( \langle \text{data}, \text{key} \rangle \) pair that comprises some element \( sl^j_i \), \( 1 \leq j \leq d_i \), of \( D_i \) by \( sl^j_i.\text{key} \). Then, if \( d_i > 1 \), \( sl^{j+1}_i.\text{key} = sl^j_i.\text{key} + 1, 1 \leq j \leq d_i \). We consider that if \( op_1 \) is an enqueue front operation, then \( f_1 = l_1 = 0 \), while if it is an enqueue back operation, then \( f_1 = l_1 = 1 \). Notice that \( l_i - f_i + 1 = d_i \).

Consider a sequence of elements \( S \). If \( e \) is the first element of \( S \), we denote by \( S \setminus e \) the suffix of \( S \) that results by removing only element \( e \) from
the first position of \( S \). If \( e \) is the last element of \( S \), we denote by \( S \setminus e \) the prefix of \( S \) that results by removing only element \( e \) from the last position of \( S \). If \( e \) is an element not included in \( S \), we denote by \( S' = S \cdot e \) the sequence that results by appending element \( e \) to the end of \( S \), and by \( S'' = e \cdot S \) the sequence that results by prefixing \( S \) with element \( e \).

**Lemma 18.** For each integer \( i \geq 1 \), the following hold at \( C_i \):

1. If \( op_i \) is an enqueue back operation, then \( tail_i = tail_{i-1} + 1 \).
2. If \( op_i \) is a dequeue back operation, then it holds that \( tail_i = tail_{i-1} - 1 \) if \( tail_{i-1} \neq head_{i-1} \); otherwise, \( tail_i = tail_{i-1} \).

**Proof.** Fix any \( i \geq 1 \). If \( op_i \) is an enqueue back operation, the linearization point of \( op_i \) is placed at the configuration resulting from the execution of line 6. By inspection of the pseudocode (lines 5-6), we have that in the instance of Algorithm 9 executed for \( op_i \), \( tail \) is incremented before it is sent to the client \( c \) that invoked \( op_i \). By Observations 16 and 17, this is the only increment that occurs on \( tail \) between \( C_{i-1} \) and \( C_i \). Thus, Claim 1 holds.

If \( op_i \) is a dequeue back operation, the linearization point of \( op_i \) is placed either at the configuration resulting from the execution of line 10 or at the configuration resulting from the execution of line 16. Let \( op_{i-1} \) be a dequeue back operation that is linearized at the execution of line 10. By inspection of the pseudocode (line 9), this occurs only in case \( head_{i-1} = tail_{i-1} \). Since the execution of this line does not modify \( tail \), \( tail_i = tail_{i-1} \) and Claim 2 holds.

Now let \( op_i \) be a dequeue back operation that is linearized at the configuration resulting from the execution of line 16. By the pseudocode (lines 15-16) and by Observation 7, it follows that the execution of line 15 is the only step in which \( tail \) is modified in the execution interval between \( C_{i-1} \) and \( C_i \). Since line 16 is executed, it holds that the condition of the if clause of line 9 evaluates to \textit{false}, i.e. it holds that \( head_{i-1} \neq tail_{i-1} \). Furthermore, because of the execution of line 15, \( tail_i = tail_{i-1} - 1 \). Thus, Claim 2 holds. \( \square \)

**Lemma 19.** For each integer \( i \geq 1 \), the following hold at \( C_i \):

1. In case \( op_i \) is an enqueue front operation, then, if \( i > 1 \) and \( op_{i-1} \) is an enqueue front operation, it holds that \( head_i = head_{i-1} - 1 \); otherwise \( head_i = head_{i-1} \).
2. In case \( op_i \) is a dequeue front operation, then, if \( head_{i-1} \neq tail_{i-1} \), \( i > 1 \), and \( op_{i-1} \) is not an enqueue front operation, it holds that \( head_i = head_{i-1} + 1 \); otherwise \( head_i = head_{i-1} \).

**Proof.** Fix any \( i \geq 1 \). Let \( op_i \) be an enqueue front operation. If \( i = 1 \), then by inspection of the pseudocode, we have that \( head \) is not modified before the execution of line 19. Since \( head_0 = 0 \) and the execution of line 19 does not modify \( head \), it follows that \( head_1 = 0 = head_0 \) and Claim 1 holds. Now let \( i > 1 \). By inspection of the pseudocode and by
Observation 16 we have that \textit{head\_key} is not modified by enqueue back and dequeue back operations. By the pseudocode, Observation 16 and the way linearization points are assigned, we have that although \textit{head\_key} is modified by dequeue front operations only before the configuration in which the operation is linearized, it is modified by enqueue front operations in the step (line 20) right after the configuration in which an enqueue front operation is linearized. Therefore, if \textit{op\_i−1} is an enqueue front operation, then \textit{head\_key} is decremented once (line 20) in the execution interval between \textit{C\_i−1} and \textit{C\_i}. Thus, if \textit{op\_i−1} is an enqueue front operation, then \textit{head\_i} = \textit{head\_i−1} + 1, while if \textit{op\_i−1} is any other type of operation, \textit{head\_i} = \textit{head\_i−1}. Thus, Claim 1 holds.

Now let \textit{op\_i} be a dequeue operation. If \textit{i} = 1, then by inspection of the pseudocode, we have that \textit{head\_key} is not modified before the execution of line 15. By the pseudocode and by Observation 16, \textit{tail\_key} is not modified as well before the execution of line 15. Thus, \textit{head\_0} = 0 = \textit{tail\_0} and the if condition of line 9 evaluates to \textbf{true}. Then, \textit{op\_1} is linearized in the configuration resulting from the execution of line 24. Notice that the execution of this line does not modify \textit{head\_key}. It follows that \textit{head\_1} = 0 = \textit{head\_0} and that Claim 2 holds.

Now let \textit{i} > 1. The linearization point of \textit{op\_i} may be placed at the configuration resulting from the execution of line 24 or line 30, whichever is executed by \textit{s\_i} for it. Let the linearization point be placed in the configuration resulting from the execution of line 24. In that case, \textit{head\_i−1} = \textit{tail\_i−1}. Notice that the execution of that line does not modify \textit{head\_key}. Therefore, \textit{head\_i} = \textit{head\_i−1}, and Claim 2 holds. Now let the linearization point be placed in the configuration resulting from the execution of line 30. In case \textit{op\_i−1} is an enqueue back or dequeue tail operation, \textit{head\_key} is not modified by it. Therefore, since line 26 is executed before line 30, \textit{head\_i} = \textit{head\_i+1} + 1 and Claim 2 holds. The same also holds if \textit{op\_i−1} is a dequeue front operation. If \textit{op\_i−1} is an enqueue front operation, then by inspection of the pseudocode (line 20), we have that \textit{head\_key} is decremented in the step following the configuration in which \textit{op\_i−1} is linearized. Therefore, in this case and by Observation 16, \textit{head\_key} is decremented and then incremented once in the execution interval between \textit{C\_i−1} and \textit{C\_i}. This in turn implies that \textit{head\_i} = \textit{head\_i−1} − 1 + 1 = \textit{head\_i−1} and Claim 2 holds.

Recall that \textit{sl\_i.key} = \textit{f\_i} + \textit{j} − 1 or \textit{sl\_i.key} = \textit{l\_i} − \textit{j} + 1. By inspection of the pseudocode (lines 19/6), we see that, when \textit{op\_i} is an enqueue front/back operation, \textit{head\_i}/\textit{tail\_i} is sent by \textit{s\_i} to the client \textit{c} that invoked \textit{op\_i}. By further inspection of the pseudocode (lines 42-43/37-37), we see that \textit{c} uses \textit{head\_i}/\textit{tail\_i} as the \textit{key} field of the element it enqueues, i.e. uses it as argument for auxiliary function \textbf{DirInsert()}/\textbf{DirDelete()}. When \textit{op\_i} is a dequeue front/back operation, by inspection of the pseudocode (lines 24/10), we have that when \textit{head\_key} = \textit{tail\_key}, \textit{s\_i} sends \textbf{NACK} to \textit{c}, and that when \textit{c} receives \textbf{NACK}, it does not enqueue any element and instead, returns \textbf{⊥}.
When head.key $\neq$ tail.key, $s_\alpha$ uses head$_i$/(tail$_i$ + 1) as the key field in order to determine which element to dequeue (lines 28/13). Then, the following observation holds.

**Observation 20.** If op$_i$ is an enqueue back operation, it inserts a pair with key = tail$_i$ into the directory. If op$_i$ is a dequeue back operation, then, if head$_i$ $\neq$ tail$_i$, it removes a pair with key = tail$_i$ + 1 from the directory; if head$_i$ = tail$_i$, it does not remove any pair from the directory. If op$_i$ is an enqueue front operation, it inserts a pair with key = head$_i$ into the directory. If op$_i$ is a dequeue front operation then, if head$_i$ $\neq$ tail$_i$, it removes a pair with key = head$_i$ from the directory; if head$_i$ = tail$_i$, it does not remove any pair from the directory.

**Lemma 21.** At $C_i$, $i \geq 1$, the following hold:

1. If op$_i$ is an enqueue back operation, then tail$_i$ = sl$_i^{d_i}$.key.
2. If op$_i$ is a dequeue back operation, then if $D_{i-1} \neq \epsilon$, tail$_i$ = sl$_{i-1}^{d_{i-1}}$.key.
   
   If $D_{i-1} = \epsilon$, then head$_i$ = tail$_i$.
3. If op$_i$ is an enqueue front operation, then head$_i$ = sl$_i^1$.key.
4. If op$_i$ is a dequeue front operation, then if $D_{i-1} \neq \epsilon$, head$_i$ = sl$_{i-1}^1$.key.
   
   If $D_{i-1} = \epsilon$, then head$_i$ = tail$_i$.

**Proof.** We prove the claims by induction.

**Base case.** We prove the claim for $i = 1$.

Consider the case where op$_1$ is an enqueue back operation. Then, $d_1 = 1$ and by definition, $D_1$ contains only the pair $\langle 1, data \rangle$. By Observation 16, it is the first operation in $\alpha$ for which an instance of Algorithm 9 is executed by $s_\alpha$. Therefore, by Lemma 18, tail$_1$ = tail$_0$ + 1 = 1. Thus, tail$_1$ = sl$_1^1$.key and Claim 1 holds.

Next, consider the case where op$_1$ is a dequeue back operation. By Observation 16, op$_1$ is the first operation in $\alpha$ for which an instance of Algorithm 9 is executed by $s_\alpha$. Notice that then, $Q_1 = \epsilon$. Therefore, by Lemma 18, tail$_1$ = tail$_0$ = 0. Since head.key is not modified by dequeue back operations, head$_1$ = head$_0$ = 0. Thus, head$_1$ = tail$_1$, so Claim 2 holds.

Next, consider the case where op$_1$ is an enqueue front operation. Again, by definition, $d_1 = 1$ and $D_1$ contains only the pair $\langle 0, data \rangle$. By Observation 16, it is the first operation in $\alpha$ for which an instance of Algorithm 9 is executed by $s_\alpha$. Therefore, by Lemma 19, head$_1$ = head$_0$ = 0. Thus, head$_1$ = sl$_1^1$.key and Claim 3 holds.

Finally, consider the case where op$_1$ is a dequeue front operation. By Observation 16, op$_1$ is the first operation in $\alpha$ for which an instance of Algorithm 9 is executed by $s_\alpha$. Notice that then, $Q_1 = \epsilon$. Therefore, by Lemma 19, head$_1$ = head$_0$ = 0. Since tail.key is not modified by dequeue front operations, tail$_1$ = tail$_0$ = 0. Thus, head$_1$ = tail$_1$ and Claim 4 holds.

**Hypothesis.** Fix any $i$, $i > 0$ and assume that the lemma holds at $C_i$.

**Induction step.** We prove that the claims also hold at $C_{i+1}$. Assume that op$_{i+1}$ is an enqueue back operation. By the induction hypothesis, if op$_i$
is an enqueue back operation, then \(sl_i^d.key = tail_i = l_i\). Similarly, if \(op_i\) is a dequeue back operation, then by the induction hypothesis, \(sl_{i-1}^d.key = tail_i\). Since the dequeue back operation removes the last element in \(D_{i-1}\), it follows that the last element \(sl_i^d\) of \(D_i\) is \(sl_{i-1}^d\). Thus, here also, tail_i = sl_i^d.key = l_i. Notice that enqueue front and dequeue front operations do not modify tail.key. Since these types of operation do not affect the back of the sequential dequeue, it still holds that tail_i = sl_i^d.key = l_i. Since \(op_{i+1}\) is an enqueue back operation, by Lemma 18, we have that tail_{i+1} = tail_i + 1.

By Observation 20, we have that the client \(c\) that initiated \(op_{i+1}\) inserts a pair with \(key = tail_{i+1} = tail_i + 1\) into the directory. By definition, \(sl_{i+1}^d.key = sl_i^d.key + 1\). Thus, \(sl_{i+1}^d.key = tail_i + 1\), and Claim 1 holds.

Now assume that \(op_{i+1}\) is a dequeue back operation. We examine two cases. First, let \(D_i \neq \epsilon\). By Lemma 18, it then holds that tail_{i+1} = tail_i - 1.

By Observation 20, we have that a pair with \(key = tail_{i+1} = tail_i - 1\) is removed from the directory. By definition, we have that \(D_{i+1} = D_i \setminus b\ sl_i^d\). Also by definition, we have that \(sl_i^b.key = sl_i^d - 1.key + 1\). Because of \(op_{i+1}\), \(sl_{i+1}^d = sl_{i+1}^d - 1\). Since tail_{i+1} = tail_i - 1, Claim 2 holds. Now let \(D_i = \epsilon\). In this case, \(op_{i+1}\) cannot have any effect on the state of the deque.

By inspection of the pseudocode, this corresponds to the operation being linearized in the configuration resulting from the execution of line 10. Notice that in order for this to be the case, the \(if\) condition of line 9 must evaluate to \(true\). This occurs if head_i = tail_i, thus Claim 2 holds.

Next assume that \(op_{i+1}\) is an enqueue front operation. By the induction hypothesis, if \(op_i\) is an enqueue front operation, then \(sl_i^f.key = head_i = f_i\).

By Lemma 19, it holds then that head_{i+1} = head_i - 1. Since \(op_{i+1}\) is an enqueue front operation, it prepends an element to \(D_i\) and therefore, \(sl_i^f = sl_{i+1}^f\). By definition of \(D_{i+1}\), \(sl_{i+1}^f.key = sl_{i+1}^f - 1.key - 1\). Since \(sl_{i+1}^f.key = head_i\), \(sl_{i+1}^f.key = head_{i+1}\) and Claim 3 holds.

On the other hand, if \(op_i\) is a dequeue front operation, then by the induction hypothesis, \(sl_i^l.key = head_i\). By Lemma 19, it also follows that in this case, head_{i+1} = head_i. Notice that by definition, \(op_i\) removes element \(sl_i^l\) from \(D_{i-1}\). Then, for element \(sl_i^l\) of \(D_i\), by definition, \(sl_i^l.key = sl_{i-1}^l.key + 1\). This means that \(sl_{i-1}^l.key = head_i = sl_i^l.key - 1\). Since head_{i+1} = head_i, Claim 3 holds.

Notice that enqueue back and dequeue back operations do not modify head.key.

Finally, assume that \(op_{i+1}\) is a dequeue front operation. We examine two cases. First, let \(D_i \neq \epsilon\). By Lemma 19, it then holds that head_{i+1} = head_i.

By Observation 20, we have that a pair with \(key = head_{i+1} = head_i\) is removed from the directory. By definition, we have that \(D_{i+1} = D_i \setminus b\ sl_i^l\). Also by definition, we have that \(sl_i^l.key = sl_i^l - 1\). Because of \(op_{i+1}\), \(sl_i^l = sl_{i+1}^l\). Since head_{i+1} = head_i, Claim 4 holds. Now let \(D_i = \epsilon\). In this case, \(op_{i+1}\) cannot have any effect on the state of the deque. By inspection of the pseudocode, this corresponds to the operation being linearized in the
configuration resulting from the execution of line 24. Notice that in order for this to be the case, the if condition of line 23 must evaluate to true. This occurs if head$_i$ = tail$_i$, thus Claim 4 holds.

From the above lemma, we have the following corollary.

**Corollary 22.** $D_i = \epsilon$ if and only if head$_i$ = tail$_i$.

**Lemma 23.** If op$_i$ is a dequeue back operation, then it returns the value of the field data of $sl^{d_{i-1}}$ or $\bot$ if $D_{i-1} = \epsilon$.

**Proof.** Consider the case where $D_{i-1} \neq \epsilon$. By definition of $D_i$, we have that $D_i = D_{i-1} \setminus b_{sl_i}^{d_{i-1}}$. Let op$_j$ be the enqueue operation that is linearized before op$_i$ and inserts an element with key tail$_i + 1$ to the queue. Notice by the pseudocode (lines 12-16), that the parameter of DirDelete is tail$_i + 1$. By the semantics of DirDelete, if at the point that the instance of DirDelete is executed in the do - while loop of lines 12-13 for op$_i$, the instance of DirInsert of op$_j$ has not yet returned, then DirDelete returns $\langle \bot, - \rangle$.

By Lemma 18, tail$_i + 1$ is the key of the last pair $sl^{d_{i-1}}_{i-1}$ in $D_{i-1}$. Therefore, when DirDelete returns a status $\neq \bot$, it holds that it returns the data field of $sl^{d_{i-1}}_{i-1}$, the last element in $D_{i-1}$. Notice that this value is sent to the client $c$ that invoked op$_i$ (line 16) and that $c$ uses this value as the return value of op$_i$ (lines 48-49). Thus, the claim holds.

Now consider the case where $D_{i-1} = \epsilon$. Since, by Corollary 22, when this is the case, head$_i$ = tail$_i$, NACK is sent $c$ and, by inspection of the pseudocode, op$_i$ returns $\bot$, i.e. the claim holds.

In a similar fashion, we can prove the following.

**Lemma 24.** If op$_i$ is a dequeue front operation, then it returns the value of the field data of $sl^1_{i-1}$ or $\bot$ if $D_{i-1} = \epsilon$.

From the above lemmas we have the following:

**Theorem 25.** The directory-based deque implementation is linearizable.

### 4.5 Hierarchical approach, Elimination, and Combining.

In this section, we outline how the hierarchical approach, described in Section 3.1, is applied to the directory-based designs.

Each island master $m_i$ performs the necessary communication between the clients of its island and $s_s$. In the stack implementation, each island master applies elimination before communicating with $s_s$. To further reduce communication with $s_s$, $m_i$ applies a technique known as combining [24]. In the case of stack, once elimination has been applied, there is only one type of requests that must be sent to the synchronizer; for all these requests, $m_i$
sends just one message containing their number \( f \) and their type to the synchronizer. In case of push operations, this method allows the synchronizer to directly increment \( \text{top} \) by \( f \) and respond to \( m_i \) with the value \( g \) that \( \text{top} \) had before the increment. Once \( m_i \) receives \( g \), it informs the clients (which initiated these requests) that the keys for their requests are \( g, g+1, \ldots, g+f-1 \). In the case of queue, each message of \( m_i \) to \( s_s \) contains two counters counting the number of active enqueue and dequeue requests from clients of island \( i \). When \( s_s \) receives such a message it responds with a message containing the current values of \( \text{tail} \) and \( \text{head} \). It then increments \( \text{tail} \) and \( \text{head} \) by the value of the first and second counter, respectively. Server \( m_i \) assigns unique keys to active enqueue and dequeue operations, based on the value of \( \text{tail} \) and \( \text{head} \) it receives, in a way similar as in stacks. Combining can be used for deques (in addition to elimination) in ways similar to those described above.
Chapter 5

Token-based Stacks, Queues, and Deques

We start with an informal description of the token-based technique that we present in this section. We assume that the servers are numbered from 0 to $\text{NS} - 1$ and form a logical ring. Each server has allocated a chunk of memory (e.g. one or a few pages) of a predetermined size, where it stores elements of the implemented DS. A DS implementation employs (at least) one token which identifies the server $s_t$, called the token server, at the memory chunk of which newly inserted elements are stored. (A second token is needed in cases of queues and deques.) When the chunk of memory allocated by the token server becomes full, the token server gives up its role and appoints another (e.g. the next) server as the new token server. A client remembers the server that served its last request and submits the next request it initiates to that server; so, each response to a client contains the id of the server that served the client’s request. Servers that do not have the token for handling a request, forward the request to subsequent servers; this is done until the request reaches the appropriate token server. A server allocates a new (additional) chunk of memory every time the token reaches it (after having completed one more round of the ring) and gives up the token when this chunk becomes full.

Section 5.1 presents the details of the token-based distributed stack. The token-based queue implementation appears in Section 5.2. Section 5.3 provides the token-based deque. We start by presenting static versions of the implementations, i.e. versions in which the total memory allocated for the data structure is predetermined during an execution and once it is exhausted the data structure becomes full and no more insertions of elements can occur. We then describe in Section 5.5, how to take dynamic versions of the data structures from their static analogs.
5.1 Token-Based Stack

To implement a distributed stack, each server uses its allocated memory chunk to maintain a local stack, `lstack`. Initially, $s_t$ is the server with id 0. To perform a push (or pop), a client $c$ sends a push (or pop) request to the server that has served $c$’s last request (or, initially, to server 0) and awaits for a response. If this server is not the current token server at the time that it receives the request, it forwards the request to its next or previous server, depending on whether its local stack is full or empty, respectively. This is repeated until the request reaches the server $s_t$ that has the token which pushes the new element onto its local stack and sends an ACK to $c$. If $s_t$’s local stack does not have free space to accommodate the new element, it sends the push request of $c$, together with an indication that it gives up its token, to the next server. POP is treated by $s_t$ in a similar way.

5.1.1 Algorithm Description

Initially the elements are stored in the memory space allocated by server $s_0$, the first server in the ring. At this point, $s_0$ is the token server; the token server manages the top of the stack. Once the memory chunk of the token server becomes full, the token server notifies the next server ($s_1$) in the ring to become the new token server.

The pseudocode for the server is presented in Algorithm 12. Each server $s$, apart from a local stack (`lstack`), maintains also a local variable `token` which identifies whether $s$ is the token server. The messages that are transmitted during the execution are of type `PUSH` and `POP`, which are sent from clients that want to perform the mentioned operation to the servers, or are forwarded from any server towards the token server. Each message has four fields: (1) `op` with the operation to be performed, (2) `data`, containing data in case of `ENQ` and ⊥ otherwise, (3) `id` that contains the id of the sender and (4) a one-bit flag `tk` which is set to `TOKEN` only when a forwarded message denotes also a token transition.

If the message is of type `PUSH` (line 6), $s$ first checks whether the message contains a token transition. If `tk` is marked with `TOKEN`, $s$ changes the `token` variable to contain its id (line 7). If $s$ is not a token server, it just forwards the message to the next server (line 9). Otherwise, it checks if there is free space in `lstack` to store the new request (line 11). If there is such space, the server pushes the data to the stack, and sends back an ACK to the client. In this implementation, the `push()` function (line 12) does not need to return any value, since the check for memory space has already been performed by the server on line 11, hence `push()` is always successful.

If $s$ does not have any free space, it must notify the next server to become the new token server. More specifically, if $s$ is not the server with id $NS - 1$ (line 14), it forwards to the next server the `PUSH` message it received from the client, after setting the `tk` field to `TOKEN` (line 16). On the other hand,
Algorithm 12 Events triggered in a server of the token-based stack.

```c
LocalStack lstack = ∅;
int my_sid; /* each server has a unique id */
int token = 0;

a message ⟨op, data, id, tk⟩ is received:
switch (op) {
  case PUSH:
    if (tk == TOKEN) token = my_sid;
    if (token ≠ my_sid) {
      send(token, ⟨op, data, id, tk⟩);
      break;
    }
    if (!IsFull(lstack)) {
      push(lstack, data);
      send(id, ⟨ACK, my_sid⟩);
    } else if (my_sid ≠ NS-1) {
      token = find_next_server(my_sid);
      send(token, ⟨op, data, id, TOKEN⟩);
    } else /* It’s the last server in the order, thus the stack is full */
    send(id, ⟨NACK, my_sid⟩);
    break;
  case POP:
    if (tk == TOKEN) token = my_sid;
    if (token ≠ my_sid) {
      send(token, ⟨op, data, id, tk⟩);
      break;
    }
    if (!IsEmpty(lstack)) {
      data = pop(lstack);
      send(id, ⟨data, my_sid⟩);
    } else if (my_sid ≠ 0) {
      token = find_previous_server(my_sid);
      send(token, ⟨op, data, id, TOKEN⟩);
    } else /* It’s the first server in the order, thus the stack is empty */
    send(id, ⟨NACK, my_sid⟩);
    break;
}
```

if s is the server with id NS – 1, all previous servers have no memory space available to store a stack element. In this case, s sends back to the client a message NACK(line 18).

If the message is of type POP (line 20) similar actions take place: s checks
whether the message contains a token transition and if its true, it changes its local variable token appropriately. Then s checks if it is the token server (line 22). If not, it just forwards the message towards the server it considers as the token server (line 23). If s is the token server, it checks if its local stack is empty (line 25). If it is not empty, the pop operation can be executed normally. At the end of the operation, s sends to the client the data of the previous top element (line 27). In case of an empty local stack, if s is not s0 (line 28), it forwards to the previous server the client’s POP message, after setting the tk field to TOKEN (line 30). On the other hand, if the server that received the POP request is s0 (id == 0), then all the servers have empty stacks and the server sends back to the client a NACK message (line 32).

**Algorithm 13** Push operation for a client of the token-based stack.

```
34 sid = 0; /* the client stores the id of the first server with id=0. */
35 Data ClientPush(int cid, Data data) {
36   send(sid, ⟨PUSH, data, cid, ⊥⟩);
37   ⟨status, sid⟩ = receive();
38   return status;
}
```

The clients execute the operations push and pop, by calling the functions `ClientPush()` and `ClientPop()`, respectively. Each of these functions sends a message to the server. Initially, the clients forward their requests to s0. Because the server that maintains the top element might change though, the clients update the sid variable through a lazy mechanism. When a client c wants to perform an operation, it sends a request to the server with id equal to the value of sid (lines 34 and 39). If the message was sent to an incorrect server, it is forwarded by the servers till it reaches the server that holds the token. That server is going to respond with the status value of the operation and with the its id. This way, c updates the variable sid.

During the execution of the `ClientPush()` function, described in Algorithm 13, the client sends a PUSH message to the server with id sid (line 36). It then, waits for its response (line 37). When the client receives the response, it updates the sid variable (line 37) and returns the status. The status is either ACK for a successful push, or NACK for a full stack.

The `ClientPop()` function operates in a similar fashion. The client sends a POP message to the server with id sid (line 41). It then, waits for its response (line 42). The server responds with a NACK (for empty queue), or with the value of the top element (otherwise). The server also forwards its id, which is stored in client’s variable sid. The client finally, returns the status value and terminates.
Algorithm 14 Pop operation for a client of the token-based stack.

```plaintext
sid = 0; /* the client stores the id of the first server with id=0. */

Data ClientPop(int cid) {
    send(sid,⟨POP,⊥,cid,⊥⟩);
    ⟨status,sid⟩ = receive();
    return status;
}
```

5.1.2 Proof of Correctness

Let $\alpha$ be an execution of the token-based stack algorithm presented in Algorithms 12, 13, and 14. Let $op$ be any operation in $\alpha$. We assign a linearization point to $op$ by considering the following cases:

- $op$ is a push operation. Let $s_t$ be the token server that responds to the client that initiated $op$ (i.e., the receive of line 37 in the execution of $op$ receives a message from $s_t$). If $op$ returns ACK, the linearization point is placed at the configuration resulting from the execution of line 13 by $s_t$ for $op$. Otherwise, the linearization point of $op$ is placed at the configuration resulting from the execution of line 18 by $s_t$ for $op$.

- $op$ is a pop operation. Let $s_t$ be the token server that responds to the client that initiated $op$ (line 42). If the operation returns NACK, the linearization point of $op$ is placed at the configuration resulting from the execution of line 32 by $s_t$ for $op$. Otherwise, the linearization point of $op$ is placed at the configuration resulting from the execution of line 27 by $s_t$ for $op$.

Denote by $L$ the sequence of operations (which have been assigned linearization points) in the order determined by their linearization points.

**Lemma 26.** The linearization point of a push (pop) operation $op$ is placed in its execution interval.

*Proof Sketch.* Assume that $op$ is a push operation and let $c$ be the client that invokes it. After the invocation of $op$, $c$ sends a message to some server $s$ and awaits a response. Recall that routine receive() (line 37) blocks until a message is received. The linearization point of $op$ is placed either in the configuration resulting from the execution of line 13 by $s_t$ for $op$, where $s_t$ is the token server in this configuration, or in the configuration resulting from the execution of line 18 by $s_t$ for $op$.

Either of these lines is executed after the request by $c$ is received, i.e., after $c$ invokes ClientPush. Furthermore, they are executed before $c$ receives the response by $s_t$ and thus, before ClientPush returns. Therefore, the linearization point is inside the execution interval of push.

The argumentation regarding pop operations is analogous. □

Each server maintains a local variable $token$ with initial value 0 (initially,
the server with id equal to 0 is the token server). Whenever some server $s_i$ receives a TOKEN message, i.e. a message with its $tk$ field equal to TOKEN (line 7), the value of token is set to $i$. By inspection of the pseudocode, it follows that the value of token is set to the id of the next server if the local stack of $s_i$ is full (line 15); then, a TOKEN message is sent to the next server (line 16). Moreover, the value of token is set to the id of the previous server if the local stack $lstack$ of $s_i$ is empty (line 28); then, a TOKEN message is sent to the previous server (lines 29-30). (Unless the server is $s_0$ in which case a NACK is sent to the client (line 32 but no TOKEN message to any server.) Thus, the following observation holds.

Observation 27. At each configuration in $\alpha$, there is at most one server $s_i$ for which the local variable token has the value $i$.

At each configuration $C$, the server $s_i$ whose token variable is equal to $i$ is referred to as the token server at $C$.

Observation 28. A TOKEN message is sent from a server with id $i$, $0 \leq i < NS - 1$, to a server with id $i+1$ only if the local stack of server $i$ is full. A TOKEN message is sent from a server with id $i$, $0 < i \leq NS - 1$, to a server with id $i-1$ only when the local stack of server $i$ is empty.

By the pseudocode, namely the if clause of line 8 and the if clause of line 22, the following observation holds.

Observation 29. Whenever a server $s_i$ performs push and pop operations on its local stack (lines 12 and 26), it holds that its local variable token is equal to $i$.

Let $C_i$ be the configuration at which the $i$-th operation $op_i$ of $L$ is linearized. Denote by $\alpha_i$, the prefix of $\alpha$ which ends with $C_i$ and let $L_i$ be the prefix of $L$ up until the operation that is linearized at $C_i$. Denote by $S_i$ the sequence of values that a sequential stack contains after applying the sequence of operations in $L_i$, in order, starting from an empty stack; let $S_0 = \epsilon$, i.e. $S_0$ is the empty sequence.

Lemma 30. For each $i$, $i \geq 0$, if $s_{ki}$ is the token server at $C_i$ and $ls_i^j$ are the contents of the local stack of server $j$, $0 \leq j \leq k_i$, at $C_i$, then it holds that $S_i = ls_i^0 \cdot ls_i^1 \cdot \ldots \cdot ls_i^{k_i}$ at $C_i$.

Proof. We prove the claim by induction on $i$. The claim holds trivially for $i = 0$. Fix any $i \geq 0$ and assume that at $C_i$, it holds that $S_i = ls_i^0 \cdot ls_i^1 \cdot \ldots \cdot ls_i^{k_i}$. We show that the claim holds for $i + 1$.

We first assume that $op_{i+1}$ is a push operation initiated by some client $c$. Assume first that $s_{ki} = s_{ki+1}$. Then, by induction hypothesis, $S_i = ls_i^0 \cdot \ldots \cdot ls_i^{k_i}$. In case the local stack of $s_{ki}$ is not full, $s_{ki}$ pushes the value $v_{i+1}$ of field data of the request onto its local stack and responds to $c$. Since no other change occurs to the local stacks of $s_0, \ldots, s_{ki}$ from $C_i$ to $C_{i+1}$, at
therefore, it pushes the value $v_{C_s}$ there is a head token server. To implement a queue, two tokens are employed: at each point in time, $5.2$ Token-Based Queue

$0$ plays the role of both $s_c$ and $s_h$. Each server $s_r$, other than $s_t$ ($s_h$), that receives a request (directly) from a client $c$, it forwards the request to the next server to ensure that it will either reach the appropriate token server or return back to $s_r$ (after traversing all servers). Servers $s_t$ and $s_h$ work in a way similar as server $s_t$ in stacks.

To prevent a request from being forwarded forever due to the completion of concurrent requests which may cause the token(s) to keep advancing, each server keeps track of the request that each client $c$ (directly) sends to it, in a client table (there can be only one such request per client). Server $s_t$ (and/or $s_h$) now reports the response to $s_r$, which forwards it to $c$. If $s_r$ receives a response for a request recorded in its client table, it deletes the request from the client table. If $s_r$ receives the token (stack, tail, or head), it serves each request (push and pop, enqueue, or dequeue, respectively) in its client array and records its response. If a request, from those included in $s_r$’s client array, reaches $s_r$ again, $s_r$ sends the response it has calculated for it to the client and removes it from its client array. Since the communication channels are FIFO, the implementations ensures that all requests, their responses, and the appropriate tokens, move from one server to the next, based on the

From the above lemmas and observations, we have the following.

**Theorem 31.** The token-based distributed stack implementation is linearizable. The time complexity and the communication complexity of each operation $op$ is $O(NS)$.

### 5.2 Token-Based Queue

To implement a queue, two tokens are employed: at each point in time, there is a head token server $s_h$ and a tail token server $s_t$. Initially, server 0 plays the role of both $s_h$ and $s_t$. Each server $s_r$, other than $s_t$ ($s_h$), that receives a request (directly) from a client $c$, it forwards the request to the next server to ensure that it will either reach the appropriate token server or return back to $s_r$ (after traversing all servers). Servers $s_t$ and $s_h$ work in a way similar as server $s_t$ in stacks.

To prevent a request from being forwarded forever due to the completion of concurrent requests which may cause the token(s) to keep advancing, each server keeps track of the request that each client $c$ (directly) sends to it, in a client table (there can be only one such request per client). Server $s_t$ (and/or $s_h$) now reports the response to $s_r$, which forwards it to $c$. If $s_r$ receives a response for a request recorded in its client table, it deletes the request from the client table. If $s_r$ receives the token (stack, tail, or head), it serves each request (push and pop, enqueue, or dequeue, respectively) in its client array and records its response. If a request, from those included in $s_r$’s client array, reaches $s_r$ again, $s_r$ sends the response it has calculated for it to the client and removes it from its client array. Since the communication channels are FIFO, the implementations ensures that all requests, their responses, and the appropriate tokens, move from one server to the next, based on the
servers’ ring order, until they reach their destination. This is necessary to argue that the technique ensures termination for each request.

5.2.1 Algorithm Description

The queue implementation follows similar ideas to those of the token-based distributed stack, presented in Section 5.1. However, the queue implementation employs two tokens, one for the queue’s tail and one for the queue’s head, called head token and tail token, respectively. The tokens for the global head and tail are initially held by $s_0$. However, they can be reassigned to other servers during the execution. If the local queue of the server that has the tail token becomes full, the token is forwarded to the next server. Similarly, if the local queue of the server that has the head token becomes empty, the head token is forwarded to the next server. If the appropriate token server receives the request and serves it, it sends an ACK message back to the server that initiated the forwarding. Then, the initial server responds to the client with an ACK message, which also includes the id of the server that currently holds the token.

The clients in their initial state store the id of $s_0$, which is the first server to hold the head and tail tokens. The clients keep track of the servers that hold either token in a lazy way. Specifically, a client updates its local variable (either $enq\_sid$ or $deq\_sid$ depending on whether its current active operation is an enqueue or a dequeue, respectively) with the id of the token server when it receives a server response.

In this scheme, a client request may be transmitted indefinitely from a server to the next without ever reaching the appropriate token server. This occurs if both the head and the tail tokens are forwarded indefinitely along the ring. Then, a continuous, never-ending race between a forwarded message and the appropriate token server may occur. To avoid this scenario, we do the following actions. When a server $s$ receives a client’s request $r$, if it does not have the appropriate token to serve it, it stores information about $r$ in a local array before it forwards it. Next time that the server receives the tail (head) token, it will serve all enqueue (dequeue) requests. Notice that since channels preserve the FIFO order and servers process messages in the order they arrive, the appropriate token will reach $s$ earlier than the $r$. When $s$ receives $r$, it has already processed it; however, it is then that $s$ sends the response for $r$ to the client.

In Algorithm 15, we present the local variables of a server. Each server $s$ holds its unique id $my\_sid$ and a local queue $lqueue$ that stores its part of the queue. Also keeps two boolean flag variables, ($has\Head$ and $has\Tail$), indicating whether $s$ has the head token or the tail token, and one more bit flag ($full\Queue$) indicating whether the queue is full. Finally, $s$ has a local array of size $n$, where $n$ is the maximum number of clients, used for storing all direct requests from clients (called $s$’s clients array). In their initial state, all servers have $full\Queue$ set to $false$ and their clients array.
Algorithm 15 Token-based queue server’s local variables.

```java
1 int my_sid;
2 LocalQueue lqueue = ∅;
3 LocalArray clients = ∅; /* Array of three values: <op, data, isServed> */
4 boolean fullQueue = false; /* True when tail and head are in the same server and tail is before head */
5 boolean hasHead; /* Initially hasHead and hasTail are true in server 0, and false in the rest */
6 boolean hasTail;
```

and local queue empty. Also, all servers apart from server 0, have both their flags hasHead and hasTail set to false, whereas in server 0, they are set to true, as described above.

The messages sent to a server $s_i$ are of type ENQ or DEQ, describing requests for enqueue or dequeue operations, respectively, sent by either a server or a client, and ACK or NACK sent by another server $s_j$ which executed a forwarded request, whose forwarding was initiated by $s_i$. The token transition is encapsulated in a message of type ENQ or DEQ. The messages have five fields: (1) $op$, which describes the type of the request (ENQ, DEQ, ACK or NACK), (2) $data$, which stores an element in case of ENQ, and ⊥ otherwise, (3) $cid$, which stores the id of the client that issued the request, (4) $sid$ which contains the id of the server if the message was sent by a server, and -1 otherwise, and (5) $tk$, which contains TAIL_TOKEN or HEAD_TOKEN in forwarded messages of type ENQ or DEQ, respectively, to indicate if an additional tail (or head, respectively) token transition occurs, and it is equal to ⊥ otherwise.

Event-driven pseudocode for the server is presented in Algorithm 16. When a server $s$ receives a message of type ENQ (line 13), it first checks if it contains a token transfer from another server (line 14). If it does, the server sets its token hasTail to true (line 15) and if it also had the head token from a previous round, it changes fullQueue flag to true as well (line 16). Then, $s$ serves all pending ENQ messages stored in its clients array (line 17).

Then, $s$ continues to execute the ENQ request. It checks first whether it has the tail token. If it does not (line 18), it finds the next server $s_{next}$ (line 19), to whom $s$ is going to forward the request. Afterwards, $s$ sends the received request to $s_{next}$ (lines 22 and 24) and if that request came directly from a client (line 20), $s$ updates its clients array storing in it information about this message (line 21).

If $s$ has the token, then it attempts to serve the request. If $s$ has remaining space in its local queue ($lqueue$), it enqueues the given data and informs the appropriate server with an ACK message (lines 25-30). If the implemented queue is full, $s$ sends a NACK message to the client (lines 31-35). In any remaining case, the server $s$ must give the tail token to the next server (line 36). So, $s$ forwards the ENQ message to $s_{next}$, after encapsulating in
the message the tail token (line 40). After releasing the tail token, \( s \) changes the values of its local variables (\( \text{hasTail} \) and \( \text{fullQueue} \)) to \text{false}.

In case a \text{DEQ} message is received (line 42), the actions performed by \( s \) are similar to those for \text{ENQ}. Server \( s \) checks whether the request message contains a token transition (line 43). If it does, the server sets its token \( \text{hasHead} \) to \text{true} (line 44). Then, \( s \) serves all pending \text{DEQ} messages stored in its \text{clients} array (line 45), and then attempts to serve the request. If \( s \)

---

**Algorithm 16** Events triggered in a server of the token-based queue.

```plaintext
7   a message \( \{\text{op, data, cid, sid, tk}\} \) is received:
8     if (!\text{clients[cid]} AND \text{clients[cid].isServed}) { /* If message has been served earlier. */
9         \text{send(cid, \{ACK, clients[cid].data, my_sid\});}
10        \text{clients[cid]} = \bot;
11     } else {
12         switch (\text{op}) {
13             case \text{ENQ}:
14                 if (tk == TAIL_TOKEN) {
15                     \text{hasTail} = \text{true};
16                     if (\text{hasHead}) \text{fullQueue} = \text{true};
17                     \text{ServeOldEnqueues();}
18                 } else {
19                     \text{nsid} = \text{find next server(\{my_sid\});}
20                     if (sid == -1) {
21                         \text{clients[cid]} = \{ENQ, data, false\};
22                         \text{send(nsid, \{ENQ, data, cid, my_sid, \bot\});}
23                     } else {
24                         \text{send(nsid, \{ENQ, data, cid, \bot\});}
25                 }
26             } else if (!\text{IsFull(lqueue))} { /* Server can enqueue. */
27                 \text{enqueue(lqueue, data);}
28                 if (sid == -1) {
29                     \text{send(cid, \{ACK, \bot, my_sid\});}
30                 } else {
31                     \text{send(sid, \{ACK, \bot, cid, my_sid, \bot\});}
32                 }
33             } else if (\text{fullQueue}) { /* Global Queue full */
34                 if (sid == -1) {
35                     \text{send(cid, \{NACK, \bot, my_sid\});}
36                 } else {
37                     \text{send(sid, \{NACK, \bot, cid, my_sid, \bot\});}
38                 }
39             } else {
40                 \text{nsid} = \text{find next server(\{my_sid\);}
41                 \text{fullQueue} = \text{false};
42                 \text{hasTail} = \text{false};
43                 \text{send(nsid, \{op, data, cid, my_sid, TAIL_TOKEN\});}
44         }
45         break;
```
case DEQ:
    if (tk == HEAD_TOKEN) {
        hasHead = true;
        ServeOldDeques();
    }
    if (!hasHead) {
        nsid = find_next_server(my_sid);
        if (sid == -1) { /* From client */
            clients[cid] = (DEQ, ⊥, false);
            send(nsid, ⟨DEQ, ⊥, cid, my_sid⟩);
        } else { /* From server */
            send(nsid, ⟨DEQ, ⊥, cid, ⊥⟩);
        }
    } else if (!IsEmpty(lqueue)) { /* Server can dequeue */
        data = dequeue(lqueue);
        if (sid == -1) { /* From client */
            send(cid, ⟨ACK, data, my_sid⟩);
        } else { /* From server */
            send(sid, ⟨ACK, data, cid, my_sid, ⊥⟩);
        }
    } else if (hasTail AND !fullQueue) { /* Queue is empty */
        if (sid == -1) { /* From client */
            send(cid, ⟨NACK, ⊥, my_sid⟩);
        } else { /* From server */
            send(sid, ⟨NACK, ⊥, cid, my_sid, ⊥⟩);
        }
    } else { /* Server moves the head token to the next server */
        nsid = find_next_server(my_sid);
        hasHead = false;
        send(nsid, ⟨op, ⊥, cid, my_sid, HEAD_TOKEN⟩);
    }
    break;
    case ACK:
    clients[cid] = ⊥;
    send(cid, ⟨ACK, data, sid⟩);
    break;
    case NACK:
    clients[cid] = ⊥;
    send(cid, ⟨NACK, ⊥, sid⟩);
    break;
}

does not hold the head token (line 46), it finds the next server $s_{next}$ in the ring (line 47), to whom $s$ is going to forward the request. Afterwards, $s$ sends the received request to $s_{next}$ (lines 50, 52) and if that request came directly from a client, $s$ updates its clients array storing in it information about this message (line 49).

If $s$ has the head token, it does the following actions. If its local queue ($lqueue$) is not empty (line 53), $s$ performs a dequeue on its local queue and
sends an ACK along with the dequeued data to the appropriate server. If \(s\) holds both head and tail tokens, but no other server has a queue element stored, and \(s\)'s queue is empty, it means that the global queue is empty (line 59). Thus, \(s\) sends a NACK message to the appropriate server. In the remaining cases, \(s\) must forward its head token (line 64). Server \(s\) finds the next server \(s_{\text{next}}\) (line 65), which is going to receive the forwarded message and the head token transition. Server \(s\) sets the message field \(tk\) to HEAD_TOKEN and sends the message (line 67). After releasing the head token, \(s\) sets the value of its local variable \(\text{hasHead}\) to false (line 66).

**Algorithm 17** Auxiliary functions for a server of the token-based queue.

```c
void ServeOldEnqueues(void) {
    if (!fullQueue) {
        for each cid such that clients[cid].op == ENQ {
            if (!IsFull(lqueue)) {
                enqueue(lqueue, clients[cid].data);
                clients[cid].isServed = true;
            }
        }
    }
}

void ServeOldDequeues(void) {
    for each cid such that clients[cid].op == DEQ {
        if (!IsEmpty(lqueue)) {
            clients[cid].data = dequeue(lqueue);
            clients[cid].isServed = true;
        }
    }
}
```

If \(s\) received a message of type ACK (line 69) or NACK (line 73), then \(s\) sets the entry \(cid\) of its clients array to ⊥ (lines 70, 74) and sends an ACK (line 71) or a NACK (line 75) to that client. The ACK and NACK messages a server \(s\) receives, are only sent by other servers and signify the result of the execution of a forwarded message sent by \(s\).

On lines 21 and 49, \(s\) stores the client request in its clients array when it does not hold the appropriate token. A request recorded in the clients array is removed from the array either when an ACK or NACK message is received for it (lines 70 and 74) or when the server receives again the request (after a round-trip on the ring) (lines 9-10). Thus, the server, upon any message receipt, first checks whether the message exists in its clients array and has already been served. In case of ENQ \(s\) answers with an ACK message, whereas in case of DEQ \(s\) answers with ACK and the dequeued data. Then, server \(s\) proceeds with the deletion of the entry in its clients array (lines 9, 10).

Functions ServeOldEnqueues() and ServeOldDequeues() are described in more detail in Algorithm 17. ServeOldEnqueues() (line 77) processes all ENQ requests stored in the clients array, if the local queue has space (line 80). Similarly, ServeOldDequeues() (line 84) processes all DEQ requests stored in the clients array, if the local queue is not empty (line 86).
The clients call the functions `ClientEnqueue()` and `ClientDequeue()`, presented in Algorithm 18, in order to perform one of these operations. In more detail, during enqueue, the client sends an `ENQ` message to the `enq_ssid` server, and waits for a response. When the client receives the response, it returns it. Likewise, in `ClientDequeue()` the client sends a `DEQ` message to server `deq_ssid` and blocks waiting for a response. When it receives the response, it returns it.

**Algorithm 18** Enqueue and Dequeue operations for a client of the token-based queue.

```plaintext
89 int enq_ssid = 0;
90 int deq_ssid = 0;

91 Data ClientEnqueue(int cid, Data data) {
92    send(enq_ssid, ⟨ENQ, data, cid, -1⟩);
93    ⟨status, ⊥, enq_ssid⟩ = receive(enq_ssid);
94    return status;
95 }

96 Data ClientDequeue(int cid) {
97    send(deq_ssid, ⟨DEQ, ⊥, cid⟩);
98    ⟨status, data, deq_ssid⟩ = receive(deq_ssid);
99    return data;
100 }
```

### 5.2.2 Proof of Correctness

Let $\alpha$ be an execution of the token-based queue algorithm presented in Algorithms 16, 17, and 18. Each server maintains local boolean variables `hasHead` and `hasTail`, with initial values `false`. Whenever some server $s_i$ receives a `TAIL_TOKEN` message, i.e. a message with its `tk` field equal to `TAIL_TOKEN` (line 14), the value of `hasTail` is set to `true` (line 15). By inspection of the pseudocode, it follows that the value of `hasTail` is set to `false` if the local queue of $s_i$ is full in $C$ (line 25, 36-39); then, a `TAIL_TOKEN` message is sent to the next server (line 40). The same holds for `hasHead` and `HEAD_TOKEN` messages, i.e. messages with their `tk` field equal to `HEAD_TOKEN`. Thus, the following observations holds.

**Observation 32.** At each configuration in $\alpha$, there is at most one server for which the local variable hasHead (hasTail) has the value true.

**Observation 33.** In some configuration $C$ of $\alpha$, TAIL_TOKEN message is sent from a server $s_j$, $0 \leq j < NS - 1$, to a server $s_k$, where $k = (j + 1) \mod NS$ only if the local queue of $s_j$ is full in $C$. Similarly, a HEAD_TOKEN message is sent from $s_j$ to $s_k$ only if the local queue of $s_j$ is empty in $C$. 

53
By inspection of the pseudocode, we see that a server performs an enqueue (dequeue) operation on its local queue \( lqueue \) either when executing line 26 (line 45) or when executing \texttt{ServeOldEnqueues} (\texttt{ServeOldDequeues}). Further inspection of the pseudocode (lines 14-17, lines 25-31, as well as lines 46-52, lines 53-59), shows that these lines are executed when \( hasTail = \texttt{true} \). Then, the following observation holds.

**Observation 34.** Whenever a server \( s_j \) performs an enqueue (dequeue) operation on its local queue, it holds that its local variable \( hasTail \) (\( hasHead \)) is equal to \texttt{true}.

By a straightforward induction, the following lemma can be shown.

**Lemma 35.** The mailbox of a client in any configuration of \( \alpha \) contains at most one incoming message.

If \( hasTail = \texttt{true} \) (\( hasHead = \texttt{true} \)) for some server \( s \) in some configuration \( C \), then we say that \( s \) has the tail (head) token. The server that has the tail token is referred to as \textit{tail token server}. The server that has the head token is referred to as \textit{head token server}.

Let \( op \) be any operation in \( \alpha \). We assign a linearization point to \( op \) by considering the following cases:

- If \( op \) is an enqueue operation for which a tail token server executes an instance of Algorithm 16, then it is linearized in the configuration resulting from the execution of either line 26, or line 81, or line 33, whichever is executed for \( op \) in that instance of Algorithm 16 by the tail token server.
- If \( op \) is a dequeue operation for which a head token server executes an instance of Algorithm 16, then it is linearized in the configuration resulting from the execution of either line 54, or line 87, or line 56, whichever is executed for \( op \) in that instance of Algorithm 16 by the head token server.

**Lemma 36.** The linearization point of an enqueue (dequeue) operation \( op \) is placed in its execution interval.

**Proof.** Assume that \( op \) is an enqueue operation and let \( c \) be the client that invokes it. After the invocation of \( op \), \( c \) sends a message to some server \( s \) (line 92) and awaits a response. Recall that routine \texttt{receive()} (line 93) blocks until a message is received. The linearization point of \( op \) is placed either in the configuration resulting from the execution of line 26 by \( s_t \) for \( op \), in the configuration resulting from the execution of line 33 by \( s_t \) for \( op \), or in the configuration resulting from the execution of line 81 by \( s_t \) for \( op \). Notice that either of these lines is executed after the request by \( c \) is received, i.e. after \( c \) invokes \texttt{ClientEnqueue}, and thus, after the execution interval of \( op \) starts.

By definition, the execution interval of \( op \) terminates in the configuration resulting from the execution of line 94. By inspection of the pseudocode,
this line is executed after line 93, i.e. after \( c \) receives a response by some server. In the following, we show that the linearization point of \( op \) occurs before this response is sent to \( c \).

Let \( s_j \) be the server that \( c \) initially sends the request for \( op \) to. By observation of the pseudocode, we see that \( c \) may either receive a response from \( s_j \) if \( s_j \) executes lines 28 or 33, or if \( s_j \) executes lines 70-71 or lines 74-75, or if \( s_j \) executes line 9. To arrive at a contradiction, assume that either of these lines is executed in \( \alpha \) before the configuration in which the linearization point of \( op \) is placed. Thus, a tail token server \( s_t \) executes lines 26, 81, or 33 in a configuration following the execution of lines 28, or 33, or 70-71 or 74-75, or line 9 by \( s_j \). Since the algorithm is event-driven, inspection of the pseudocode shows that in order for a tail token server to execute these lines, it must receive a message containing the request for \( op \) either from a client or from another server.

Assume first that a tail token server executes the algorithm after receiving a message containing a request for \( op \) from a client. This is a contradiction, since, on one hand, \( c \) blocks until receiving a response, and thus, does not send further messages requesting \( op \) or any other operation, and since \( op \) terminates after \( c \) receives the response by \( s_j \), and on the other hand, any other request from any other client concerns a different operation \( op' \).

Assume next that a tail token server executes the algorithm after receiving a message containing the request for \( op \) from some other server. This is also a contradiction since inspection of the pseudocode shows that after \( s_j \) executes either of the lines that sends a response to \( c \), it sends no further message to some other server and instead, terminates the execution of that instance of the algorithm.

The argumentation regarding dequeue operations is analogous.

Denote by \( L \) the sequence of operations which have been assigned linearization points in \( \alpha \) in the order determined by their linearization points. Let \( C_i \) be the configuration at which the \( i \)-th operation \( op_i \) of \( L \) is linearized. Denote by \( \alpha_i \), the prefix of \( \alpha \) which ends with \( C_i \) and let \( L_i \) be the prefix of \( L \) up until the operation that is linearized at \( C_i \). Denote by \( Q_i \) the sequence of values that a sequential queue contains after applying the sequence of operations in \( L_i \), in order, starting from an empty queue; let \( Q_0 = \epsilon \), i.e. \( Q_0 \) is the empty sequence. In the following, we denote by \( s_t \), the tail token server at \( C_i \) and by \( s_h \), the head token server at \( C_i \).

**Lemma 37.** For each \( i, i \geq 0 \), if \( lq^j_i \) are the contents of the local queue of server \( s_j \) at \( C_i \), \( h_i \leq j \leq t_i \), at \( C_i \), then it holds that \( Q_i = lq^h_i \cdot lq^{h_i+1}_i \cdot \ldots \cdot lq^t_i \) at \( C_i \).

**Proof.** We prove the claim by induction on \( i \). The claim holds trivially at \( i = 0 \).

Fix any \( i \geq 0 \) and assume that at \( C_i \), it holds that \( Q_i = lq^h_i \cdot lq^{h_i+1}_i \cdot \ldots \cdot lq^t_i \). We show that the claim holds for \( i + 1 \).
First, assume that \( op_{t+1} \) is an enqueue operation by client \( c \). Furthermore, distinguish the following two cases:

- Assume that \( t_i = t_{i+1} \). Then, by the induction hypothesis, \( Q_i = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \). In case the local queue of \( s_t \) is not full, \( s_t \) enqueues the value \( v_{i+1} \) of the data field of the request for \( op_{t+1} \) in the local queue (line 26 or line 31). Notice that, by Observation 34 changes on the local queues of servers occur only on token servers. Notice also that those changes occur only in a step that immediately precedes a configuration in which an operation is linearized. Thus, no further change occurs on the local queues of servers between \( C_i \) and \( C_{i+1} \), other than the enqueue on \( lq_i^{t_i} \). Then, it holds that \( Q_{i+1} = Q_i \cdot v_{i+1} = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot v_{i+1} = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot v_{i+1} = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot v_{i+1} \). If the head token server does not change between \( C_i \) and \( C_{i+1} \), then \( h_{i+1} = h_i \) and \( Q_{i+1} = lq_i^{h_i+1} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i+1} \) and the claim holds. If the head token server changes, i.e., if \( h_{i+1} \neq h_i \), then by Observation 33, \( lq_i^{h_i} = 0 \) and the claim holds again.

In case the local queue of \( s_t \) is full and since by assumption, \( s_t = s_{t+1} \), it follows by inspection of the pseudocode (line 31) and the definition of linearization points, that \( s_{t+1} = s_{h_{i+1}} \). In this case, \( s_{t+1} \) responds with a NACK to \( c \) and the local queue remains unchanged. Since no token server changes between \( C_i \) and \( C_{i+1} \), \( Q_{i+1} = Q_i = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} = lq_i^{h_i+1} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i+1} \) and the claim holds.

- Next, assume that \( t_i \neq t_{i+1} \). This implies that the local queue of \( s_t \) is full just after \( C_i \). Observation 33 implies that \( s_t \) forwarded the token to \( s_{t+1} \) in some configuration between \( C_i \) and \( C_{i+1} \). Notice that then, \( s_{t+1} = s_{t+1} \). If the local queue of \( s_{t+1} \) is not full, then the condition of line 25 evaluates to true and therefore, line 26 is executed, enqueueing value \( v_{i+1} \) to it. Then at \( C_{i+1} \), \( lq_i^{t_i} = v_{i+1} \). By definition, \( Q_{i+1} = Q_i \cdot v_{i+1} \), and therefore, \( Q_{i+1} = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot v_{i+1} = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot v_{i+1} = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot v_{i+1} \). If the local queue of \( s_{t+1} \) is full, then the condition of line 25 evaluates to false and therefore, line 35 is executed. The operation is linearized in the resulting configuration and NACK is sent to \( c \). Notice that in that case, the local queue of the server is not updated. Then, \( Q_{i+1} = Q_i = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot lq_i^{t_i+1} = lq_i^{h_i} \cdot lq_i^{h_i+1} \cdot \ldots \cdot lq_i^{t_i} \cdot lq_i^{t_i+1} \) and the claim holds.

The reasoning for the case where \( op_{t+1} \) is an instance of a dequeue operation is symmetrical. \( \square \)

From the above lemmas and observations we have the following theorem.
Theorem 38. The token-based distributed queue implementation is linearizable. The time complexity and the communication complexity of each operation \( op \) is \( O(\text{NS}) \).

5.3 Token-Based Double Ended Queue (Deque)

The dequeue implementation is a natural generalization of the stack and queue implementations described previously. The dequeue implementation is analogous to the queue implementation described in section 5.2. To provide a deque, we add actions to the queue’s design to support the additional operations supported by a deque. We retain the static ordering of the servers and the head and tail tokens. Again, each server uses a local data structure, this time a deque, on which the server is allowed to execute enqueues or dequeues to the appropriate end, only if it has either the tail token or the head token. The head and tail tokens are initially held by \( s_0 \), but can be reassigned to other servers during the execution.

5.3.1 Algorithm Description

Algorithm 19 presents the events triggered in a server \( s \) and \( s \)'s actions for each event. Each server, in addition to its id (\( \text{my}_\text{sid} \)), maintains a local deque (\( \text{ldeque} \)) to store elements of the implemented deque. For the token management, each server has two boolean flags (\( \text{hasHead} \) and \( \text{hasTail} \)), which are initialized true for the server \( s_0 \), and false for the rest. Furthermore, the servers maintain a fullDeque flag, similar to the fullQueue flag in Algorithm 15 of Section 5.2. Finally, the servers maintain a local array (called \( s \)'s clients array) for storing the requests they receive directly from clients, which is used in a similar way as in Section 5.2.

The types of messages a server \( s \) can receive are ENQ_T, DEQ_T for enqueuing at and dequeuing from the tail, ENQ_H, DEQ_H for enqueuing at and dequeuing from the head, and ACK or NACK sent by other servers as responses to \( s \)'s forwarded messages. Every message \( m \) a server receives, contains five fields: (1) the op field that represents the type of the request, (2) a data field, that contains either the data to be enqueued or \( \perp \), (3) a cid field that contains the client id that requested the operation op, (4) a sid field that contains the id of the server which started forwarding the request, and -1 otherwise, and (5) a tk field, a flag used to pass tokens from one server to another. The values that tk can take are either HEAD_TOKEN for the head token transition, TAIL_TOKEN for the tail token transition, or \( \perp \) for no token transition.

When server \( s \) receives the head token, that means that \( s \) can serve all operations in its client array regarding the head endpoint (\( \text{EnqueueHead}() \), \( \text{DequeueHead}() \)). In analogous way, when server \( s \) receives the token for the tail, that means that \( s \) can serve all operations in its client array for the tail (\( \text{EnqueueTail}() \), \( \text{DequeueTail}() \)). For this purpose, we use two functions,
Algorithm 19 Events triggered in a server

1. int my_sid;
2. LocalDeque ldeque = ∅;
3. LocalArray clients = ∅; /* Array of three values op, data, isServed */
4. boolean fullDeque = false; /* True when tail and head are in the same server and tail is before head */
5. boolean hasHead; /* Initially hasHead and hasTail are true in server 0, and false in the rest */
6. boolean hasTail;
7. a message ⟨op, data, cid, sid, tk⟩ is received:
8. if (clients[cid] ≠ ⊥ AND clients[cid].isServed) { /* If request was served earlier. */
9. send(cid, ⟨ACK, clients[cid].data, my_sid⟩);
10. clients[cid] = ⊥;
} else {
12. switch (op) {
13. case ENQ.T:
14. ServerEnqueueTail(op, data, cid, sid, tk);
15. break;
16. case DEQ.T:
17. ServerDequeueTail(op, cid, sid, tk);
18. break;
19. case ENQ.H:
20. ServerEnqueueHead(op, data, cid, sid, tk);
21. break;
22. case DEQ.H:
23. ServerDequeueHead(op, cid, sid, tk);
24. break;
25. case ACK:
26. clients[cid] = ⊥;
27. send(cid, ⟨ACK, data, sid⟩);
28. break;
29. case NACK:
30. clients[cid] = ⊥;
31. send(cid, ⟨NACK, ⊥, sid⟩);
32. break;
33. }
}

ServeOldHeadOps() and ServeOldTailOps(). The clients whose requests are served, are not informed until server \( s \) receives their requests completes a round-trip on the ring and returns back to \( s \).

When a message is received (line 7), server \( s \) checks if the message is stored in its client array (line 8). If it is, \( s \) sends an ACK message to client with \( cid \) (line 9) and removes the entry in the clients array (line 10). If it is not, \( s \) checks the message’s operation code and acts accordingly (line 12). Messages with operation code ENQ.T, DEQ.T, ENQ.H, DEQ.H (lines 13, 16, 19 and 22, respectively) are handled by functions ServerEnqueueTail()
and $\mathrm{ServerDequeueHead}()$ (line 23), respectively. These functions are presented in Algorithms 20-23. If the message is of type $\mathrm{ACK}$ (line 25) or $\mathrm{NACK}$ (line 29), $s$ sets the $\mathrm{cid}$ entry of the clients array to $\perp$ (lines 26, 30), and sends an $\mathrm{ACK}$ (line 27) or $\mathrm{NACK}$ (line 31) to the client.

Algorithm 20 presents pseudocode for function $\mathrm{ServerEnqueueTail}()$, which is called by a server $s$ when an $\mathrm{ENQ\_T}$ message is received. First, $s$ checks whether the $\mathrm{tk}$ field of the message contains $\mathrm{TAIL\_TOKEN}$ (line 34). In such a case, the message received by $s$ denotes a tail token transition. So, $s$ sets its $\mathrm{hasTail}$ flag to true (line 35) and if it also had the head token from a previous round, it sets its $\mathrm{fullDeque}$ flag to true (line 36). At this point, $s$ has just received the global deque’s tail, so it must serve all old operations that clients have requested directly from this server to be performed in the deque’s tail. For that purpose, the server calls the $\mathrm{ServeOldTailOps}()$ function (line 37). Then, the server checks whether the global deque is full and also the $\mathrm{ldeque}$ is full and if it is full, means that there is no free space left for the operation, thus $s$ responds to the request with with a $\mathrm{NACK}$ (lines 40, 42). Otherwise, if the server can serve the request, it enqueues the received data to the tail of $\mathrm{ldeque}$ (line 44) and responds with an $\mathrm{ACK}$ (lines 46, 48). In any other case, $s$ cannot serve the request received, but some other server might be capable to do so, so the message must be forwarded. Server $s$ finds the next server (line 50) and if $s$ handles the global deque’s tail, turns its flags ($\mathrm{hasTail}$ and $\mathrm{fullDeque}$) to false (line 52) and marks the token as $\mathrm{TAIL\_TOKEN}$ for the token transition (line 54). If the message was send by a client, $s$ stores the request to the $\mathrm{clients}$ array (line 58). Finally, $s$ forwards the message to the next server (lines 59, 61).

Algorithm 21 presents pseudocode for function $\mathrm{ServerDequeueTail}()$, which is called by a server $s$ when a $\mathrm{DEQ\_T}$ message is received. First, $s$ checks whether the $\mathrm{tk}$ field of the message contains $\mathrm{TAIL\_TOKEN}$ (line 63). In such a case, the message received by $s$, denotes a tail token transition. So, $s$ sets its $\mathrm{hasTail}$ flag to true (line 64). then, $s$ serves all old operations that clients have requested directly from this server to be performed in the deque’s tail. For that purpose, the server calls $\mathrm{ServeOldTailOps}()$ (line 65). Afterwards, $s$ checks if the “global” deque is empty and if it is empty, $s$ responds with a $\mathrm{NACK}$ (lines 68, 70). Otherwise, if $s$ can serve the request, it dequeues the data from the tail of its local deque (line 71), and sends them to the appropriate server or client with an $\mathrm{ACK}$ (lines 74, 76). In any other case, $s$ cannot serve the request received, but some other server might be capable to do so, thus the message must be forwarded. Server $s$ finds the previous server $s_{\text{prev}}$ (line 78) and if $s$ handles the global deque’s tail, turns its flag for the tail to false (line 80) and marks $\mathrm{tk}$ as $\mathrm{TAIL\_TOKEN}$ for the token transition (line 81). If the message received was send by a client, $s$ stores the request to its $\mathrm{clients}$ array (line 85). Finally, $s$ forwards the message to the previous server (lines 58, 59).

Algorithm 22 presents pseudocode for function $\mathrm{ServerEnqueueHead}()$,
which is called by a server when an ENQ_H message is received. First, checks if it has the token for global deque’s head (line 90) and if this is the case, the server sets its hasHead flag to true (line 91). If already has the token for the global deque’s tail, it sets its fullDeque flag to true (line 92). At this point, has just received the global deque’s head, so it must serve all old operations that clients have requested directly from this server to be performed in the deque’s head. For that purpose, server calls ServeOldHeadOps() (line 93), which iterates server’s clients array and serves all operations for the deque’s head. Then, the server checks whether the global deque is full and also the ldeque is full and if it is full, means that
Algorithm 21 Server helping function for handling a dequeue request to the global deque’s tail

```c
void ServerDequeueTail(int op, int cid, int sid, enum tk) {
    if (tk == TAIL_TOKEN) {
        hasTail = true;
        ServeOldTailOps(clients, fullDeque);
    }
    if (hasHead AND hasTail AND !fullDeque AND IsEmpty(ldeque)) {
        /* Deque is empty, can’t dequeue */
        if (sid == -1) /* From client */
            send(cid, ⟨NACK, ⊥, my_sid⟩);
        else /* from server */
            send(sid, ⟨NACK, ⊥, cid, my_sid, ⊥⟩);
    } else if (hasTail AND !IsFull(ldeque)) {
        /* Server can dequeue */
        data = dequeue_tail(ldeque);
        if (sid == -1) /* From client */
            send(cid, ⟨ACK, data, my_sid⟩);
        else /* from server */
            send(sid, ⟨ACK, data, cid, my_sid, ⊥⟩);
    } else {
        /* Server can’t dequeue and global deque is not empty. */
        psid = find_previous_server(my_sid);
        if (hasTail) {
            hasTail = false;
            tk = TAIL_TOKEN;
        } else {
            tk = ⊥;
        }
        if (sid == -1) /* From client */
            clients[cid] = ⟨DEQ_T, ⊥, false⟩;
        send(psid, ⟨DEQ_T, ⊥, cid, my_sid, tk⟩);
    }
}
```

there is no free space left for the operation, thus s responds to the request with a NACK (lines 96, 98). Otherwise, if the server can serve the request, it enqueues the received data to the deque’s head (line 100) and responds with an ACK (lines 102, 104). In any other case, s cannot serve the request received, but some other server might be capable to do so, so the message must be forwarded. Server s finds the previous server (line 106) and if s handles the global deque’s head, turns its flags (hasHead and fullDeque) to false (line 108) and marks the tk as HEAD_TOKEN for the token transition (line 110). If the message received was send by a client, s stores the request to its clients array (line 114). Finally, s forwards the message to the previous server (lines 115, 117).
Algorithm 22 Server helping function for handling an enqueue request to the global deque’s head

```c
void ServerEnqueueHead(int op, Data data, int cid, int sid, enum tk) {
    if (tk == HEAD_TOKEN) {
        hasHead = true;
        if (hasTail) fullDeque = true;
        ServeOldHeadOps(clients);
    }
    if (fullDeque AND IsFull(ldeque)) { /* Deque is full, can’t enqueue */
        if (sid == -1) /* From client */
            send(cid, ⟨NACK, ⊥, my_sid⟩);
        else /* from server */
            send(sid, ⟨NACK, ⊥, cid, my_sid, ⊥⟩);
    } else if (hasHead AND !IsFull(ldeque)) { /* Server can enqueue. */
        enqueue_head(ldeque, data);
        if (sid == -1) /* From client */
            send(cid, ⟨ACK, ⊥, my_sid⟩);
        else /* from server */
            send(sid, ⟨ACK, ⊥, cid, my_sid, ⊥⟩);
    } else { /* Server can’t dequeue and global deque is not full. */
        psid = find_previous_server(my_sid);
        if (hasHead) {
            hasHead = false;
            fullDeque = false;
            tk = HEAD_TOKEN;
        } else {
            tk = ⊥;
        }
        if (sid == -1) { /* From client */
            clients[cid] = ⟨ENQ_H, data, false⟩;
            send(psid, ⟨ENQ_H, data, cid, my_sid, tk⟩);
        } else { /* from server */
            send(psid, ⟨ENQ_H, data, cid, sid, tk⟩);
        }
    }
}
```

Algorithm 23 presents pseudocode for function `ServerDequeueHead()`, which is called by a server when a `DEQ_H` message is received by some server `s`. First, `s` checks if it has the token for global deque’s head (line 119) and if this is the case, the server sets its `hasHead` flag to `true` (line 120). At this point, server `s` has just received the global deque’s head, so it must serve all old operations that clients have requested directly from this server to be performed in the deque’s head. For that purpose, the server calls the `ServeOldHeadOps()` routine (line 121), which iterates `s`’s `clients` array and serves all operations for the deque’s head. Then, the server checks if the global deque is empty and if it is, `s` responds to the request with a NACK.
Algorithm 23: Server helping function for handling an dequeue request to
the global deque's head

```c
void ServerDequeueHead(int op, int cid, int sid, enum tk) {
    hasHead = true;
    ServeOldHeadOps(clients);
    if (hasHead AND hasTail AND !fullDeque AND IsEmpty(ldeque)) {
        /* Deque is empty, can't dequeue */
        if (sid == -1) /* From client */
            send(cid, ⟨NACK, ⊥, my_sid⟩);
        else /* from server */
            send(sid, ⟨NACK, ⊥, cid, my_sid, ⊥⟩);
    } else if (hasHead AND !IsFull(ldeque)) { /* Server can Server can enqueue. */
        data = dequeue_head(ldeque);
        if (sid == -1) /* From client */
            send(cid, ⟨ACK, data, my_sid⟩);
        else /* from server */
            send(sid, ⟨ACK, data, cid, my_sid, ⊥⟩);
    } else { /* Server can't dequeue and global deque is not empty. */
        nsid = find_next_server(my_sid);
        hasHead = false;
        tk = HEAD_TOKEN;
    }
    if (sid == -1) { /* From client */
        clients[cid] = ⟨DEQ_H, ⊥, false⟩;
        send(nsid, ⟨DEQ_H, ⊥, cid, my_sid, tk⟩);
    } else { /* from server */
        send(nsid, ⟨DEQ_H, ⊥, cid, sid, tk⟩);
    }
}
```

(lines 124, 126). Otherwise, if the server can serve the request, it dequeues
the data from the head (line 128) and sends them to the sender or client
together with an ACK (lines 130, 132). In any other case, s cannot serve the
request received, but some other server might be capable to do so, so the
message must be forwarded. Server s finds the next server (line 134) and
if s has the token for the global head, turns its flag for the head to false
(line 135) and and marks the tk as HEAD_TOKEN for the token transition
(line 136). If the message received was send by a client, s stores the request
to its clients array (line 140). Finally, s forwards the message to the next
server (lines 141, 143).

Algorithm 24 presents pseudocode for the client functions. These are
EnqueueTail(), DequeueTail(), EnqueueHead(), DequeueHead(). All
clients have two local variables, head_sid and tail_sid, to store the last known server to have the head token and the tail token, respectively. These variables initially store the id of the server zero, but may change values during runtime. The messages received by clients contain two fields: status which contains either the data from a dequeue operation, or ⊥ in case of an enqueue, and tail_sid or head_sid, depending on the function, which contains the id of the server that currently holds the token.

**Algorithm 24** Enqueue and dequeue operations for a client of the token-based deque.

```plaintext
144 int tail_sid = 0, head_sid = 0;

145 Data EnqueueTail(int cid, Data data) {
    146    send(tail_sid, ⟨ENQ_T, data, cid, −1, ⊥⟩);
    147    ⟨status, tail_sid⟩ = receive();
    148    return status;
}

149 Data DequeueTail(int cid) {
    150    send(tail_sid, ⟨DEQ_T, ⊥, cid, −1, ⊥⟩);
    151    ⟨status, tail_sid⟩ = receive();
    152    return status;
}

153 Data EnqueueHead(int cid, Data data) {
    154    send(head_sid, ⟨ENQ_H, data, cid, −1, ⊥⟩);
    155    ⟨status, head_sid⟩ = receive();
    156    return status;
}

157 Data DequeueHead(int cid) {
    158    send(head_sid, ⟨DEQ_H, ⊥, cid, −1, ⊥⟩);
    159    ⟨status, head_sid⟩ = receive();
    160    return status;
}
```

For enqueuing at the deque’s tail, clients calls EnqueueTail() (line 145). This function sends an ENQ_T message to the last known server, which has the token for the global tail (line 146). The server with the tail token may have changed, but the client is still unaware of the change. In that case, server s, which received the message, stores this message in its clients array and then forwards the message to the next server in the order, as described in Algorithm 20. During this time, the client blocks waiting for a server’s response. Once the client receives the response (line 147), it returns the contents of the status variable (line 148).
Algorithm 25 Auxiliary functions for a server of the token-based deque.

```c
void ServeOldTailOps(void) {
    LocalSet eliminated = ∅
    for each cid1 ∉ eliminated such that clients[cid1].op == ENQ.T {
        if there is cid2 ∉ eliminated such that clients[cid2].op == DEQ.T {
            clients[cid2].data = clients[cid1].data;
            clients[cid1].isServed = true;
            clients[cid2].isServed = true;
            eliminated = eliminated ∪ {cid1, cid2};
        }
    }
    if (!fullDeque) {
        for each cid such that clients[cid].op == ENQ.H {
            if (!IsFull(ldeque)) {
                enqueue_tail(ldeque, clients[cid].data);
            }
        }
        for each cid such that clients[cid].op == DEQ.T {
            if (!IsEmpty(ldeque)) {
                clients[cid].data = dequeue_tail(ldeque);
            }
        }
    }
}

void ServeOldHeadOps(void) {
    LocalSet eliminated = ∅
    for each cid1 ∉ eliminated such that clients[cid1].op == ENQ.H {
        if there is cid2 ∉ eliminated such that clients[cid2].op == DEQ.H {
            clients[cid2].data = clients[cid1].data;
            clients[cid1].isServed = true;
            clients[cid2].isServed = true;
            eliminated = eliminated ∪ {cid1, cid2};
        }
    }
    if (!fullDeque) {
        for each cid such that clients[cid].op == ENQ.H {
            if (!IsFull(ldeque)) {
                enqueue_head(ldeque, clients[cid].data);
            }
        }
        for each cid such that clients[cid].op == DEQ.H {
            if (!IsEmpty(ldeque)) {
                clients[cid].data = dequeue_head(ldeque);
            }
        }
    }
}
```
The enqueuing at the deque’s head is symmetrical to this approach and is achieved with the function `EnqueueHead()` (line 153). The only difference is that the server which is send the message to, is the server with id `head_sid` (line 154) and the message’s operation code is `ENQ_H` instead of `ENQ_T`.

For dequeuing at the deque’s tail, clients call `DequeueTail()` (line 149). This function sends a `DEQ_T` message to the last known server which has the token for the global tail (line 150). Then, the client waits for server to respond. Once the client receives the response (line 151), it returns the contents of the `status` variable (line 152).

The de dequeuing at the deque’s head is symmetrical to this approach and is achieved with the function `DequeueHead()` (line 157). The only difference is that the server, to whom the message was sent, is the server with id `head_sid` (line 158) and the message’s operation code is `DEQ_H` instead of `DEQ_T`.

### 5.3.2 Proof of Correctness

Let $\alpha$ be an execution of the token-based deque algorithm presented in Algorithms 19, 20, 21, 22, 23, 24, and 25.

Each server maintains local boolean variables `hasHead` and `hasTail`, with initial values `false`. Whenever some server $s_i$ receives a `TAIL_TOKEN` message, i.e. a message with its `tk` field equal to `TAIL_TOKEN` (line 34, line 63), the value of `hasTail` is set to `true` (line 35, line 64). By inspection of the pseudocode, it follows that the value of `hasTail` is set to `false` if the local deque of $s_i$ is full (line 52, line 80); then, a `TAIL_TOKEN` message is sent to the next or previous server (line 61, line 88). The same holds for `hasHead` and `HEAD_TOKEN` messages, i.e. messages with their `tk` field equal to `HEAD_TOKEN`. Thus, the following observations holds.

**Observation 39.** At each configuration in $\alpha$, there is at most one server for which the local variable `hasHead` (`hasTail`) has the value `true`.

**Observation 40.** In some configuration $C$ of $\alpha$, a `TAIL_TOKEN` message is sent from a server $s_j$, $0 \leq j < NS - 1$, to a server $s_k$, where $k = (j + 1) \mod NS$, only if the local deque of $s_j$ is full in $C$. A `TAIL_TOKEN` message is sent from a server $s_j$, $0 \leq j < NS - 1$, to a server $s_k$, where $k = (j - 1) \mod NS$, only if the local deque of $s_j$ is empty in $C$.

Similarly, a `HEAD_TOKEN` message is sent from $s_j$ to $s_k$, where $k = (j + 1) \mod NS$, only if the local deque of $s_j$ is empty in $C$. A `HEAD_TOKEN` message is sent from $s_j$ to $s_k$, where $k = (j - 1) \mod NS$, only if the local deque of $s_j$ is full in $C$.

By inspection of the pseudocode, we see that a server performs an enqueue (dequeue) back operation on its local deque `ldeque` either when executing line 44 (line 72) or when it executes `ServeOldTailOps`. Further inspection of the pseudocode (lines 34-36, line 43, as well as lines 63-65, line 67, line 92, line 114)
71), shows that these lines are executed when $hasTail = \text{true}$. By inspection of the pseudocode, the same can be shown for $hasHead$. Then, the following observation holds.

**Observation 41.** Whenever a server $s_j$ performs an enqueue or dequeue back (front) operation on its local deque, it holds that its local variable $hasTail$ (hasHead) is equal to $\text{true}$.

If $hasTail = \text{true}$ (hasHead = $\text{true}$) for some server $s$ in some configuration $C$, then we say that $s$ has the tail (head) token. The server that has the tail token is referred to as *tail token server*. The server that has the head token is referred to as *head token server*.

By a straight-forward induction, the following lemma can be shown.

**Lemma 42.** The mailbox of a client in any configuration of $\alpha$ contains at most one incoming message.

Let $op$ be any operation in $\alpha$. We assign a linearization point to $op$ by considering the following cases:

- If $op$ is an enqueue back operation for which a tail token server executes an instance of Algorithm 19, then it is linearized in the configuration resulting from the execution of either line 40, or line 44, or line 165, or line 172, whichever is executed for $op$ in that instance of Algorithm 19 by the tail token server.
- If $op$ is a dequeue back operation for which a head token server executes an instance of Algorithm 19, then it is linearized in the configuration resulting from the execution of either line 68, or line 72, or line 165, or line 176, whichever is executed for $op$ in that instance of Algorithm 19 by the tail token server.
- If $op$ is an enqueue front operation for which a tail token server executes an instance of Algorithm 19, then it is linearized in the configuration resulting from the execution of either line 96, or line 100, or line 182, or line 189, whichever is executed for $op$ in that instance of Algorithm 19 by the head token server.
- If $op$ is a dequeue front operation for which a head token server executes an instance of Algorithm 19, then it is linearized in the configuration resulting from the execution of either line 124, or line 128, or line 182, or line 193, whichever is executed for $op$ in that instance of Algorithm 19 by the head token server.

**Lemma 43.** The linearization point of an enqueue (dequeue) operation $op$ is placed in its execution interval.

*Proof.* Assume that $op$ is an enqueue back operation and let $c$ be the client that invokes it. After the invocation of $op$, $c$ sends a message to some server $s$ (line 146) and awaits a response. Recall that routine $\text{receive()}$ (line 147) blocks until a message is received. The linearization point of $op$ is placed
in the configuration resulting from the execution of either line 40, or line 44, or line 165, or line 172 by \( s_t \) for \( op \). Notice that since the execution of Algorithm 19 by \( s_t \) is triggered by a message that contains the request for \( op \), either of these lines is executed after the request by \( c \) is received, i.e. after \( c \) invokes \texttt{EnqueueTail}, and thus, after the execution interval of \( op \) starts.

By definition, the execution interval of \( op \) terminates in the configuration resulting from the execution of line 148. By inspection of the pseudocode, this line is executed after line 147, i.e. after \( c \) receives a response by some server. In the following, we show that the linearization point of \( op \) occurs before this response is sent to \( c \).

Let \( s_j \) be the server that \( c \) initially sends the request for \( op \) to. By observation of the pseudocode, we see that \( c \) may either receive a response from \( s_j \) if \( s_j \) executes lines 9, or 40, or 46. To arrive at a contradiction, assume that either of these lines is executed in \( \alpha \) before the configuration in which the linearization point of \( op \) is placed. Thus, a tail token server \( s_t \) executes lines line 40, or line 44, or line 165, or line 172, in a configuration following the execution of lines 9, or 40, or 46 by \( s_j \). Since the algorithm is event-driven, inspection of the pseudocode shows that in order for a tail token server to execute these lines, it must receive a message containing the request for \( op \) either from a client or from another server.

Assume first that a tail token server executes the algorithm after receiving a message containing a request for \( op \) from a client. This is a contradiction, since, on one hand, \( c \) blocks until receiving a response, and thus, does not send further messages requesting \( op \) or any other operation, and since \( op \) terminates after \( c \) receives the response by \( s_j \), and on the other hand, any other request from any other client concerns a different operation \( op' \).

Assume next that a tail token server executes the algorithm after receiving a message containing the request for \( op \) from some other server. This is also a contradiction since inspection of the pseudocode shows that after \( s_j \) executes either of the lines that sends a response to \( c \), it sends no further message to some other server and instead, terminates the execution of that instance of the algorithm.

The argumentation regarding dequeue back, enqueue front, and dequeue front operations is analogous.

Denote by \( L \) the sequence of operations which have been assigned linearization points in \( \alpha \) in the order determined by their linearization points. Let \( C_i \) be the configuration at which the \( i \)-th operation \( op_i \) of \( L \) is linearized. Denote by \( \alpha_i \), the prefix of \( \alpha \) which ends with \( C_i \) and let \( L_i \) be the prefix of \( L \) up until the operation that is linearized at \( C_i \). Denote by \( D_i \) the sequence of values that a sequential deque contains after applying the sequence of operations in \( L_i \), in order, starting from an empty deque; let \( D_0 = \epsilon \), i.e. \( D_0 \) is the empty sequence. In the following, we denote by \( s_{t_i} \) the tail token
server at \(C_i\) and by \(s_{h_i}\) the head token server at \(C_i\).

**Lemma 44.** For each \(i, i \geq 0, \) if \(ld_i^j\) are the contents of the local deque of server \(s_j\) at \(C_i, \) \(h_i \leq j \leq t_i, \) at \(C_i, \) then it holds that \(D_i = ld_i^{h_i} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i}\) at \(C_i.\)

**Proof.** We prove the claim by induction on \(C\) at \(s = 0.\)

Assume that \(op_{i+1}\) is an enqueue back operation by client \(c.\) Furthermore, distinguish the following two cases:

- Assume that \(t_i = t_{i+1}.\) Then, by the induction hypothesis, \(D_i = ld_i^{h_i} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i}.\) In case the local queue of \(s_{t_i}\) is not full, \(s_{t_i}\) enqueues the value \(v_{i+1}\) of the data field of the request for \(op_{i+1}\) in the local deque (line 44 or line 172). Notice that, by Observation 41 changes on the local deques of servers occur only on token servers. Notice also that those changes occur only in a step that immediately precedes a configuration in which an operation is linearized. Thus, no further change occurs on the local deques of \(s_{h_i}, s_{h_{i+1}}, \ldots, s_{t_i}\) between \(C_i\) and \(C_{i+1},\) other than the enqueue on \(ld_i^{h_i}\). Then, it holds that \(D_{i+1} = D_i \cdot v_{i+1} = ld_i^{h_i} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i} \cdot v_{i+1} = ld_i^{h_i} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i+1} = ld_i^{h_i} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i+1},\) and if the head token server does not change between \(C_i\) and \(C_{i+1},\) then \(h_{i+1} = h_i\) and \(D_{i+1} = ld_i^{h_i} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i+1}\) and the claim holds. If the head token server changes, i.e., if \(h_{i+1} \neq h_i,\) then by Observation 40, \(ld_i^{h_i} = \emptyset\) and the claim holds again.

In case the local deque of \(s_{t_i}\) is full and since by assumption, \(s_{t_i} = s_{t_{i+1}},\) it follows by inspection of the pseudocode (line 31) and the definition of linearization points, that \(s_{t_{i+1}} = s_{h_{i+1}}.\) In this case, \(s_{t_{i+1}}\) responds with a NACK to \(c\) and the local deque remains unchanged. Since no token server changes between \(C_i\) and \(C_{i+1}, D_{i+1} = D_i = ld_i^{h_i} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i+1}\) and the claim holds.

- Next, assume that \(t_i \neq t_{i+1}.\) This implies that the local deque of \(s_{t_i}\) is full just after \(C_i.\) Observation 40 implies that \(s_{t_i}\) forwarded the token to \(s_{t_{i+1}}\) in some configuration between \(C_i\) and \(C_{i+1}.\) Notice that then, \(s_{t_{i+1}} = s_{t_{i+1}}.\) If the local deque of \(s_{t_{i+1}}\) is not full, then the condition of line 43 evaluates to \(\text{true}\) and therefore, line 44 is executed, enqueuing value \(v_{i+1}\) to it. Then at \(C_{i+1}, ld_{i+1}^{t_i+1} = v_{i+1}.\) By definition, \(D_{i+1} = D_i \cdot v_{i+1},\) and therefore, \(D_{i+1} = ld_i^{h_i} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i} \cdot v_{i+1} = ld_i^{h_i} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i} \cdot v_{i+1} = ld_i^{h_i+1} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i} \cdot v_{i+1} = ld_i^{h_i+1} \cdot ld_i^{h_i+1} \cdots ld_i^{t_i+1} \cdot ld_i^{t_i+1} \cdot ld_i^{t_i+1} \cdots ld_i^{t_i+1} \cdot ld_i^{t_i+1} \) and the claim holds. If the local deque of \(s_{t_{i+1}}\) is full, then the condition of line 43 evaluates to \(\text{false}\) and therefore, line 40 is executed. The operation is linearized in the resulting configuration and NACK is sent.
to c. Notice that in that case, the local deque of the server is not updated. Then, $D_{i+1} = D_i = ld_i^h_i \cdot ld_i^{h+1}_i \cdot \ldots \cdot ld_i^{t+1} \cdot ld_{i+1}^h \cdot ld_{i+1}^{h+1} \cdot \ldots \cdot ld_{i+1}^t \cdot ld_{i+1}^{t+1}$, and the claim holds.

The reasoning for the case where $op_{i+1}$ is an instance of a dequeue back, enqueue front, or enqueue back operation is symmetrical.

From the above lemmas and observations we have the following theorem.

**Theorem 45.** The token-based distributed deque implementation is linearizable. The time complexity and the communication complexity of each operation $op$ is $O(NS)$. 

### 5.4 Hierarchical approach.

In this section, we outline how the hierarchical approach, described in Section 3.1, is applied to the token-based designs.

Only the island masters play the role of clients to the algorithms described in this section. So, it is each island master $m_i$ that keeps track of the last server(s), which responded to its batches of requests. In the stack and deque implementations, $m_i$ performs elimination before contacting a server. In the queue implementation, batching is done by having each batch containing requests of the same type. In the deque implementation, each batch contains requests of the same type that are to be applied to the same endpoint. A batch can be sent to a server using DMA: the same could be done for getting back the responses. A server that does not hold the appropriate token to serve a batch of requests, forwards the entire batch to the next (or previous) server. Since token-based algorithms exploit locality, a batch of requests will be processed by at most two servers.

### 5.5 Dynamic Versions of the Implementations

The implementations presented above (in Section 5) are static. Their dynamic versions retain the placement of servers in a logical ring, and the token that renders the server able to execute operations in its local partition. In the static versions of the algorithms, when the servers consume all their predefined space for the data structure, the global (implemented) data structure is considered full, and the token server was sending NACK to clients to notify them of this event.

In the dynamic version, though, there is no upper bound to the number of elements that can be stored in the data structure. In order to modify the static version of the structures of this section, we remove the mechanism that sends NACK messages to clients. Instead, every time a server $s$ receives the token (regarding inserts), it allocates an additional chunk of memory for its local partition. Because of this circular movement of the token, the
elements are stored along a spiral path, that spans over all servers. Each chunk is marked with a sequence number, associated with the coil of the spiral, to distinguish the order of allocation.

An example of the transformation of a static algorithm to a dynamic is the dynamic version of the queue algorithm, presented in Algorithm 26. In this design a server $s$ uses two tokens, the head and tail token. In analogy, $s$ deploys two variables ($\text{tail\_round}$ and $\text{head\_round}$) to count the times the

**Algorithm 26** Events triggered in a server of a dynamic token-based deque.

1. a message $\langle op, data, cid, sid, tk \rangle$ is received:
2. if $(\text{clients[cid]} \text{ AND clients[cid].isServed})$ {
   /* If message has been served earlier. */
   3. send(cid, $\langle$ ACK, clients[cid].data, my_sid$\rangle$);
   4. clients[cid] = $\bot$;
   } else {
   6. switch (op) {
      7. case ENQ:
         8. if ($tk == \text{TAIL\_TOKEN}$) {
            9. $\text{hasTail}$ = true;
            10. ServeOldEnqueues();
         }
         11. if (!$\text{hasTail}$) { /* Server does not have token */
            12. nsid = find_next_server(my_sid);
            13. if (sid == $-1$) { /* From client. */
               14. clients[cid] = $\langle$ ENQ, data, false$\rangle$;
               15. send(nsid, $\langle$ ENQ, data, cid, my_sid, $\bot$);}
            } else {
               16. send(nsid, $\langle$ ENQ, data, cid, $\bot$); /* From server. */
            }
         }
      17. else if (!IsFull(lqueue)) {
         18. enqueue(lqueue, data, tail_round);
         19. if (sid == $-1$) /* From client. */
            20. send(cid, $\langle$ ACK, $\bot$, my_sid$\rangle$);
         21. else /* From server. */
            22. send(sid, $\langle$ ACK, $\bot$, cid, my_sid, $\bot$);
         }
      else { /* Server moves the tail token */
         25. nsid = find_next_server(my_sid);
         26. send(nsid, $\langle$ op, data, cid, my_sid, TAIL\_TOKEN$\rangle$);
         27. tail_round + +;
         28. allocate_new_space(lqueue, tail_round);
         29. $\text{hasTail}$ = false;
      }
   }
   30. break;
case DEQ:
  if (tk == HEAD_TOKEN) {
    hasHead = true;
    ServeOldDeques();
  }
  if (!hasHead) {
    nsid = find_next_server(my_sid);
    if (sid == -1) {/* From client */
      clients[cid] = ⟨DEQ, ⊥, false⟩;
      send(nsid, ⟨DEQ, ⊥, cid, my_sid⟩);
    } else {/* From server */
      send(nsid, ⟨DEQ, ⊥, cid, sid, ⊥⟩);
    }
  } else if (!IsEmpty(lqueue)) {/* can dequeue. */
    data = dequeue(lqueue, head_round);
    if (sid == -1) {/* From client */
      send(cid, ⟨ACK, data, my_sid⟩);
    } else {/* From server */
      send(sid, ⟨ACK, data, cid, my_sid, ⊥⟩);
    } else if (tail_round == head_round) {
      tail_round = head_round = 0;
      if (sid == -1) {/* empty to client */
        send(cid, ⟨NACK, ⊥, my_sid⟩);
      } else {/* empty to server */
        send(sid, ⟨NACK, ⊥, cid, my_sid, ⊥⟩);
      } else {/* Move the head token to next */
        nsid = find_next_server(my_sid);
        send(nsid, ⟨op, ⊥, cid, my_sid, HEAD_TOKEN⟩);
        head_round += 1;
        hasHead = false;
      }
    break;
  case ACK:
    clients[cid] = ⊥;
    send(cid, ⟨ACK, data, sid⟩);
    break;
  case NACK:
    clients[cid] = ⊥;
    send(cid, ⟨NACK, ⊥, sid⟩);
    break;
}
tokens have come to its possession. When a server $s$ receives an ENQ message (line 7) but has no space left to store the element (line 24), it forwards the request along with the token to the next server in the ring. Afterwards, $s$ increases by one the variable $\text{tail\_round}$ and allocates a new memory chunk, by calling $\text{allocate\_new\_space()}$, to be used during the next time the token comes to its possession.

For the DEQ operation, the server performs additional actions concerning the empty queue state (line 48), where after responding with a NACK, it re-initializes $\text{tail\_round}$ and $\text{head\_round}$ to be equal to zero (line 49). An empty queue implies that the allocated chunks for $\text{lqueue}$ are also empty, hence they can be recycled and be used again anew. During the head token transition, $s$ increases $\text{head\_round}$ by one chunk (line 57), so that when the head token comes to its possession to dequeue from the next memory.

The double ended queue (deque) algorithm is going to work verbatim after these modifications. For the stack implementation the modifications are analogous. In this design there is one token, hence each server associates one counter with the token rounds. Each time a server $s$ moves the token to another server because the local stack is full, it increases the counter and allocates a new chunk for future use, and $s$ moves the token due to an empty stack, the counter is decreased by one. However, the dynamic design for the stack would introduce the termination problem described for queues. Nevertheless, the problem can be solved by applying the same technique of using client arrays as we did to solve the problem in the queue implementation.
Chapter 6

Distributed Lists

A list is an ordered collection of elements. It can either be sorted, in which case the elements appear in the list in increasing (or decreasing) order of their keys, or unsorted, in which case the elements appear in the list in some arbitrary order (e.g. in the order of their insertion). A list \( L \) supports the operations \( \text{Insert} \), \( \text{Delete} \), and \( \text{Search} \). Operation \( \text{Insert}(L, k, I) \) inserts an element with key \( k \) and associated info \( I \) to \( L \). Operation \( \text{Delete}(L, k) \) removes the element with key \( k \) from \( L \) (if it exists), while operation \( \text{Search}(L, k) \) detects whether an element with key \( k \) is present in \( L \) and returns the information \( I \) that is associated with \( k \).

In this section, we first provide an implementation of an unsorted distributed list in which we follow a token-based approach for implementing \( \text{Insert} \). In this implementation, Search and Delete are highly parallel. We then build on this approach in order to get a distributed implementation of a sorted list.

6.1 Unsorted List

The list state is stored distributedly in the local memories of several of the available servers, potentially spreading among all of them, if its size is large enough. The proposed implementation follows a token-based approach for implementing insert. Thus, we assume that the servers are arranged on a logical ring, based on their ids.

At each point in time, there is a server (not necessarily always the same), denoted by \( s_t \), which holds the insert token, and serves insert operations. Initially, server \( s_0 \) has the token, thus the first element to be inserted in the list is stored on server \( s_0 \). Further element insertions are also performed on it, as long as the space it has allocated for the list does not exceed a threshold. In case server \( s_0 \) has to service an insertion but its space is filled up, it forwards the token by sending a message to the next server, i.e. server \( s_1 \). Thus, if server \( s_i \), \( 0 \leq i < N_S \), has the token, but cannot service an insertion request without exceeding the threshold, it forwards the token to
server \(s_{(i+1)} \mod NS\). When the next server receives the token, it allocates a memory chunk of size equal to threshold, to store list elements. When the token reaches \(s_{NS-1}\), if \(s_{NS-1}\) has filled all the local space up to a threshold, it sends the token again to \(s_0\). Then, \(s_0\) allocates more memory (in addition to the memory chunk it had initially allocated for storing list elements) for storing more list elements. The token might go through the server sequence again without having any upper-bound restrictions concerning the number of round-trips. In order for a server to know whether the token has performed a round-trip on the ring, and hence all servers have stored list elements, it deploys a variable to count the number of ring round-trips it knows that the token has performed.

Event-driven code for the server is presented in Algorithm 27. Each server \(s\) maintains a local list (\(llist\) variable) allocated for storing list elements, a \(token\) variable which indicates whether \(s\) currently holds the token, and a variable \(round\) to mark the ring round-trips the token has performed; \(round\) is initially 0, and is incremented after every transmission of the token to the next server.

Each message a server receives has five fields: (1) \(op\) that denotes the operation to be executed, (2) \(cid\) that holds the id of the client that initiated a request, (3) \(key\) that holds the value to be inserted, (4) \(mloop\) stands for “message loop”, a \texttt{boolean} value that denotes if the message has traversed the whole server sequence and (5) \(tk\) that is set when a forwarded message also denotes a token transition from one server to the other.

When a message is received, the server \(s\) first checks its type. If the message is of type \texttt{INSERT} (line 5), \(s\) first checks whether the message has the \(tk\) field marked. If it is marked (line 6), \(s\) sets a local variable \(token\) equal to its own id (line 7) and allocates additional space for its local part of the list (line 8).

Afterwards, \(s\) searches the part of the list that it stores locally, for an element with the same key (\(key\) variable in the algorithm) as the one to be inserted (line 9). Searching \(llist\) for the element has to be performed independently of whether the server holds the token or not. Since this design does not permit duplicate entries, if such an element is found, the server responds with \texttt{NACK} to the client (line 10). Otherwise (line 11), \(s\) checks whether the new element can be stored in \(llist\).

In case \(s\) does not hold the token (line 12), it is not allowed to perform an insertion, therefore it must forward the message to the next server in the ring. If \(s\) is not \(s_{NS-1}\) (line 14), it forwards to the next server the request (15). In case \(s\) is \(s_{NS-1}\), it means that all servers have been searched for the element and the element was not found. Server \(s\) sends the message to the next server (in order to eventually reach the token server), after marking the \(mloop\) field of the message as \texttt{true}, to indicate that the message has completed a full round-trip on the ring (line 16).
Algorithm 27 Events triggered in a server of the distributed unsorted list.

1. List llist = ∅;
2. int my_id, next_id, token = 0, round = 0;
3. a message ⟨op, cid, key, data, mloop, tk⟩ is received:

4. switch (op) {
5.   case INSERT:
6.     if (tk == TOKEN) {
7.       token = my_id;
8.       allocate_new_memory_chunk(llist, round);
9.     }
10.    status1 = search(llist, key);
11.   if (status1) send(cid, NACK);
12.   else {
13.      if (token ≠ my_id) {
14.        next_id = get_next(my_id);
15.        if (my_id ≠ NS − 1) {
16.          send(next_id, ⟨op, cid, key, data, mloop, tk⟩);
17.        } else send(next_id, ⟨op, cid, key, data, true, tk⟩);
18.      } else {
19.        if ((my_id ≠ NS − 1) AND (round > 0) AND !(mloop)) {
20.          next_id = get_next(my_id);
21.          send(next_id, ⟨op, cid, key, data, mloop, tk⟩);
22.        } else {
23.          status2 = insert(llist, round, key, data);
24.          if (status2 == false) {
25.            round += ;
26.            token = get_next(my_id);
27.            send(token, ⟨op, cid, key, data, mloop, TOKEN⟩);
28.          } else send(cid, ACK);
29.        }
30.      }
31.   }
32.   break;
33.   case SEARCH:
34.     status1 = search(llist, key);
35.   if (status1) send(cid, ⟨ACK, my_id⟩);
36.   else send(cid, ⟨NACK, my_id⟩);
37.   break;
38.   case DELETE:
39.     status1 = delete(llist, key);
40.   if (status1) send(cid, ACK);
41.   else send(cid, NACK);
42.   break;
43. }
44.

On the other hand, if s holds the token (line 17), it must first check whether there is room in llist to insert the element in it. If there is room in llist and the local variable round of s equals to false (which means that
the list does not expand to the next servers) or the message has already performed a round-trip on the ring, then \( s \) inserts the element and returns \( \text{ACK} \). If however, \( \text{round} > 0 \) and the message has not performed a round trip on the ring (\( \text{mloop} == \text{false} \)), \( s \) continues forwarding the message.

If the token server’s local memory is out of sufficient space (line 23) (i.e. the \( \text{insert}() \) function was unsuccessful), \( s \) forwards the message to the next server the \( tk \) field with \( \text{TOKEN} \) (line 26) to indicate that this server will become the new token server after \( s \). Also, \( s \) increments \( \text{round} \) by one to count the number of times the token has passed from it. The \( \text{round} \) variable is also used by function \( \text{allocate\_new\_memory\_chunk()} \) that allocates additional space for the list (line 8).

Notice that, contrary to other token-based implementations presented in previous sections, the token server of the unsorted list does not need to rely on client tables in order to stop a message from being incessantly forwarded from one server to another, without ever being served. By virtue of having clients always sending their insert requests to \( s_0 \), an insert request \( r_j \) that arrives at \( s_0 \) before some other insert request \( r_k \), is necessarily served before \( r_k \). The scenario where insert requests constantly arrive at the token server before \( r_j \), making the token travel to the next server before \( r_j \) can be served, is thus avoided.

Upon receiving a \( \text{SEARCH} \) request from a client (line 29), a server searches for the requested element in its local list (line 30) and sends \( \text{ACK} \) to the server if the element is found (line 31) and \( \text{NACK} \) otherwise (line 32).

Upon receiving a \( \text{DELETE} \) request from a client (line 34), a server attempts to delete the requested element from its local list (line 35) and sends \( \text{ACK} \) to the server if the deletion was successful (line 36). Otherwise it sends \( \text{NACK} \) (line 37).

The pseudocode of the client is presented in Algorithm 28. Notice that insert operations in the proposed implementation are executed in sequence and must necessarily pass through server 0 and be forwarded through the server ring, if necessary due to space constraints. Search and Delete operations, on the contrary, are executed in parallel.

In order to execute an insertion, a client calls \( \text{ClientInsert()} \) (line 39) which sends an \( \text{INSERT} \) message (line 41) to server 0, regardless of which server holds the token in any given configuration, and then blocks waiting for a response (line 42). If the client receives \( \text{ACK} \) from a server, then the element was inserted correctly. If the client receives \( \text{NACK} \), then the insertion failed, due to either limited space, or the existence of another element with the same key value.

For a search operation the client calls \( \text{ClientSearch()} \) (line 44). The client sends a \( \text{SEARCH} \) request to all servers (line 49) and waits to receive a response message (line 51) from each server (do while loop of lines 50-54). The requested element is in the list if the client receives \( \text{ACK} \) from some server (line 52). A delete operation proceeds similarly to \( \text{ClientSearch()} \). It is initiated by a client by sending a \( \text{DELETE} \) request to all servers (line
The client then waits to receive a response message (line 63) from each server (do while loop of lines 62-66). The requested element has been found in the list of some client and deleted from there, if the client receives ACK from some server $s$.

**Algorithm 28** Insert, Search and Delete operation for a client of the distributed list.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>boolean ClientInsert(int $cid$, int $key$, data $data$) {</td>
</tr>
<tr>
<td>40</td>
<td>boolean $status$;</td>
</tr>
<tr>
<td>41</td>
<td>send(0, ⟨INSERT, $cid$, $key$, $data$, false, −1⟩);</td>
</tr>
<tr>
<td>42</td>
<td>$status$ = receive();</td>
</tr>
<tr>
<td>43</td>
<td>return $status$;</td>
</tr>
<tr>
<td>44</td>
<td>}</td>
</tr>
<tr>
<td>45</td>
<td>boolean ClientSearch(int $cid$, int $key$) {</td>
</tr>
<tr>
<td>46</td>
<td>int $sid$;</td>
</tr>
<tr>
<td>47</td>
<td>int $c$ = 0;</td>
</tr>
<tr>
<td>48</td>
<td>boolean $status$;</td>
</tr>
<tr>
<td>49</td>
<td>boolean $found$ = false;</td>
</tr>
<tr>
<td>50</td>
<td>send_to_all_servers(⟨SEARCH, $cid$, $key$, ⊥, false, −1⟩);</td>
</tr>
<tr>
<td>51</td>
<td>do {</td>
</tr>
<tr>
<td>52</td>
<td>⟨$status$, $sid$⟩ = receive();</td>
</tr>
<tr>
<td>53</td>
<td>if ($status$ == ACK) $found$ = true;</td>
</tr>
<tr>
<td>54</td>
<td>$c$ + +;</td>
</tr>
<tr>
<td>55</td>
<td>} while ($c$ &lt; NS);</td>
</tr>
<tr>
<td>56</td>
<td>return $found$;</td>
</tr>
<tr>
<td>57</td>
<td>}</td>
</tr>
<tr>
<td>58</td>
<td>boolean ClientDelete(int $cid$, int $key$) {</td>
</tr>
<tr>
<td>59</td>
<td>int $sid$;</td>
</tr>
<tr>
<td>60</td>
<td>int $c$ = 0;</td>
</tr>
<tr>
<td>61</td>
<td>boolean $status$;</td>
</tr>
<tr>
<td>62</td>
<td>boolean $deleted$ = false;</td>
</tr>
<tr>
<td>63</td>
<td>send_to_all_servers(⟨DELETE, $cid$, $key$, ⊥, false, −1⟩);</td>
</tr>
<tr>
<td>64</td>
<td>do {</td>
</tr>
<tr>
<td>65</td>
<td>⟨$status$, $sid$⟩ = receive();</td>
</tr>
<tr>
<td>66</td>
<td>if ($status$ == ACK) $deleted$ = true;</td>
</tr>
<tr>
<td>67</td>
<td>$c$ + +;</td>
</tr>
<tr>
<td>68</td>
<td>} while ($c$ &lt; NS);</td>
</tr>
<tr>
<td>69</td>
<td>return $deleted$;</td>
</tr>
</tbody>
</table>

### 6.1.1 Proof of Correctness

We sketch the correctness argument for the proposed implementation by providing linearization points. Let $\alpha$ be an execution of the distributed unsorted list algorithm presented in Algorithms 27 and 28. We assign linearization points to insert, delete and search operations in $\alpha$ as follows:
• *Insert.* Let \( op \) be any instance of \texttt{ClientInsert} for which an \texttt{ACK} or a \texttt{NACK} message is sent by a token server. Then, if \texttt{ACK} is sent by a token server for \( op \) (line 27), the linearization point is placed in the configuration resulting from the execution of line 22 that successfully inserted the required element into the server’s local list. If \texttt{NACK} is sent for \( op \) (line 10), then the linearization point is placed in the configuration resulting from the execution of line 9, where the search operation on the local list of the server returned \texttt{true}.

• Let \( op \) be any instance of \texttt{ClientDelete} for which an \texttt{ACK} or a \texttt{NACK} message is sent by a server. Then, if \texttt{ACK} is sent by a server \( s \) for \( op \), the linearization point is placed in the configuration resulting from the execution of line 35 by the server that sent the \texttt{ACK}. Otherwise, if the key \( k \) that \( op \) had to delete was not present in any of the local lists of the servers in the beginning of the execution interval of \( op \), then the linearization point of \( op \) is placed at the beginning of its execution interval. Otherwise, if \( k \) was present but was deleted by a concurrent instance \( op' \) of \texttt{ClientDelete}, then the linearization point is placed right after the linearization point of \( op' \).

• Let \( op \) be any instance of \texttt{ClientSearch} for which an \texttt{ACK} or a \texttt{NACK} message is sent by a server. Then, if \texttt{ACK} is sent by a server \( s \) for \( op \), the linearization point is placed in the configuration resulting from the execution of line 30 by the server that sent the \texttt{ACK}. Otherwise, if the key \( k \) that \( op \) had to find was not present in the list in the beginning of its execution interval, then the linearization point is placed there. Otherwise, if \( k \) was present but was deleted by a concurrent instance \( op' \) of \texttt{ClientDelete}, then the linearization point is placed right after the linearization point of \( op' \).

Lemma 46. Let \( op \) be any instance of an insert, delete, or a search operation executed by some client \( c \) in \( \alpha \). Then, the linearization point of \( op \) is placed in its execution interval.

*Proof.* Let \( op \) be an instance of an insert operation invoked by client \( c \). A message with the insert request is sent on line 41, after the invocation of the operation. Recall that routine \texttt{receive()} blocks until a message is received. Notice that both line 22 as well as line 9 are executed by a server before it sends a message to the client. Therefore, whether \( op \) is linearized at the point some server sends it a message on line 27 or on line 10, it terminates only after receiving it. Notice also that the operation terminates only after the client receives it. Thus, the linearization point is included in its execution interval.

By similar reasoning, if \( op \) is an instance of a delete operation that is linearized in the configuration resulting from the execution of line 35 or a search operation that is linearized in the configuration resulting from the execution of line 30, then the linearization point is included in the execution interval of \( op \).
Let \( op \) be an instance of a delete operation that deletes key \( k \) and that terminates after receiving only NACK messages on line 63. If \( k \) is not present in the list in the beginning of the execution interval of \( op \), then \( op \) is linearized at that point and the claim holds.

Consider the case where \( k \) is included in the list when \( op \) is invoked. By observation of the pseudocode (lines 34-38), we have that when a server receives a delete request by a client, it traverses its local part of the list and deletes the element with key equal to \( k \) (line 35), if it is included in it. By further observation of the pseudocode (lines 61-67), we have that after \( c \) invokes \( op \), it sends a delete request to all servers (line 61) and then awaits for a response from all of them (do while loop of lines 62-66). By assumption, all servers responds with NACK. Notice that this implies that between the execution of line 63 and 65 the element with key \( k \) is removed from the local list of \( s \) because of some other concurrent delete operation \( op' \) invoked by some client \( c' \). By scrutiny of the pseudocode, we have that a server that deletes an element from its local list, does so on line 35, which occurs before the server sends a response to the delete request. By definition, then, \( op' \) is linearized at the point \( s \) executes line 35, before it sends an ACK message to \( c' \). Since \( op' \) causes the element with key \( k \) to be removed from the local list of \( s \) between the execution of lines 63 and 65 by \( c \), its linearization point is included in the execution interval of \( op \). Since we place the linearization point of \( op \) right after the linearization point of \( op' \), the claim holds.

The argument is similar for when \( op \) is an instance of a search operation for key \( k \) that terminates after receiving a NACK message from all the servers on lines 50-54. □

Each server maintains a local variable \( token \) with initial value 0. Let some server \( s \) receive a message \( m \) in some configuration \( C \). If the field \( tk \) of \( m \) is equal to \( TOKEN \), we say that \( s \) receives a token message. Observe that when \( s \) receives a token message (line 7), the value of \( token \) is set to \( s \). Furthermore, when \( s \) executes line 25, where the value of \( token \) changes from \( s \) to \( s + 1 \), \( s \) also sends a token message to \( s + 1 \) (line 26). Notice that \( s \) can only reach and execute this line if the condition of the if clause of line 12 evaluates to \( false \), i.e. if \( token = s \). Then, the following holds:

\textbf{Observation 47.} At each configuration in \( \alpha \), there is at most one server \( s \) for which the local variable \( token \) has the value \( s \).

This server is referred to as \( token \) server. By the pseudocode, namely the if else clause of lines 12, 17, and by line 22, the following observations holds.

\textbf{Observation 48.} A server \( s \) performs insert operations on its local list in \( \alpha \) only during those subsequences of \( \alpha \) in which it is the token server.

Each server maintains a local list collection, \( llist \). By observation of the pseudocode, lines 9 and 10, we have that if an insert operation attempts to
insert key $k$ in either of the lists of a server $s$, but an element with that key already exists, then no second element for $k$ is inserted and the operation terminates. Thus, the following holds:

**Observation 49.** The keys contained in the list collection of $s$ in any configuration $C$ of $\alpha$ form a set.

We denote this set by $ll^s$. By scrutiny of the pseudocode, we see that a new list object is allocated in $llist$ each time a server receives a token message (lines 6-8). The new object is identified by the value of local variable round. By observation of the pseudocode, we further have that each time a server inserts a key into $ll^s$, it does so on the list object identified by round (line 22). We refer to this object as current list object. Then, based on lines 23-26 we have the following:

**Observation 50.** A token message is sent from a server $s$ to a server $(s+1) \mod NS$ in some configuration $C$ only if the current local list object of server $s$ is full at $C$.

Further inspection of the pseudocode shows that the local list object of a server is only accessed by the execution of line 9, 22, 30, or 35. From this, we have the following observation.

**Observation 51.** If an operation $op$ modifies the local list object of some server, then this occurs in the configuration in which $op$ is linearized.

Let $C_i$ be the configuration in which the $i$-th linearization point in $\alpha$ is placed. Denote by $\alpha_i$, the prefix of $\alpha$ which ends just after $C_i$ and let $L_i$ be the sequence of linearization points that is defined by $\alpha_i$. Denote by $S_i$ the set of keys that a sequential list contains after applying the sequence of operations that $L_i$ imposes. Denote by $S_i = \epsilon$ the empty sequence (the list is empty).

**Lemma 52.** Let $k$ be the token server in some configuration $C$ in which it receives a message $m$ for an insert operation $op$ with key $k$ invoked by client $c$. Then at $C$, no element with key $k$ is contained in the local list set of any other server $s \neq k$.

**Proof.** By inspection of the pseudocode, when a client $c$ sends a message $m$ to some server either on line 41, line 49, line 61, or line 65, the $mloop$ field of $m$ is equal to $\text{false}$. This field is set to $\text{true}$ when server $s_{NS-1}$ executes line 16. Notice that in the configuration in which this line is executed by $s_{NS-1}$, it is not the token server (otherwise the condition of line 12 would not evaluate to $\text{true}$ and the line would not be executed).

Consider the case where $m$ reaches a server $s$ at some configuration $C$ and let $ll^s$ contain an element with key $k$ in $C$. By inspection of the pseudocode (lines 9-10) we have that in that case, $m$ is not forwarded to a subsequent server.
Furthermore, by lines 12-16, we have that if \( s \) is not the token server and not \( s_{NS-1} \), and provided that \( ll^s \) does not contain an element with key \( k \), then \( s \) forwards \( m \) without modifying the \( mloop \) field. This implies that the \( mloop \) field of \( m \) is changed at most once in \( \alpha \) from \texttt{false} to \texttt{true}, and that by server \( NS-1 \), in a configuration \( C'' \) in which \( k \) is not contained in \( ll^{NS-1} \).

\[ \square \]

**Lemma 53.** Let \( C_i \), \( i \geq 0 \), be a configuration in \( \alpha \) in which server \( s_i \) is the token server. Let \( ll_i^j \) be the local list set of server \( s_j \), \( 0 \leq j < NS \), in \( C_i \). Then it holds that \( S_i = \bigcup_{j=0}^{NS-1} ll_i^j \).

**Proof.** We prove the claim by induction on \( i \).

**Base case (i = 0).** The claim holds trivially at \( C_0 \).

**Hypothesis.** Fix any \( i > 0 \) and assume that at \( C_i \), it holds that \( S_i = \bigcup_{j=0}^{NS-1} ll_i^j \). We show that the claim holds for \( i + 1 \).

**Induction step.** Let \( op_{i+1} \) be the operation that corresponds to the linearization point placed in \( C_{i+1} \). We proceed by case study.

Let \( op_{i+1} \) be an insert operation for key \( k \). Assume first that the linearization point of \( op_{i+1} \) is placed at the execution of line 9 by \( s_{t_{i+1}} \) for it. Notice that when this line is executed, \( k \) is searched for in the local list of \( s_{t_{i+1}} \). Recall that, by the way linearization points are assigned, the client \( c \) that invoked \( op_{i+1} \) receives \texttt{NACK} as response. Notice also that \( s_{t_{i+1}} \) sends \texttt{NACK} as a response to \( c \) if \( k \) is present in the local list of \( s_{t_{i+1}} \), and thus \( status_1 = \texttt{true} \). In that case, lines 12 to 27 are not executed, and therefore, no new element is inserted into the local list of \( s_{t_{i+1}} \) (line 22). Thus \( ll_{t_{i+1}}^{s_{t_{i+1}}} = ll_{t_{i+1}}^{s_{t_{i+1}}} \). By the induction hypothesis, \( S_i = \bigcup_{j=0}^{NS-1} ll_i^j \). By Observation 51 it follows that for any other server \( s_j \), where \( j \neq t_{i+1} \), \( ll_{t_{i+1}}^{s_{t_{i+1}}} = ll_i^j \) as well. Then, \( \bigcup_{j=0}^{NS-1} ll_{t_{i+1}}^{s_{t_{i+1}}} = \bigcup_{j=0}^{NS-1} ll_i^j \). Notice that since the server responds with \texttt{NACK}, \( S_{i+1} = S_i \) by definition. Thus, \( S_{i+1} = \bigcup_{j=0}^{NS-1} ll_{t_{i+1}}^j \) and the claim holds.

Now, assume that \( op_{i+1} \) is linearized at the execution of line 22 by the token server for it. By the way linearization points are assigned, this implies that when this line is executed, \( status_2 = \texttt{true} \), and the insertion of an element with key \( k \) into the local list of \( s_t \) was successful. This in turn implies that at \( C_{i+1} \), \( ll_{t_{i+1}}^{s_{t_{i+1}}} = ll_i^t \cup \{ k \} \). By Observation 51 it follows that for any other server \( s_j \), where \( j \neq t_{i+1} \), \( ll_{t_{i+1}}^{s_{t_{i+1}}} = ll_i^j \) as well. Notice that since the server responds with \texttt{ACK}, by definition the insertion is successful and thus \( S_{i+1} = S_i \cup \{ k \} \). Since by the induction hypothesis, \( S_i = \bigcup_{j=0}^{NS-1} ll_i^j \), it holds that \( S_{i+1} = \bigcup_{j=0}^{NS-1} ll_i^j \cup \{ k \} = \bigcup_{j=0}^{NS-1} ll_{t_{i+1}}^j \), thus, the claim holds.

Now consider that \( op_{i+1} \) is a delete operation for key \( k \). Assume first that some server \( s_\alpha \) responds with \texttt{ACK}, by executing line 36, to the client \( c \) that invoked \( op_{i+1} \). Then \( op_{i+1} \) is linearized at the execution of this line by \( s_\alpha \). Notice that this line is executed by a server if \( status_1 = \texttt{true} \), i.e. if the server was successful in locating and deleting an element with key \( k \) from its
local list. Thus, $ll_{i+1}^i = ll_i^i \backslash \{k\}$. Furthermore, by definition, $S_{i+1} = S_i \backslash \{k\}$.

By the induction hypothesis, $S_i = \bigcup_{j=0}^{NS-1} ll_j^i$ and since by Observation 51 no other modification occurred on the local list of some other server between $C_i$ and $C_{i+1}$, it follows that $S_{i+1} = S_i \backslash \{k\} = \bigcup_{j=0}^{NS-1} ll_j^i \backslash \{k\} = \bigcup_{j=0}^{NS-1} ll_j^{i+1}$.

Assume now that $op_{i+1}$ is a delete operation for which no server responds with ACK to the invoking client. Recall that in this case, by definition, $S_{i+1} = S_i$. By inspection of the pseudocode, it follows that no server finds an element with key $k$ in its local list when it is executing line 35 for $op_{i+1}$.

We examine two cases: (i) either no element with key $k$ is contained in any local list of any server in the beginning of the execution interval of $op_{i+1}$, or (ii) an element with key $k$ is contained in the local list of some server $s_d$ in the beginning of $op_{i+1}$’s execution interval, but $s_d$ deletes it while serving a different delete operation $op'$, before it executes line 35 for $op_{i+1}$.

Assume that case (i) holds. Then, the linearization point is placed in the beginning of the execution interval of $op_{i+1}$. Notice that in this case, the invocation (nor in fact the further execution) of $op_{i+1}$ has no effect on the local list of any server. Thus, between $C_i$ and $C_{i+1}$ no server local list is modified and, by the induction hypothesis, the claim holds.

Assume now that case (ii) holds. By Lemma 46, we have that a concurrent delete operation $op'$ removes the element with key $k$ from the local list of $s_d$ during the execution interval of $op_{i+1}$. By the assignment of linearization points, Observation 51 and Lemma 46, it further follows that $op' = op_i$. Notice that in this case (ii) also, $op_{i+1}$ has no effect on the local list of any server. Thus, since by the induction hypothesis it holds that $S_i = \bigcup_{j=0}^{NS-1} ll_j^i$, it also holds that $S_i = \bigcup_{j=0}^{NS-1} ll_j^{i+1}$, and since $S_i = S_{i+1}$, the claim holds.

Since a search operation does not modify the local list of any server, the argument is analogous as for the case of the delete operation.

From the above lemmas and observations, we have the following.

**Theorem 54.** The distributed unsorted list is linearizable. The insert operation has time and communication complexity $O(NS)$. The search and delete operations have communication complexity $O(1)$.

### 6.1.2 Alternative Implementation

At each point in time, there is a server (not necessarily always the same), denoted by $s_t$, which holds the insert token, and serves insert operations. Initially, server $s_0$ has the token, thus the first element to be inserted in the list is stored on server $s_0$. Further element insertions are also performed on it, as long as the space it has allocated for the list does not exceed a threshold. In case server $s_0$ has to service an insertion but its space is filled up, it forwards the token by sending a message to the next server, i.e. server $s_1$. Thus, if server $s_i$, $0 \leq i < NS$, has the token, but cannot service an insertion request without exceeding the threshold, it forwards the token to
server $s_{(i+1) \mod NS}$. When the next server receives the token, it allocates a memory chunk of size equal to threshold, to store list elements. When the token reaches $s_{NS-1}$, if $s_{NS-1}$ has filled all the local space up to a threshold, it sends the token again to $s_0$. Then, $s_0$ allocates more memory (in addition to the memory chunk it had initially allocated for storing list elements) for storing more list elements. The token might go through the server sequence again without having any upper-bound restrictions concerning the number of round-trips.

Event-driven code for the server is presented in Algorithm 29. Each server $s$ maintains a local list ($l_{list}$ variable) allocated for storing list elements, a $token$ variable which indicates whether $s$ currently holds the token, and a variable $round$ to mark the ring round-trips the token has performed; $round$ is initially 0, and is incremented after every transmission of the token to the next server. The pseudocode of the client is presented in Algorithm 30.

A client $c$ sends an insert request for an element with key $k$ to all servers in parallel and awaits a response. If any of the servers contains $k$ in its local list, it sends $ACK$ to $c$ and the insert operation terminates. If no server finds $k$, then all reply $NACK$ to $c$. In addition, the token server $s_t$ encapsulates its id in the $NACK$ reply. After that, $c$ sends an insert request for $k$ to $s_t$ only. If $s_t$ can insert it, it replies $ACK$ to $c$. If $k$ has in the meanwhile been inserted, $s_t$ replies $NACK$ to $c$. If $s_t$ is no longer the token server, it forwards the request along the server ring until it reaches the current token server. Servers along the ring should check whether they contain $k$ or not, and if some server does, then it replies $NACK$ to $c$. Let $s'_t$ be a token server that receives such a request. It also checks whether it contains $k$ or not. If not, it attempts to insert $k$ into its local list. Otherwise it replies $NACK$.

When attempting to insert the element in the local list, it may occur that the allocated space does not suffice. In this case, the server forwards the request as well as the token to the next server in the ring, and increments the value of $round$ variable. If the insertion at a token server is successful, the server then replies $ACK$ to $c$.

To perform a search for an element $e$, a client $c$ sends a search request to all servers and awaits their responses. A server $s$ that receives a search request, checks whether $e$ is present in its local part of the list and if so, it responds with $ACK$ to $c$. Otherwise, the response is $NACK$. If all responses that $c$ receives are $NACK$, $e$ is not present in the list. Notice that if $e$ is contained in the list, exactly one server responds with $ACK$. Delete works similarly: if a server $s$ responds with $ACK$, then $s$ has found and deleted $e$ from its local list. Given that communication is fast and the number of servers is much less than the total number of cores, forwarding a request to all servers does not flood the network.
6.2 Sorted List

The proposed implementation is based on the distributed unsorted list, presented in Section 6.1. Each server $s$ has a memory chunk of predetermined size where it maintains a part of the implemented list so that all elements stored on server $s_i$ have smaller keys than those stored on server $s_{i+1}$, $0 \leq i < N_S - 1$. Because of this sorting property, an element with key $k$ is not appended to the end of the list, so a token server is useless in this case. This is an essential difference with the unsorted list implementation.

Similarly to the unsorted case, a client sends an insert request for key $k$ to server $s_0$. The server searches its local part of the list for a key that is greater than or equal to $k$. In case that it finds such an element that is not equal to $k$, it can try to insert $k$ to its local list, $llist$. More specifically, if the server has sufficient storage space for a new element, it simply creates a new node with key $k$ and inserts it to the list. However, in case that the server does not have enough storage space, it tries to free it by forwarding a chunk of elements of $llist$ to the next server. If this is possible, it serves the request. In case $s_0$ does not find a key that is greater than or equal to $k$ in its $llist$, if forwards the message with the insert request to the next server, which in turn tries to serve the request accordingly. Notice that this way, a request may be forwarded from one server to the next, as in the case of the unsorted list. However, for ease of presentation, in the following we present a static algorithm where this forwarding stops at $s_{N_S-1}$. In case that an element with $k$ is already present in the $llist$ of some server $s$ of the resulting sequence, then $s$ sends an NACK message to the client that requested the insert.

As in the case of the unsorted list, a client performs a search or delete operation for key $k$ by sending the request to all servers. If not handled correctly, then the interleaving of the arrival of requests to servers may cause a search operation to “miss” the key $k$ that it is searching, because the corresponding element may be in the process to be moved from one server to a neighboring one. In order to avoid this, servers maintain a sequence number for each client that is incremented at every search and delete operation. Neighboring servers that have to move a chunk of elements among them, first verify that the latest (search or delete) requests that they have served for each client have compatible sequence numbers and perform the move only in this case.

Event-driven code for the server is presented in Algorithms 31 and 32. The clients access the sorted list using the same routines as they do in the case of the unsorted list (see Algorithm 28).

When an insert request for key $k$ reaches a server $s$, $s$ compares the maximal key stored in its local list to $k$. If $k$ is greater than the maximal key and $s$ is not $s_{N_S-1}$, the request must be forwarded to the next server (line 117). Otherwise, if $k$ is to be stored on $s$, $s$ checks if $llist$ has enough space to serve the insert. If it does, $s$ inserts the element and sends an ACK to
the client (line 105-106). If \( s \) does not have space for inserts, the operation cannot be executed, hence \( s \) must check whether a chunk of its elements can be forwarded to the next server to make room for further inserts. To move a chunk, \( s \) calls \texttt{ServerMove()} (presented in Algorithm 32) (line 110). If \texttt{ServerMove()} succeeds in making room in \( s \)'s \texttt{llist}, the insert can be accommodated (line 111). In any other case, \( s \) responds to the client with \texttt{NACK} (line 114).

A server process a search request as described for the unsorted list, but it now pairs each such request with a sequence number (line 122). Delete is processed by a server in a way analogous to search.

In order to move a chunk of \texttt{llist} to the next server, a server \( s_i \) invokes the auxiliary routine \texttt{ServerMove()} (line 110). \texttt{ServerMove()} sends a \texttt{REQC} message to server \( s_{i+1} \) (line 138). When \( s_{i+1} \) receives this request, it sends its client vector to \( s_i \) (line 88). Upon reception (line 139), \( s_i \) compares its own client vector to that of \( s_{i+1} \) and as long as it lags behind \( s_{i+1} \) for any client, it services search and delete requests until it catches up to \( s_{i+1} \) (lines 140-142). Notice that during this time, \( s_{i+1} \) does not serve further client request, in order allow \( s_i \) to catch up with it. As soon as \( s_i \) and \( s_{i+1} \) are compatible in the client delete and search requests that they have served, \( s_i \) sends to \( s_{i+1} \) a chunk of the elements in its local list (lines 143-144) and awaits the response of \( s_{i+1} \). We remark that in order to perform this kind of bulk transfer, as the one carried out between a server executing line 145 and another server executing line 89, we consider that remote DMA transfers are employed. This is omitted from the pseudocode for ease of presentation.

If \( s_{i+1} \) can store the chunk of elements, then it does so and sends \texttt{ACK} to \( s_i \). Upon reception, \( s_i \) may now remove this chunk from its local list (line 111) and attempt to serve the insert request. Notice that if \( s_{i+1} \) cannot store the chunk of elements of \( s_i \), then it itself initiates the same chunk moving procedure with its next neighbor (lines 92-94), and if it is successful in moving a chunk of its own, then it can accommodate the chunk received by \( s_i \). Notice that in the static sorted list that is presented here, this protocol may potentially spread up to server \( s_{NS-1} \) (line 91). If \( s_{NS-1} \) does not have available space, then the moving of the chunk fails (line 113). The client then receives a \texttt{NACK} response, corresponding to a full list.

We remark that this implementation can become dynamic by appropriately exploiting the placement of the servers on the logical ring, in a way similar to what we do in the unsorted version.
Algorithm 32 Auxiliary routine ServerMove for the servers of the distributed sorted list.

```java
boolean ServerMove(int cid, data chunk1) {
    boolean status;
    data chunk2;
    send(next_id, (REQC, cid, 0, ⊥));
    nbr_cv = receive(next_id);
    while (for any element i, cv[i] < nbr_cv[i]) {
        receiveMessageOfType(SEARCH or DELETE);
        service request
    }
    chunk2 = getChunkOfElementsFromLocalList(llist);
    send(next_id, chunk2);
    status = receive(next_id);
    if (status == true) {
        removeChunkOfElementsFromLocalList(llist, chunk2);
        insertChunkOfElementsInLocalList(llist, chunk1);
        return true;
    } else return false;
    }
```
Algorithm 29 Events triggered in a server of the distributed unsorted list.

1. List $llist = \emptyset$;
2. int $my_id$, $next_id$, $token = 0$, $round = 0$;

3. a message $(op, cid, key, data, tk)$ is received:
4. switch $(op)$ {
5.     case INSERT:
6.         if $(tk == TOKEN)$ {
7.             $token = my_id$;
8.             allocate_new_memory_chunk$(llist, round)$;
9.         }
10.        $status_1 = search(llist, key)$;
11.        if $(tk == -2)$ {
12.            if $(status_1)$ {
13.                if $(token == my_id)$ send$(cid, (ACK, true))$;
14.                else send$(cid, (ACK, false))$;
15.            } else {
16.                if $(token == my_id)$ send$(cid, (NACK, true))$;
17.                else send$(cid, (NACK, false))$;
18.            }
Algorithm 30 Insert, Search and Delete operation for a client of the distributed list.

```java
41 boolean ClientInsert(int cid, int key, data data) {
42     boolean status;
43     boolean found = false;
44     int tid;
45     send_to_all_servers(⟨INSERT, cid, key, ⊥, -2⟩);
46     do {
47         ⟨status, sid, is_token⟩ = receive();
48         if (status == ACK) found = true;
49         if (is_token) tid = sid;
50         c++;
51     } while (c < NS);
52     if (found == true) return false;
53     send(tid, ⟨INSERT, cid, key, data, -1⟩);
54     status = receive();
55     if (status == NACK) return false;
56     else return true;
57 }
58 boolean ClientSearch(int cid, int key) {
59     int sid;
60     int c = 0;
61     boolean status;
62     boolean found = false;
63     send_to_all_servers(⟨SEARCH, cid, key, ⊥, -1⟩);
64     do {
65         ⟨status, sid⟩ = receive();
66         if (status == ACK) found = true;
67         c++;
68     } while (c < NS);
69     return found;
70 }
71 boolean ClientDelete(int cid, int key) {
72     int sid;
73     int c = 0;
74     boolean status;
75     boolean deleted = false;
76     send_to_all_servers(⟨DELETE, cid, key, ⊥, -1⟩);
77     do {
78         ⟨status, sid⟩ = receive();
79         if (status == ACK) deleted = true;
80         c++;
81     } while (c < NS);
82     return deleted;
83 }
```
Algorithm 31 Events triggered in a server of the distributed sorted list.

1. List $llist = \emptyset$;
2. int $my\_id$, $next\_id$, $k_{max}$, $cv[MC]$, $nbr\_cv[MC]$;
3. data[0...CHUNKSIZE] chunk1, chunk2;
4. boolean status = false, served = false;

A message $(op, cid, key, data)$ is received:

switch $(op)$ {

  case REQC:
  send(cid, cv);
  chunk2 = receive(cid);
  if (not enough free space in local list to fit elements of chunk2) {
    if ($my\_id == NS - 1$) status = false;
    else {
      chunk1 = getChunkOfElementsFromLocalList(llist);
      status = ServerMove($next\_id$, chunk1);
    }
  } else status = true;
  if (status == true) {
    insertChunkOfElementsInLocalList(llist, chunk2);
    send(cid, ACK);
  } else send(cid, NACK);
  break;

  case INSERT:
  while (served \neq true) {
    $k_{max} = find\_max(llist)$;
    if ($k_{max} > key$ and isFull(llist) \neq true) {
      status = insert(llist, key, data);
      served = true;
    } else if ($k_{max} > key$) {
      chunk1 = getChunkOfElementsFromLocalList(llist);
      status = ServerMove($next\_id$, chunk1);
      if (status == true) {
        removeChunkOfElementsFromLocalList(llist, chunk1);
        send(cid, NACK);
        served = true;
      }
    } else {
      if ($my\_id \neq NS - 1$) send($next\_id$, (INSERT, cid, key, data));
      else send(cid, NACK);
      served = true;
    }
  }
  break;

  case SEARCH:
  $cv[cid]++$;
  status = search(llist, key);
  if (status == false) send(cid, NACK);
  else send(cid, ACK);
  break;

  case DELETE:
  $cv[cid]++$;
  status = search(llist, key);
  if (status == true) {
    delete(llist, key);
    send(cid, ACK);
  } else send(cid, NACK);
  break;

}
Chapter 7

Distributed Search Tree

In this section, we present a leaf-oriented distributed (2,4)-tree. An (2,4)-tree structure is a perfect balanced tree where the number of children of an internal node is between 2 and 4. In a leaf-oriented (2,4) tree, the data are stored only in the leaves of the tree, whereas the internal nodes are used for indexing. Leaf nodes may store up to 3 keys. Such a tree provides three operations for accessing and updating the stored data, namely insert, delete and search. The insert operation inserts a key into the tree, either by appending it to an existing node or by creating a new node and inserting it to the structure. The delete operation removes a key from the tree and the search operation detects whether a given key is present in the tree. Notice that insert and delete operations must maintain the ordering property of the (2,4)-tree, as well as equal depth for all leaves.

We define as a distributed (2,4)-tree the data structure that maintains the attributes of the (2,4)-tree but allows the operations to be initiated by different clients and to be executed concurrently. For presentation purposes, we assume that each server maintains only one node of the (2,4)-tree (either an internal or a leaf node).

Initially only the root node of the (2,4)-tree is allocated, which is known to all clients at the beginning. Notice that in some execution \( \alpha \), the root node of the \((a,b)\)-tree is maintained by the same sever in any configuration of \( \alpha \). Whenever a client wants to execute an insert, delete or search operation to the simulated tree, it sends an INSERT, DELETE or SEARCH message, respectively, to the root server of the tree.

The local variables of a node \( r \) are presented in Algorithm 33. Each node stores the number of the stored keys (size variable), the key values \( (k_0 - k_2 \) variables) and pointers to its four children. Also each node \( r \) stores boolean flags to distinguish if it is a root node or a leaf node (isRoot and isLeaf variables).

INSERT, DELETE or SEARCH messages have similar structure. More specifically, they contain the following fields: (1) \( op \) that specifies the operation, (2) \( sender \) that contains the id of the node that sent the request
Algorithm 33 Private variables and data structures on a node of a (2,4)-tree.

1 int size = 0;
2 int k0, k1, k2; /* node’s keys */
3 int child0, child1, child2, child3; /* node’s children */
4 boolean isRoot = true;
5 boolean isLeaf = true;

(either a client or a node server), (3) the key to be inserted, and (4) cid filed, with the client’s id.

The messages that are transmitted between a pair of nodes \((r_x, r_y)\) in order to accommodate rotations and key transfers, have different structure. Their fields contain: (1) an ack field to define the type of message \((\text{DONE}, \text{SPLIT}, \text{EMPTY}, \text{MERGE}, \text{NEW_NODE}, \text{FUSION}, \text{LOAN}, \text{BIG_LOAN})\), (2) key that contains information whenever a key is transferred, and (3) ch1 and ch2 that store one or two whole children, in case of a large data movement. Their usage is going to be described in more detail in the paragraphs below.

To keep the tree balance, we apply preventive splitting during inserts and preventive merging during deletes in a top-down approach (from the root to the leaf). To do so, we introduce the concepts of insert- (delete-) safe nodes. A node is insert-safe if it has less than 4 children; it is insert-unsafe otherwise. A node is delete-safe if it has more than 2 children; it is delete-unsafe otherwise. A node is always search-safe.

Initially, there is only the root node of the tree, stored in some server \(s\). A client sends insert, delete or search requests to \(s\), and then awaits for the response. Each request is forwarded from a server to the server storing the next node on the path that leads to the appropriate leaf. During INSERT and DELETE a node \(r\) (root included) before executing the operation, it is essential to clarify its state. If \(r\) is in unsafe state, it must communicate with its parent in order to overcome it (i.e. preventing splitting is applied in the case of an insert-unsafe node and preventing merging is applied in the case of a delete-unsafe node), and then proceed with the execution of the operation.

Event driven code of a node \(r\) is described in Algorithms 34 to 40. We first describe the insert operation, presented in Algorithm 34. Whenever a node \(r\) receives an INSERT message, it first examines the number of its stored elements. If its size is less than 4, the node is in safe mode and there is no need for preventing splitting. In case that \(r\) is a leaf node (line 7), it responds to its parent node \((\text{sender} \text{ in the message})\) with a DONE message (line 8) and it tries to add the new key that was sent with the INSERT message to the set of keys that it maintains (line 9). If the insert was successful, the size of keys is increased by one, and \(r\) sends an ACK to the client. Otherwise, \(r\) sends a NACK message to the client. When the operation is completed, \(r\) returns (line 12). In case that \(r\) is an internal node (line 13), it responds to
its parent node with a DONE message (line 14) and chooses the appropriate child to forward the message according to the number of its keys (lines 15-22). The DONE message towards the parent, in both cases, gives the ability to the parent to unblock and process its next message.

If the size of \( r \) is equal to 4, \( r \) is in unsafe mode which requires preventing splitting of its elements in to two nodes. If \( r \) is not the root node (line 23), it calls function AllocateNewNode() in order to get a new node id, which is going to host the half of \( r \)'s elements with the biggest key values. Then, \( r \) sends to the new node a NEW_NODE to wake the newly allocate node up and then a SPLIT message to inform it about the splitting, and blocks waiting for its response (lines 24-27). In the SPLIT message, \( r \) includes the data that the new node should have, i.e. two children and a key if the new node is an internal node or just a key in case that the new node is a leaf. When the new node \( nnode \) responds, \( r \) sends a SPLIT message to its parent to inform it that it must store a new child (line 32), and returns. The SPLIT message contains the id of the newly allocated node and \( r \)'s previous key, \( k_1 \). Assume first that \( r \) is an internal node. Then, it forwards the INSERT message to one of its children, or to the new node (\( nnode \)) depending on the new key’s value. If the new key is less than \( k_0 \) it is forwarded to \( r \)'s left child (line 34). If the new key is less than \( k_1 \) it is forwarded to \( r \)'s right child (line 35). Otherwise it is forwarded to \( nnode \) (line 36). Assume now that \( r \) is a leaf (line 28). Then \( r \) it tries to store the new key.

If \( r \) is the root node (line 40) and it is in unsafe-insert mode, the height of the tree should be increased by one. Since the root node must always be hosted by the same server, \( r \) allocates two new nodes that are going

---

**Algorithm 34** Insert operation on a (2,4)-tree node.

```plaintext
6    node nd receives message ⟨INSERT, sender, key, cid⟩:
7    if (size < 4 AND isLeaf == true) {
8        send(sender, ⟨DONE, ⊥, ⊥, ⊥, ⊥⟩) ; /* Node is insert safe */
9        status = insert(key);
10       if (status) send(cid, ⟨ACK⟩);
11       else send(cid, ⟨NACK⟩);
12       return;
13    } else if (size < 4 AND isLeaf == false) {
14        send(sender, ⟨DONE, ⊥, ⊥, ⊥, ⊥⟩);
15        if (size == 2) {
16            if (key ≤ k_0) child = child_0;
17            else child = child_1;
18        } else if (size == 3) {
19            if (key ≤ k_0) child = child_0;
20            else if (key ≤ k_1) child = child_1;
21            else child = child_2;
22        } send(child, ⟨INSERT, nd, key, cid⟩);
23    }
```

95
```c
else if (size == 4 AND isRoot == false) { /* Node is insert unsafe */
    nnode = AllocateNewNode();
    send(nnode, (NEW_NODE, nd, ⊥, ⊥));
    send(nnode, (SPLIT, k2, ⊥, child2, child3));
    response = receive(nnode);
    if (isLeaf == true AND key < k0) {
        status = insert_key(key);
        if (status) send(cid, (ACK));
        else send(cid, (NACK));
        send(sender, (SPLIT, k1, ⊥, nd, nnode));
        return;
    }
    if (isLeaf == false AND key < k0) child = child0;
    else if (isLeaf == false AND key < k1) child = child1;
    else if (key > k1) child = nnode;
    size = 2;
    send(child, (INSERT, nd, key, cid));
    send(sender, (SPLIT, k1, ⊥, nd, nnode));
} else if (size == 4 and isRoot == true) {
    lnode = AllocateNewNode();
    rnode = AllocateNewNode();
    send(lnode, (NEW_NODE, nd, ⊥, ⊥));
    send(lnode, (SPLIT, k0, ⊥, child0, child1));
    response = receive(lnode);
    send(rnode, (NEW_NODE, nd, ⊥, ⊥));
    send(rnode, (SPLIT, k2, ⊥, child2, child3));
    response = receive(rnode);
    size = 2;
    k0 = k1;
    child0 = lnode;
    child1 = rnode;
    isLeaf = false;
    if (key > k0) child = child1;
    else child = child0;
    send(child, (INSERT, nd, key, cid));
} response = receive(child);
if (response.ack == DONE) return;
else { /* A node split should occur */
    insert_key(response.key);
    add response.ch1 and response.ch2 to children;
}
```
to store two of its keys (lines 41-42). The left new node is going to store the smallest key, the right new node is going to store the largest key and r keeps for itself the middle key. Thus, r sends a NEW_NODE message to both of the new nodes to wake them up, and a SPLIT message to each of them with the appropriate keys and children, and blocks waiting for their response (lines 24-27). After updating its local variables, r forwards the INSERT message to the appropriate node (lines 54-56).

The delete operation is presented in Algorithm 35. Whenever a node r receives a DELETE message, it first examines if its size is equal to 2 (line 63), which if is true, the node is in unsafe state. If r is in unsafe state, it sends an EMPTY message to the parent node (line 64). When the parent node receives this type of message, it tries to change the state of its child node to safe based on the following rules:

- The parent of r is the root, and none of r’s neighbors have more than 2 children. The root responds with a FUSION message to r, and fuses its node with its children’s nodes, reducing the height of the tree by one.
In this case r moves its two children to its parent (line 66) and frees the node, because it is going to be absorbed by the parent (line 68). Then, the root node adds them to its children and keys (lines 100-102). Then it informs its other child with a regular FUSION message (presented in Algorithm 39) that it is going to be absorbed (line 104). After the second child answers with its data, the root node adds them with its local data (lines 105-107). The fusion operation reduces the tree’s height by one, since the root absorbs its two children. Because the root’s children were updated, the root node continues to execute at label delete_start in order to execute the delete operation (line 108).

- One of r’s siblings contain more than two children. The parent gets a pair of a key and a value from that child by sending a MOVE message and forwards the pair of the key and the value to r with a LOAN message. This is plausible, because the parent node is in safe state. So, the parent node answers with a LOAN response to r and sends one of its keys (lines 111). In this case r adds the new key and the new child to its own (lines 71-72).

- The parent of r is not the root node, and none of r’s neighbors have more than 2 children. The parent has more than two children, so it gives r a key and a child of its own with a BIG_LOAN message in order to force r to change into safe state.
If r receives a BIG_LOAN response (line 73), it assimilates the whole sibling and its data it received from the response with its own.
The parent node sends a BIG_LOAN to its child (line 112) in order to merge it with the sibling with the smallest number of keys. So, the parent node sends to the neighbor of r that is going to be merged a message MERGE, in order for the child to transmit its key and children with a response message (line 113-116). When the parent node receives
the response from its child, it sends it with another message `BIG_LOAN` to `r` (lines 117-0). In other words, this operation is a rotation, where one of the children sends a key to the parent of `r`, and the parent of `r` sends a key and two children to `r` in order to apply the rotation.

In case that `r` is in safe state or after becoming safe, it sends a `DONE` message to its parent (line 76). Then, if `r` is a leaf node (line 78), it proceeds with serving the `DELETE` operation. It deletes its key if it exists, and according to `delete_key`'s function, it returns the appropriate message to the client (lines 79-82). Otherwise, `r` forwards the message the appropriate child (lines 83-84) and blocks waiting for its child node to respond. Depending on the response, `r` might need to handle the case of its child being in unsafe state, in a similar way as described above (lines 74-75).

The search operation is straightforward. It performs exactly like a binary tree search. The operation starts from the root and is forwarded downwards by visiting the internal nodes, directed by the keys of the internal nodes (lines 131)-132). If the operation reaches a leaf node (line 126), it checks whether the key exists in its saved keys. If it does, `r` sends an `ACK` message to the client (line 127), otherwise it sends a `NACK` (line 128) and it returns.

Algorithm 37 presents the actions of a node `r` when it receives a message `NEW_NODE`. This message is sent by a node when it wants to split its data and transmit them to a new sibling node, which is going to be `r`. Hence, `r` takes the data transmitted inside the message, which are the key and two children, and adds them to its local variables `key0, child0` and `child1`. It also initializes the two flags `isLeaf` and `isRoot` with `false`, and returns a `DONE` message to the sender.

Algorithm 38 presents the actions of a node `r` when it receives a message `MOVE`. Such a message is sent during a `DELETE` operation, so `r` after

---

**Algorithm 35** Delete operation on a (2,4)-tree node.

```
62 node nd receives message (DELETE, sender, key, cid):
63     if (size == 2 AND isRoot == false) { /* r in unsafe mode */
64         send(sender, (EMPTY, k0, ⊥, child0, child1));
65         response = receive(sender);
66         if (response.ack == FUSION) {
67             send(sender, (DONE, ⊥, ⊥, ⊥, ⊥));
68             FREE_NODE(nd);
69             return;
70         } else if (response.ack == LOAN) {
71             insert_key(response.key);
72             add response.ch0 to children;
73         } else if (response.ack == BIG_LOAN) {
74             add response.ch1, response.ch2 to children;
75             insert_key(response.key);
76         }
77     }
```
if (isRoot == false) send(sender, ⟨DONE, ⊥, ⊥, ⊥, ⊥⟩);
else send(cid, DONE);

delete_start:
/* nd in safe state */

if (isLeaf == true) {
    status = delete_key(key);
    if (status) send(cid, ⟨ACK⟩);
    else send(cid, ⟨NACK⟩);
    return;
}

i = 0; while (i < size-1 and key > ki) i++;
send(childi, ⟨DELETE, nd, key, cid⟩);
response = receive(childi);
if (response.ack == EMPTY) {
    int chnode = 0, chkey;
    if (i ≠ 0) {
        send(childi−1, ⟨MOVE, nd, LEFT, ⊥⟩);
        response = receive(childi−1);
        chnode = response.ch0;
        chkey = response.key;
    }
    if (i ≠ size AND chnode == 0) {
        send(childi+1, ⟨MOVE, nd, RIGHT, ⊥⟩);
        response = receive(childi+1);
        chnode = response.ch0;
        chkey = response.key;
    }
    if (chnode == 0 and isRoot == true) /* Parent node fuses all of its children */ {
        send(childi, ⟨FUSION, ⊥, ⊥, ⊥, ⊥⟩);
        response = receive(childi);
        add response.ch0, response.ch1 to children;
        add response.key to keys;
        for each children j except i {
            send(childj, ⟨FUSION, nd, ⊥, ⊥⟩);
            response = receive(childj);
            add response.ch0, response.ch1 to children;
            add response.key to keys;
        }
        goto delete_start;
    } else if (chnode ≠ 0) {
        send(childi, ⟨LOAN, ki, ⊥, chnode, ⊥⟩);
        ki = chkey;
    } else {
        if (i ≠ 0) child = childi−1;
        else child = childi+1;
        send(child, ⟨MERGE, ⊥, ⊥, ⊥⟩);
        response = receive(child);
        if (i ≠ 0) {
            send(child, ⟨BIG_LOAN, ki−1, response.key, response.ch0, response.ch1⟩);
            delete_key(ki−1);
            delete childi−1 form keys and children;
        } else {
            send(child, ⟨BIG_LOAN, ki+1, response.key, response.ch0, response.ch1⟩);
            delete_key(ki+1);
            delete childi+1 form keys and children;
        }
    }
} } }
Algorithm 36 Search operation on a (2,4)-tree node.
125 node nd receives message (SEARCH, ⊥, key, cid):
126     if (isLeaf == true) {
127         if (key ∈ {k₀,...,kₙsize}) send(cid, (ACK));
128         else send(cid, (NACK));
129         return;
130     }
131     i = 0;
132     while (i < nsize-1 and key > kᵢ) i++;
133     send(childᵢ, (SEARCH, ⊥, key, cid));

Algorithm 37 NEW_NODE message on a node of a (2,4)-tree.
133 node nd receives message (NEW_NODE, sender, ⊥, ⊥)
134     response = receive(sender);
135     k₀ = response.key;
136     child₀ = response.ch₀;
137     child₁ = response.ch₁;
138     size = 1;
139     isLeaf = false;
140     isRoot = false;
141     send(sender, (DONE, ⊥, ⊥, ⊥, ⊥));

Algorithm 38 MOVE message on a node of a (2,4)-tree.
142 node nd receives message (MOVE, sender, pos, ⊥):
143     if (size > 2) {
144         if (pos == LEFT) {
145             send(sender, (DONE, k₀, ⊥, child₀, ⊥));
146             for (i = 0; i < size; i++)
147                 kᵢ = kᵢ+1, childᵢ = childᵢ+1;
148         } else send(sender, (DONE, kₙsize-1, ⊥, childₙsize, ⊥));
149         size--;
150     } else {
151         send(sender, (NA, ⊥, ⊥, ⊥, ⊥));
152     }
receiving it, it must check its state. If it is in unsafe state it returns a message NA (not available). This will inform the sender that r has only one key stored. Otherwise, r, depending on its position (declared by pos field in message), sends the largest or smallest key to the sender. The sender in such a situation is r’s parent, which will store the key as its own.
Algorithm 39 FUSION message on a node of a (2,4)-tree.

node \( nd \) receives message (FUSION, sender, \( \bot, \bot \))

send(sender, (DONE, \( k_0, \bot, ch_0, ch_1 \)));

FREE_NODE(nd);

return;

Algorithm 40 MERGE message on a node of a (2,4)-tree.

node \( nd \) receives message (MERGE, sender, \( \bot, \bot \))

send(sender, (DONE, \( k_0, \bot, ch_0, ch_1 \)));

FREE_NODE(nd);

return;

When a node \( r \) receives a regular message of type FUSION or MERGE (Algorithm 39 and Algorithm 40), it must send all of its data to the sender and then be removed from the tree structure. It is guaranteed that \( r \) has only one key and two children, so it sends its children and the key to the sender that requested them. Then calls function FREE_NODE() to be removed from the structure.
Chapter 8

Details on Hierarchical Approach

We interpolate one or more communication layers by using intermediate servers between the servers that maintain parts of the data structure and the clients. The number of intermediate servers and the number of layers of intermediate servers between clients and servers can be tuned for achieving better performance.

For simplicity of presentation, we focus on the case where there is a single layer of intermediate servers. We present first the details for a fully non cache-coherent architecture.

For each island $i$, we appoint one process executing on a core of this island, called the island master (and denoted by $m_i$), as the intermediate server. Process $m_i$ is responsible to gather messages from all the other cores of the island, and batch them together before forwarding them to the appropriate server. This way, we exploit the fast communication between the cores of the same island and we minimize the number of messages sent to the servers by putting many small messages to one batch. We remark that each batch can be sent to a server by performing DMA. In this case, $m_i$ initiates the DMA to the server’s memory, and once the DMA is completed, it sends a small message to the server to notify it about the new data that it has to process.

Algorithm 41 presents the events triggered in an island master $m_i$ and its actions in order to handle them. $m_i$ receives messages from clients that have type OUT (outgoing messages) and from servers that have type IN (incoming messages). The outbatch messages are stored in the $outbuf$ array. Each time a client from island $i$ wants to execute an operation, it sends a message to $m_i$; $m_i$ checks the destination server id ($sid$), recorded in the message, and packs this message together with other messages directed to $sid$ (lines 4-6). Server $m_i$ has set a timer, and it will submit this batch of messages to server $sid$ (as well as other batches of messages to other servers), when the timer expires. When $m_i$ receives an incoming message from a server, it unpacks
Algorithm 41 Events triggered in an island master - Case of fully non cache-coherent architectures.

1. LocalArray outbuf = ∅; /* stores outgoing messages */
2. LocalArray inbuf = ∅; /* stores incoming messages */

3. a message \((type, msg)\) is received:
   4. if\((type == OUT)\) {
      5.   sid = read field sid from msg;
      6.   add_message(outbuf, sid, msg);
   }
   7. else if \((op == IN)\) { /* \(m_i\) received a batch of messages from a server */
      8.   inbuf = split(msg); /* unbundle the message to many small ones */
      9.   for each message \(m\) in inbuf { /* the batch is for all cores in the island */
         10.  cid = read field client from msg;
         11.  send(cid, msg);
      }
   }

12. timer is triggered: /* Every timeout, \(m_i\) sends outgoing messages */
13. for each batch of messages in outbuf { /* send a batch to a server */
      14.  send(sid, batch); /* this send can be done using DMA */
      15.  delete(outbuf, sid, batch);
   }

it, and sends each message to the appropriate client on its island. When the timer is triggered, \(m_i\) places each batch of messages in an outbuf array that \(m_i\) maintains for the appropriate server. The transfer of all these messages to the server may occur using DMA. For simplicity, we use two auxiliary functions: add_message to add a message to an outbuf buffer of \(m_i\), and split to split a batch of messages msg that have arrived to the particular messages of the batch which are then placed in the inbuf buffer of \(m_i\).

The code of the client does not change much. Instead of sending messages directly to the server, it sends them to the board master.

We remark that in order to improve the scalability of a directory-based algorithm, a locality-sensitive hash functions could be a preferable choice. For instance, the simple currently employed mod hash function, can be replaced by a hash function that divides by some integer \(k\). Then, elements of up to \(k\) subsequent insert (i.e. push or enqueue) operations may be sent to the same directory server. This approach suites better to bulk transfers since it allows for exploiting locality. Specifically, consider that \(m_i\) sends a batch of elements to be inserted to the synchronizer \(s_s\) of a directory-based data structure. Server \(s_s\) unpacks the batch and processes each of the requests contained therein separately. Thus, if mod is used, each of these elements will be stored in different buckets (of different directory
servers). As a sample alternative, if the div hash function is used, more than one elements may end up to be stored in the same bucket. When later on a batch of remove (i.e. pops or dequeues) operations arrives to \( s_s \), it can request from the directory server that stores the first of the elements to be removed, to additionally remove and send back further elements with subsequent keys that are located in the same bucket. Notice that in this case, the use of DMA can optimize these transfers.

We now turn attention to partially non cache-coherent architectures where the cores of an island communicate via cache-coherent shared memory. Algorithm 42 presents code for the hierarchical approach in this case. For each island, we use an instance of the CC-Synch combining synchronization algorithm, presented in [18]. All clients of island \( i \) participate to the instance of CC-Synch for island \( i \), i.e. each such client calls CC-Synch (see Algorithm 42) to execute an operation.

CC-Synch employs a list which contains one node for each client that has initiated an operation; the last node of the list is a dummy node. After announcing its request by by recording it in the last node of the list (i.e., in the dummy node) and by inserting a new node as the last node of the list (which will comprise the new dummy node), a client tries to acquire a global lock (line 20) which is implemented as a queue lock. The client that manages to acquire the lock, called the combiner, batches those active requests, recorded in the list, that target the same server (lines 28-32) and forwards them to this server (line 33). Thus, at each point in time, the combiner plays the role of the island master. When the island master receives (a batch of) responses from a server, it records each of them in the appropriate element of the request list to inform active clients of the island about the completion of their requests (lines38-44). In the meantime, each such client performs spinning (on the element in which it recorded its request) until either the response for its request has been fullfilled by the island master or the global lock has been released (line 24).

We use the list of requests to implement the global lock as a queue lock [35, 36]. The process that has recorded its request in the head node of the list plays the role of the combiner.
Algorithm 42: Pseudocode of hierarchical approach - Case of partially non-cache-coherent architectures.

```c
struct Node {
    Request req;
    RetVal ret;
    int id;
    boolean wait;
    boolean completed;
    int sid;
    Node *next;
};

shared Node *Tail;
private Node *node_i;
LocalArray outbuf = ∅; /* stores outgoing messages */

RetVal CC-Synch(Request req) {
    /* Pseudocode for thread p_i */
    Node *nextNode, *tmpNode, *tmpNodeNext;
    int counter = 0;
    node_i→wait = true;
    node_i→next = null;
    node_i→completed = false;
    nextNode = node_i;
    node_i→req = req; /* p_i announces its request */
    node_i→sid = destination server;
    node_i→next = nextNode;
    while (node_i→wait == true) /* p_i spins until it is unlocked */
        nop;
    if (node_i→completed==true) /* if p_i’s req is already applied */
        return node_i→ret; /* p_i returns its return value */
    tmpNode = node_i; /* otherwise p_i is the combiner */
    while (tmpNode→next ≠ null AND counter < h){
        counter = counter + 1;
        tmpNodeNext=tmpNode→next;
        add_message(outbuf, tmpNode→sid, tmpNode→req);
        tmpNode = tmpNodeNext; /* proceed to the next node */
    }
    for each batch of messages in outbuf { /* send a batch of messages to servers */
        send(sid, batch of message); /* where sid is the destination server for this batch */
        delete(outbuf, sid, fatm);
    }
    inbuf = split(receive());
    tmpNode = node_i;
    while (counter ≥ 0) {
        counter = counter - 1;
        tmpNodeNext=tmpNode→next;
        tmpNode→ret = find in inbuf the response for tmpNode→sid;
        tmpNode→completed = true;
        tmpNode→wait = false; /* unlock the spinning thread */
        tmpNode = tmpNodeNext;
    }
    tmpNode→wait = false; /* unlock next node’s owner */
    return node_i→ret;
}
```
Chapter 9

Experimental Evaluation

We run our experiments on the Formic-Cube [4], which is a hardware prototype of a 512 core, non-cache-coherent machine. It consists of 64 boards with 8 cores each (for a total of 512 cores). Each core owns 8 KB of private L1 cache, and 256 KB of private L2 cache. None of these caches is hardware coherent. The boards are connected with a fast, lossless packet-based network forming a 3D-mesh with a diameter of 6 hops. Each core is equipped with its own local hardware mailbox, an incoming hardware FIFO queue, whose size is 4 KB. It can be written by any core and read by the core that owns it. One core per board plays the role of the island master (and could be one of the algorithm’s servers), whereas the remaining 7 cores of the board serve as clients.

Our experiments are similar to those presented in [37, 18, 12]. More specifically, $10^7$ pairs of requests (PUSH and POP or ENQUEUE and DEQUEUE) are executed in total, as the number of cores increases. To make the experiment more realistic, a random local work (up to 512 dummy loop iterations) is simulated between the execution of two consecutive requests by the same thread as in [37, 18, 12]. To reduce the overheads for the memory allocation of the stack nodes, we allocate a pool of nodes (instead of allocating one node each time).

In Figure 9.1.a, we experimentally compare the performance of the centralized queue (CQueue) to the performance of its hierarchical version (HQueue), and to those of the hierarchical versions of the directory-based queue (DQueue) and the token-based queue (TQueue). We measure the average throughput achieved by each algorithm. As expected, CQueue does not scale well. Specifically, the experiment shows that for more than 16 cores in the system, the throughput of the algorithm remains almost the same. We remark that the clients running on these 16 cores do not send enough messages to fill up the mailbox of the server. This allows us to conclude that when the server receives about 16 messages or more, for reading these messages, processing the requests they contain, and sending back the responses to clients, the server ends up to be always busy. We remark that reading each message...
from the mailbox causes a cache miss to the server. So, the dominant factor at the server side in this case is to perform the reading of these messages from the mailbox.

These remarks are further supported by studying the experiment for the HQueue implementation. Specifically, the throughput of HQueue does not further increase when the number of cores becomes 64 or more. Remarkably, the 64 active cores are located in 8 boards, so there exist 8 island masters in the system. Each island master sends two messages to the server – one for batched enqueue and one for batched dequeue requests. So, again, the server becomes saturated when it receives about 16 messages. These messages are read from the mailbox in about the same time as in the setting of CQueue with 16 running cores. However, in the case of HQueue, each message contains more requests to be processed by the server. Therefore, the average time needed to process a request is now smaller since the overhead of reading and processing a message is divided over the number of requests it contains. Notice that now, the server has more work to do in terms of processing requests. The experiment shows that the time required for this is evened out by the time saved for processing each request.

Similarly to CQueue and HQueue, the DQueue implementation uses a centralized component, namely the synchronizer. However, contrary to the HQueue case, in the DQueue, each island master sends one message instead of two, which contains the number of both the enqueues and dequeues requests. Since we do not see the throughput stop increasing at 128 cores, we conclude that now the dominant factor is not the time that the server requires to read the messages from its mailbox. The DQueue graph of Figure 9.1.a shows that DQueue scales well for up to 512 cores. Therefore, in the DQueue approach, the synchronizer does not pose a scalability problem. The reason for this is, not only that the synchronizer receives a smaller number of messages, but also that it has to do a simple arithmetic addition or subtraction for each batch of requests that it receives. This computational effort is significantly smaller than those carried out by the centralized component in the CQueue and HQueue implementations. Therefore, it is important that the local computation done by a server be small.

Notice that in the DQueue implementation, the actual request processing takes place on the hash table servers. So, clients do not initiate requests as frequently as in the previous algorithms, since they also have to communicate with the hash table servers. Moreover, the processing in this case is shared among the hash-table servers and therefore, this processing does not cause scalability problems. It follows that load balancing is also an important factor affecting scalability. In the case of the DQueue the local work on the synchronizer is a linear function of the amount of island masters. On the contrary, in the other two implementations, it is a function of the amount of clients. It follows that this is another reason for the good scalability observed on the DQueue implementation.

We remark that when the amount of clients, and therefore, of island
masters, is so large as to cause saturation on the synchronizer, a tree-like hierarchy of island masters would solve the scalability problem. There, the DQueue algorithm can offer a trade-off between overloaded activity of the centralized component and the latency that is caused by the height of the tree hierarchy.

Figure 9.1.a further shows the observed throughput of TQueue. The behavior of TQueue in terms of scalability follows that of HQueue. Notice that TQueue works in a similar way as HQueue, with the difference that the identity of the centralized component may change, as the token moves from server to server. This makes TQueue more complex than HQueue. As
a result, its throughput is lower than that of HQueue. However, TQueue is a nice generalization of HQueue, which can be used in cases where the expected size of the required queue is not known in advance and where a moderate number of cores are active.

In Figure 9.1.b, we fix the number of cores to 512 and perform the experiment for several different random work values (0 - 32K). It is shown that, for a wide range of values (0-512), we see no big difference on the performance of each algorithm. This is so because, for this range of values, the cost to perform the requests dominates the cost introduced by the random work. When the random work becomes too high (greater than 32K dummy loop iterations), the throughput of all algorithms degrades and the performance differences among them become minimal, since the amount of random work becomes then the dominant performance factor.

In Figure 9.2.a, we experimentally compare the performance of the centralized stack (CStack) with its hierarchical version where the island master performs elimination (EStack), an improved version of EStack where the island master performs batching (BStack), and the hierarchical versions of the directory-based stack (DStack) and the token based-stack (TStack). As expected, the centralized implementation does not scale for more than 16 cores. All other algorithms scale well for up to 512 cores. This shows that the elimination technique is highly-scalable. It is so efficient that it results in no significant performance differences between the algorithms that apply it.

In order to get a better estimation of the effect that the elimination technique has on the algorithms, we experimentally compared the performance that CStack, EStack, BStack, TStack, and DStack can achieve, when they are not performing elimination. Figure 9.2.b shows the obtained throughput. Notice that the scalability characteristics of CStack, BStack, and TStack are similar to those of CQueue, HQueue, and TQueue. This is not the case for DStack. The reason for this is the following. In our experiment, each client performs pairs of push and pops. By the way the synchronizer works, the push request and the pop request of each pair are often assigned the same key. This results in contention at the hash table. In an experiment where the number of pushes is not equal to the number of pops, the observed performance would be much better than that of other algorithms for all numbers of cores.

Figure 9.3.b shows the total number of messages sent in each experiment presented in Figure 9.1.a. Remarkably, there is an inverse relationship between these results and those of Figure 9.1.a. For instance, DQueue circulates the largest number of messages in the system. However, DQueue is the implementation that scales best. Thus, the total number of messages is not necessarily an indicative factor of the actual performance. This is so because a server may easily become saturated even by a moderate amount of messages, if the required processing is important. This is for example the case in the CQueue implementation. Therefore, in order to be scalable, an
implementation should avoid overloading the servers in this manner. Backed up by Figure 9.3.b, becomes apparent that good scalability is offered when the implementation ensures good load balancing between the messages that the servers have to process.

We distill the empirical observations of this section into a metric. As observed, achieving load balancing in terms of evenly distributing both messages and the processing of requests to servers is important for ensuring scalability. If we fix an algorithm $\mathcal{A}$, this is reflected in the number of messages received by each server $s$ for performing $m$ requests when executing $\mathcal{A}$. By denoting this number as $msg_s$, we can define the scalability factor $sf_s$ of a server $s$ as follows:

$$ sf_s = \lim_{m \to \infty} \frac{msg_s}{m} $$

(9.1)

where we assume that a maximum number of clients repeatedly initiate requests and these clients are scheduled fairly. We define the scalability factor $sf$ of $\mathcal{A}$ as the maximum of the scalability factors of all servers.

$$ sf = \max_s \{ sf_s \} $$

(9.2)
We remark that a low scalability factor indicates good scalability behavior. The intuition behind this metric is that the lower the $sf_s$ fraction is, the more requests are batched in a message that reaches a server. Computational overhead for the reading, handling and decoding of a single message is then spread over multiple requests. As a scalability indicator, the scalability factor shows that it is of more interest to attempt not to minimize the total number of messages that are trafficked in a system, but to design implementations that minimize the total number of messages that a server has to process. Figure 9.4 shows graphs of the scalability factors of the the proposed implementations. The results agree with this theoretical perception.
Chapter 10

Implementation of Shared-Memory Primitives

In Java, synchronization primitives are provided as methods of the library `sun.misc.unsafe`. In order to support most of the methods `sun.misc.unsafe` uses, we need to implement the atomic updates, the compare-and-swap primitives, and the pair of `park()`/`unpark()` methods used for thread synchronization. Once these primitives have been supported, the implementations of several data structures that are provided in `java.util.concurrent` will come for free.

In addition to these primitives, we have implemented the primitives fetch-and-increment and swap, to implement numeric operations used by the methods of package `java.util.concurrent.atomic`, described in Section A.1, e.g. `getAndSet()`. These primitives need to be implemented on a lower level than the Java Virtual Machine, since `sun.misc.unsafe` mostly calls primitives that need either to access the system or hardware resource.

We have to point out that Formic does support reads and stores to remote memory locations but does not provide coherence. Thus, this remote store is different from the atomic write primitive we want to implement, due to the atomicity that the second provides. By adding this primitive, we aim to give safety when simple writes occur concurrently with other atomic primitives, such as CAS, etc.

Hence, we implemented the following atomic primitives:

- Read, which takes as arguments a memory address and returns the value that is stored in this address.

- Write, which takes as arguments a memory address and a value, and stores the new value to this address.

- Fetch-And-Increment, which takes as arguments a memory address and a value, and adds the new value to the existing one stored in the address. It then returns the new value.
• Swap, which takes as arguments a memory address, and a new value, and replaces the stored value with the new one. It then returns the previous value.

• Compare-And-Swap, which takes as arguments a memory address, an old value and a new value. If the value stored in the memory address equals to the old value, then it is replaced by the new value. Then, the function returns the old value.

10.1 Atomic Accesses Support

The java.util.concurrent.atomic package provides a set of atomic types (as classes). Instances of these classes must be atomically accessed. In shared-memory architectures these classes are implemented by delegating the complexity of synchronization handling to instructions provided by the underlying architecture, e.g., memory barriers, compare and swap etc. In the case of Formic, however, such instructions are not available. As a result the atomic types need to be implemented in software.

A naive implementation is to delegate the handling of such operations to a manager similar to the monitor manager, presented in Section 2.3 of Deliverable 1.1. This manager would be responsible for holding the values associated to instances of atomic types, as well as, for performing atomic operations on them. Such a manager is capable of reducing the memory traffic regarding synchronization in some cases.

Algorithm 43 Example of getAndSet implementation.

```java
public final int getAndSet(int newValue) {
    for (;;) {
        int current = get(); /* Synchronization point */
        if (compareAndSet(current, newValue)) /* Synchronization point */
            return current;
    }
}
```

For instance, in the implementation depicted in Algorithm 43, the JVM constantly tries to set the value of the object to `newValue` before its value changes between the `get()` and the `compareAndSet` invocations. This may result in many failures and unnecessary synchronizations in case of high contention. When using a manager there is no need for such a loop. Since the accesses are only possible by the manager, the manager can assume that the value will never change between the `get()` and a consequent `set()` invocation. As a result we could implement the logic of the above function in the manager and avoid the extra communication from the loop iterations. The equivalent algorithm in the manager is shown in Algorithm 44.
Algorithm 44 Implementation of getAndSet through the monitor manager.

8 public final int getAndSet(int newValue) {
9     int current = get();
10     set(newValue);
11     return current;
12 }

A similar effect can be achieved by using the synchronized classifier. Since the code inside a synchronized block or method can be seen as atomic we can implement getAndSet as above but without the need of a centralized manager by just adding synchronized to the method classifiers, as shown in Algorithm 45.

Algorithm 45 Implementation of getAndSet using synchronized.

13 public final synchronized int getAndSet(int newValue) {
14     int current = get();
15     set(newValue);
16     return current;
17 }

Note that, in shared-memory processors, this approach is probably less efficient than the first one when there is no contention on the object. In the lack of contention the first implementation would just perform a read instruction and a compare and swap, while in the third implementation it would also need to go through the monitor implementation of the virtual machine (at minimum an extra compare and swap, and a write). On the Formic, however, where there is no compare and swap instruction, the first implementation would require at least two request messages (get and compareAndSet) to the manager and the corresponding two replies per iteration. The third implementation would also need two requests (a monitor acquire and a monitor release) and the corresponding two replies but with the difference that it never needs to repeat this process, resulting in reduced memory traffic.

To further improve this approach we avoid the use of regular monitors and replace them with readers-writers monitors. This way in the case of contented reads we do not restrict access to a single thread, allowing for increased parallelism. As a further optimization we slightly change the semantics of compareAndSet and make it lazily return False in case that the object is owned by another writer. This behavior is based on the heuristic that if an object is owned by a writer it is going to be written and the value will probably be different than the one the programmer provided as expected to the compareAndSet.
10.1.1 Readers-Writers Implementation

The readers-writers monitors are implemented using the same monitor manager, presented in Section 2.3 of Deliverable 1.1, but with different operation codes. We introduce four new operations code, `READ_LOCK`, `WRITE_LOCK`, `READ_UNLOCK`, and `WRITE_UNLOCK`. For each readers-writers monitor we keep two different queues, the readers queue and the writers queue. The readers queue holds the thread IDs of the threads waiting to acquire a read-lock on the monitor. Correspondingly, the writers queue holds the thread IDs of the threads waiting to acquire a write-lock on the monitor.

When a monitor is read-locked we hold the count of threads sharing that read monitor, while for write-locked monitors we hold the thread ID of the thread owning that write monitor. As long as the writers lock is empty the monitor manager services read-lock requests by increasing the counter and sending an acknowledgement message to the requester. When a write-lock request arrives, it gets queued to the writers queue and read-lock requests start being queued in the readers-queue instead of being served, essentially giving priority to the writer-lock requests. Eventually, when all the readers release the read-locks they hold the write-lock requests will start being served, and read-lock requests will be served again only after the writers-queue gets empty again.

We chose to give priority to the write locks since in most algorithms writing a variable is less common than reading it (e.g., polling). In order to implement a more fair mechanism we could set a threshold on the number of read-lock and write-lock requests being served at each phase.

Finally, to support the lazy fail for `compareAndSet` we introduce another operation code, the `TRY_LOCK`. This is a special write-lock request that if the monitor is not free, it does not get queued in the write-queue, but a negative acknowledgement is send back to the requester, notifying him that the monitor is not free.
Chapter 11

Related Work

Previous research results [38, 39, 40, 41] propose how to support dynamic data structures on distributed memory machines. Some are restricted on tree-like data structures, other focus on data-parallel programs, some favor code migration, whereas other focus on data replication. We optimize beyond simple distributed memory architectures by exploiting the communication characteristics of non cache-coherent multicore architectures. Some techniques from [38, 39, 40] could be of interest though to further enhance performance and scalability in our implementations.

Transactional memory (TM) [42, 43] is a programming paradigm which provides the transaction’s abstraction; a transaction executes a piece of code containing accesses to data items. TM ensures that each transaction seems to execute sequentially and in isolation. Distributed transactional memory (DTM) [44, 45, 46, 47, 48, 49, 50, 51] is a generic approach for achieving synchronization, so data structures can be implemented on top of them. However, to do so, DTM systems introduce not only significant space overheads by maintaining metadata for every object and every transaction, but also performance overheads whenever reads from or writes to data items take place. Moreover, DTM requires the programmer to write its code in a transactional-compatible way. (When the transactions dynamically allocate data, as when they synchronize operations on dynamic data structures, compilers cannot detect all possible data races without trading performance, by introducing many false positives.) Our work is on a different avenue: towards providing a customized library of highly-scalable data structures, specifically tailored for non cache-coherent machines.

TM$^2$C [47] is a DTM proposed for non cache-coherent machines. The paper presents a simple distributed readers/writers lock service where nodes are responsible for controlling access to memory regions. It also proposes two contention management (CM) schemes (Wholly and FairCM) that could be used to achieve starvation-freedom. However, in Wholly, the number of times a transaction $T$ may abort could be as large as the number of transactions the process executing $T$ has committed in past, whereas in
FairCM, progress is ensured under the assumption that there is no drift \([30, 52]\) between the clocks of the different processors of the non cache-coherent machine. Read-only transactions in TM²C can be slow since they have to synchronize with the lock service each time they read a data item, and in case of conflict, they must additionally synchronize with the appropriate CM module and may have to restart several times from scratch. Other existing DTM\s [44, 45, 53, 51, 54], not only impose common DTM overheads, but also may cause livelocks thus not providing strong progress guarantees.

The data structure implementations we propose do not cause any space overhead, read-only requests are fast, since all nodes that store data of the implemented structure search for the requested key in parallel, and the number of steps executed to perform each request is bounded. We remark that, in our algorithms, information about active requests is submitted to the nodes where the data reside, and data are not statically assigned to nodes, so our algorithms follow neither the data-flow approach [55, 54] nor the control-flow approach [44, 53] from DTM research.

Distributed directory protocols [56, 57, 58, 59, 60] have been suggested for locating and moving objects on a distributed system. Most of the directory protocols follow the simple idea that each object is initially stored in one of the nodes, and as the object moves around, nodes store pointers to its new location. They are usually based either on a spanning tree [61, 60] or a hierarchical overlay structure [56, 58, 59]. Remarkably, among them, COMBINE [56] attempts to cope with systems in which communication is not uniform. Directory protocols could potentially serve for managing objects in DTM. However, to implement a DTM system using a directory protocol, a contention manager must be integrated with the distributed directory implementation. As pointed out in [57], this is not the case with the current contention managers and distributed directory protocols. It is unclear how to use these protocols to get efficient versions of the distributed data structures we present in this paper.

Distributed data structures have also been proposed [62, 63, 64, 65, 66] in the context of peer-to-peer systems or cluster computing, where dynamicity and fault-tolerance are main issues. They tend to provide weak consistency guarantees. Our work is on a different avenue.

Hierarchical lock implementations and other synchronization protocols for NUMA cache-coherent machines are provided in [16, 17, 18, 19, 20, 21, 22]. We extend some of the ideas from these papers, and combine them with new techniques to get hierarchical implementations for a non cache-coherent architecture. Tudor et al. [67] attempt to identify patterns on search data structures, which favor implementations that are portably scalable in cache-coherent machines. The patterns they came up with cannot be used to automatically generate a concurrent implementation from its sequential counterpart; they rather provide hints on how to apply optimizations when designing such implementations.

Hazelcast [26] is an in-memory data grid middleware which offers im-
implementations for maps, queues, sets and lists from the Java concurrency utilities interface. These implementations are optimized for fault tolerance, so some form of replication is supported. Lists and sets are stored on a single node, so they do not scale beyond the capacity of this node. The queue stores all elements to the memory sequentially before flushing them to the datastore. Like Hazelcast, GridGain [68], an in-memory data fabric which connects applications with datastores, provides a distributed implementation of queue from the Java concurrency utilities interface. The queue can be either stored on a single grid node, or be distributed on different grid nodes using the datastore that exists below GridGain.
Chapter 12

Conclusions

We have implemented the work outlined in Workpackage 2 of GreenVM. This workpackage is concerned with providing library support for power-efficient data structures for the encore and similar architectures. With this aim in mind, we have presented a comprehensive collection of data structures for future many-core architectures. The collection could be utilized by runtimes of high-productivity languages ported to such architectures. Specifically, our collection implements Task 2.1 of Workpackage 2, by providing all types of concurrent data structures supported in Java’s concurrency utilities. Other high-level productivity languages that provide shared memory for thread communication could also benefit from our library. Specifically, we provide several different kinds of queues, including static, dynamic and synchronous; our queue (or deque) implementations can be trivially adjusted to provide the functionality of delay queues (or delay deques) \[10\]. We do not provide a priority queue implementation, since it is easy to adopt a simplified version of the priority queue presented in \[69\] in our setting. Our list implementations provide the functionality of sets, whereas the simple hash table that we utilize to design some of our data structures can serve as a hash-based map.

We have implemented Task 2.2 of the package by providing hierarchical versions of the data structures that we have implemented. These implementations take into consideration challenges that are raised in realistic scalable multicore architectures where communication is implicit only between the cores of an island whereas explicit communication is employed among islands.

We have performed experimental evaluation of the implemented data structures in order to examine both their throughput and energy efficiency. Detailed analysis of the obtained results is presented in chapter 5 of Deliverable 3.2. of the GreenVM project. Our experiments show the performance and scalability characteristics of some of the techniques on top of a non cache-coherent hardware prototype. They also illustrate the scalability power of the hierarchical approach in such machines. We believe that the proposed implementations will exhibit the same performance characteris-
tics, if programmed appropriately, in prototypes with similar characteristics as FORMIC, like Tilera or SCC. We expect this also to be true for future, commercially available, such machines.
Appendix A

Java Concurrency Utilities package

The Java concurrency library contains all the concurrent data structures of the Java language. The library is called java.util.concurrent and we focus on the following three packages it contains: the main package that includes the data structures and some useful frameworks; a package named locks with implementations of locking mechanisms for the Java threads; and a package atomic that provides support for thread-safe operations without the use of locks on single variables, class fields or array elements. We provide a short description of the contents of each of these packages. The full description can be found in [11].

We start by describing packages atomic and locks in Appendix A.1 and A.2, given that java.util.concurrent relies on primitives implemented in those packages. Appendix A.3 describes the data structures included in the library itself.

A.1 java.util.concurrent.atomic

The classes of this package provide atomic operations to a single variable, a class field, or an array element. The functionality that they offer can be used to implement data structures that do not use locks in order to keep their data consistent. In more detail:

- AtomicBoolean, AtomicInteger, AtomicLong, and AtomicReference are classes that provide atomic reads and writes to a variable of the corresponding type,

- AtomicIntegerArray, AtomicLongArray and AtomicReferenceArray are classes that further extend atomic operation support for arrays of the corresponding type, and offer volatile access semantics to each of their elements separately,
• **AtomicReferenceFieldUpdater**, **AtomicIntegerFieldUpdater**, and **AtomicLongFieldUpdater** are classes that allow atomic operations to specific volatile fields of a class. These classes are used mainly by data structures that require atomic updates to different fields of the same structure node, independently from one another or the other fields of the same node.

• **AtomicMarkableReference** a class that associates a boolean variable, a mark, with a class reference, in order to be atomically updated as a pair,

• **AtomicStampedReference** a class that pairs an integer variable, a stamp, with a reference, so that this stamp and the reference to be accessed and updated atomically.

All the aforementioned classes include the following methods:

• **get** and **set** that provide the same memory effects with a read/write to a volatile variable,

• **lazySet** that sets a variable to a given value, which will eventually be visible to the calling thread. This method has the same memory effects with a write to a volatile variable but also allows the re-ordering of the memory actions that follow **lazySet**, provided that these memory actions respect the re-orderings constraints of other non-volatile writes,

• **getAndSet** that atomically sets a variable to a given value and then returns the previous value,

• **weakCompareAndSet** that offers atomic reads and conditional writes to a target variable, but does not guarantee a strict order of previous or subsequent reads or writes of other variables,

• **compareAndSet** that atomically sets the value of a variable with the given value if and only if its value is equal to an expected value.

Apart from these methods, some classes include some other read-and-update methods that are implemented with the **compareAndSwap** primitive. Compare-and-swap (CAS) compares the value of a memory address, with a given value, and only if they are equal, updates this address with a new given value. These methods that are implemented using this primitive are included in classes that access and update arithmetic variables, like the class **AtomicInteger**, and are needed to offer more functionality to the programmer. These methods are **addAndGet**, **getAndAdd**, **incrementAndGet**, **getAndIncrement**, **decrementAndGet** and **getAndDecrement**.
A.2 java.util.concurrent.locks

The java.util.concurrent.locks package provides a framework that offers locking semantics that are distinct from the built-in synchronization (Java monitors) provided by the Java language. It offers support for different lock implementations that have a more complex syntax than the built-in monitor enter and monitor exit that provide simple mutual exclusion.

This package consists of three interfaces, classes that implement those interfaces, and helping classes that offer further functionality. The interfaces are the following:

- **Lock**, that offers additional locking semantics and enhances Java’s built-in synchronization (synchronous monitors), such as a reentrant lock, adds fairness, etc.,

- **Condition**, which is combined together with a lock and offers to an object the methods `wait` and `notify`. This interface offers additional functionality to the methods `wait` and `notify` of the `Object` class, such as the capability of associating multiple `Condition` objects to a single `Lock`,

- **ReadWriteLock** that offers a pair of `Lock` objects, one for read and one for write.

The interface implementations consist of the class `ReentrantLock` and the class `ReentrantReadWriteLock` that implement the interfaces `Lock` and `ReadWriteLock` respectively. These classes that implement the `Lock` interface use a synchronizer, a class named `AbstractQueuedSynchronizer` also located in the same package, which is implemented as an MCS locking queue in order to maintain the Java thread synchronization. The other helping class is the `LockSupport` that offers methods for blocking and unblocking threads that wait for a lock.

A.3 java.util.concurrent

The package java.util.concurrent contains a collection of various abstract data types (ADT) that provide thread safety. These ADTs are Queues, a collection of elements that are stored in a first-in-first-out (FIFO) order; Maps, a collection of key-value pairs; Lists, an ordered and indexed collection of elements that allows duplicate objects and Sets, an unordered collection of elements that does not permit duplicate objects. This package contains the following data structure implementations:

- **Queues**:
  - `ArrayBlockingQueue<E>`, a blocking queue with static size, that is implemented as an array and uses a `ReentrantLock` for thread synchronization,
ConcurrentLinkedQueue<E>, a non-blocking queue implementation from the paper "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms" by Michael and Scott [12] that uses the function compareAndSet for atomic updates of the queue’s head and tail pointers.

LinkedBlockingQueue<E>, the blocking queue implementation from the previously mentioned paper [12], that uses two instances of the ReentrantLock class for the thread synchronization, one for enqueues and one for dequeues,

LinkedBlockingDeque<E> a doubly-linked, blocking list that uses a ReentrantLock for the thread synchronization,

PriorityBlockingQueue<E> a PriorityQueue implementation that uses a ReentrantLock for thread synchronization,

DelayQueue<E extends Delayed> a collection of elements, which are associated with a delay value that represents the time the element must be stored in the queue before it can be removed from it. A PriorityQueue is used for the storage of these elements, with the priority playing the role of the delay. The PriorityQueue uses a ReentrantLock for thread synchronization, and

SynchronousQueue<E> an implementation based on the paper ”Nonblocking Concurrent Data Structures with Condition Synchronization” by Scherer and Scott [70].

Maps:

ConcurrentHashMap<K,V> a thread safe Hash Map using atomic operations for element updates, and

ConcurrentSkipListMap<K,V>, that is based on the Skip Lists by Pugh [13]. It uses the class Updaters from the package java.util.concurrent.atomic,

Lists:

CopyOnWriteArrayList<E>, a thread safe implementation of the class ArrayList, that produces a copy of the current Array after every mutative action (e.g. set, modify). It results in creating different snapshots of the Array, and thus any iterator that goes through the elements of the Array may use one of these snapshots without being interfered by any subsequent mutative action. The thread safety is provided by a ReentrantLock.

Sets:

CopyOnWriteArraySet<E>, that internally uses the previously mentioned class CopyOnWriteArrayList<E> for element store.
– ConcurrentSkipListSet\(E\), that uses internally the previously mentioned class ConcurrentSkipListMap\(K,V\) for element store.

This package also includes the classes CountDownLatch, CyclicBarrier and Semaphore as additional implementations to further enhance the options for thread synchronization. These classes use AbstractQueuedSynchronizer also, as the basic framework for thread safety.

The CountDownLatch is a mechanism to block a set of threads until a specific set of actions has been executed by other threads. The latch is initialized with the number of times the set of actions is to be executed. The set of threads blocks while waiting for the set of operations to be completed as many times as was initialized. Each time the set of actions is finished, the latch is decreased. When the latch equals to zero, the set of threads unblocks.

The CyclicBarrier is a mechanism that allows a set of threads to wait one another until all of them have perform a set of actions. All of the thread must finish before they continue. After the set of actions has been performed by all threads, the barrier can be triggered again and reset to its original value and reused.

A Semaphore, is a synchronization mechanism that allows a number of threads to execute a set of actions. The semaphore is initialized with a number of permits, which may differ to the number of threads executing this set of actions. When a thread begins to execute the set of actions, the number of permits is decreased, and when the thread competes the set, the number of permits is increased. If the permits reaches zero, no more threads are allowed to execute the set of actions, and must wait until one of the executing thread releases its permit.

All the aforementioned classes do excessive use both of the atomic primitives and the thread synchronization techniques offered by the packages lock and atomic. Those primitives though, are implemented by calling native methods provided by the sun.misc.unsafe class. These methods provide low-level, unsafe operations to Java object fields, located in the Java Heap. They are implemented in platform-dependent code, probably in C or assembly, that grant access to system or hardware resources.

Specifically, those methods are:

- Methods that obtain the value stored in the address of a Java object \(o\) plus an offset
  
  – public native boolean getBoolean(Object o, long offset)
  – public native byte getByte(Object o, long offset)
  – public native int getInt(Object o, long offset)
  – public native long getLong(Object o, long offset)

- Likewise, methods that store a value \(x\) to an object \(o\).
- public native boolean putBoolean(Object o, long offset, boolean x)
- public native byte putByte(Object o, long offset, byte x)
- public native int putInt(Object o, long offset, int x)
- public native long putLong(Object o, long offset, long x)

- The volatile versions of the get methods
  - public native boolean getBooleanVolatile(Object o, long offset)
  - public native byte getByteVolatile(Object o, long offset)
  - public native int getIntVolatile(Object o, long offset)
  - public native long getLongVolatile(Object o, long offset)
  - public native Object getObjectVolatile(Object o, long offset)

- Likewise, methods that store a value x to a volatile field of an object o
  - public native void putBooleanVolatile(Object o, long offset, boolean x)
  - public native void putByteVolatile(Object o, long offset, byte x)
  - public native void putIntVolatile(Object o, long offset, int x)
  - public native void putLongVolatile(Object o, long offset, long x)
  - public native void putObjectVolatile(Object o, long offset, Object x)

- Methods similar to the previous set, with the difference that they do not guarantee that the newly stored value will become instantly visible to the other threads running simultaneously
  - public native void putOrderedObject(Object o, long offset, Object x)
  - public native void putOrderedInt(Object o, long offset, int x)
  - public native void putOrderedLong(Object o, long offset, long x)
• Compare-And-Swap operations. They compare the contents of the object \( o \) to the value \( \text{expected} \), and only if they are the same, they change them to the given new value \( x \). It returns \text{true} upon success or \text{false} upon failure

- public final native boolean compareAndSwapObject(Object \( o \), long offset, Object expected, Object x)
- public final native boolean compareAndSwapInt(Object \( o \), long offset, int expected, int x)
- public final native boolean compareAndSwapLong(Object \( o \), long offset, long expected, long x)

• And finally, methods that block and unblock Java threads

- public native void unpark(Object thread). This method is unsafe because the caller must make sure that the thread was not destroyed before or during the unpark.
- public native void park(boolean isAbsolute, long time). This call is inevitable in this library due to unpark being there.

The library \texttt{sun.misc.unsafe} contains also many other native methods, but the class \texttt{java.util.concurrent} does not use them. Hence, presenting them here, is out of the scope of this work.
Bibliography


[49] Kaloian Manassiev, Madalin Mihaiescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. 135


