SKUEUE: A Scalable and Sequentially Consistent Distributed Queue*

Michael Feldmann, Christian Scheideler and Alexander Setzer

Department of Computer Science

Paderborn University, Germany

{michael.feldmann, scheideler, alexander.setzer}@upb.de

Abstract—**We propose a distributed protocol for a queue, called SKUEUE, which spreads its data fairly onto multiple processes, avoiding bottlenecks in high throughput scenarios. SKUEUE can be used in highly dynamic environments, through the addition of JOIN() and LEAVE() requests to the standard queue operations ENQUEUE() and DEQUEUE(). Furthermore SKUEUE satisfies sequential consistency in the asynchronous message passing model. Scalability is achieved by aggregating multiple requests to a batch, which can then be processed in a distributed fashion without hurting the queue semantics. Operations in SKUEUE need a logarithmic number of rounds w.h.p. until they are processed, even under a high rate of incoming requests.**

Index Terms—**Distributed Systems; Distributed Data Structures; Distributed Queue; Queue Semantics**

I. INTRODUCTION

Like in the sequential world, efficient distributed data structures are important in order to realize efficient distributed applications. The most prominent type of distributed data structure is the distributed hash table (DHT). Many distributed data stores employ some form of DHT for lookup. Important applications include file sharing (e.g., BitTorrent), distributed file systems (e.g., PAST), publish subscribe systems (e.g., SCRIBE), and distributed databases (e.g., Apache Cassandra). Other distributed forms of well-known data structures, however, like queues, stacks, and heaps has been given much less attention though queues, for example, have a number of interesting applications as well. A distributed queue can be used to come up with a unique ordering of messages, transactions, or jobs, and it can be used to realize fair work stealing [1] since tasks available in the system would be fetched in FIFO order. Other applications are distributed mutual exclusion, distributed counting, or distributed implementations of synchronization primitives. Server-based approaches of realizing a queue in a distributed system already exist, like Apache ActiveMQ, IBM MQ, or JMS queues. Many other implementations of message and job queues can be found at http://queues.io/. However, none of these implementations provides a queue that allows massively parallel accesses without requiring powerful servers. The major problem of coming up with a fully distributed version of a queue is that its semantics are inherently

sequential. Nevertheless, we are able to come up with a distributed protocol for a queue ensuring sequential consistency that fairly distributes the communication and storage load among all members of the distributed system and that can efficiently process even massive amounts of ENQUEUE() and DEQUEUE() requests. Our protocol works in the asynchronous message passing model and can also handle massive amounts of join and leave requests efficiently. We are not aware of any distributed queue with a comparable performance.

A. Basic notation

A *Distributed Queue* provides four operations: ENQUEUE(), DEQUEUE(), JOIN() and LEAVE(). ENQUEUE() adds an element to the queue and DEQUEUE() removes an element from the queue so that the FIFO requirement is satisfied. JOIN() allows a process to enter the system while LEAVE() allows a process to leave the system. Let $\mathcal{E}$ be the universe of all elements that may possibly be put into the distributed queue.

While in a standard, sequential queue it is very easy to guarantee the FIFO property, it is much harder to guarantee in a distributed system, especially when messages have arbitrary finite delays and the processes do not have access to a local or global clock, as is usually assumed in the asynchronous message passing model. In essence, a global serialization of the requests has to be established without creating bottlenecks in the system. We will show that it is possible to obtain a serialization ensuring sequential consistency even under a high request rate. In order to define sequential consistency, we first need some notation.

Let $\text{ENQ}_{v,i}$ refer to the i-th ENQUEUE() request that was called in process v. Analogously, $\text{DEQ}_{v,i}$ refers to the i-th DEQUEUE() request that was called in process v. Furthermore, $\text{OP}_{v,i}$ denotes the i-th (ENQUEUE() or DEQUEUE()) request that was called in process v. We assume w.l.o.g. that every $e \in \mathcal{E}$ is enqueued at most once into the system (an easy way to achieve this is to make the calling process and the current count of requests performed a part of e). Let S be the set of all ENQUEUE() and DEQUEUE() requests issued by the processes in the system. We say that $\text{ENQ}_{v,i}$ is *matched* with $\text{DEQ}_{w,j}$ if the $\text{DEQ}_{w,j}$ request returns the element contained in the $\text{ENQ}_{v,i}$ request. Let M be the set of all matchings. Note that there may be requests that are not matched and thus not contained in M.

Definition 1. *A Distributed Queue protocol with operations* ENQUEUE() *and* DEQUEUE() *is* sequentially consistent *if and only if there is an ordering $\prec$ on the set S of all* ENQUEUE() *and* DEQUEUE() *requests issued to the system so that the set of all enqueue-dequeue matchings M established by the protocol satisfies:*

1) *for all* $(\mathrm{ENQ}_{v,i}, \mathrm{DEQ}_{w,j}) \in M$: $\mathrm{ENQ}_{v,i} \prec \mathrm{DEQ}_{w,j}$,
2) *for all* $(\mathrm{ENQ}_{v,i}, \mathrm{DEQ}_{w,j}) \in M$: *during the execution, there is no $\mathrm{DEQ}_{u,k}$ not contained in M such that $\mathrm{ENQ}_{v,i} \prec \mathrm{DEQ}_{u,k} \prec \mathrm{DEQ}_{w,j}$, and there is no $\mathrm{ENQ}_{u,k}$ not contained in M such that $\mathrm{ENQ}_{u,k} \prec \mathrm{ENQ}_{v,i} \prec \mathrm{DEQ}_{w,j}$,*
3) *for all distinct $(\mathrm{ENQ}_{u,i}, \mathrm{DEQ}_{v,j}), (\mathrm{ENQ}_{w,k}, \mathrm{DEQ}_{x,l}) \in M$ it does not hold: $\mathrm{ENQ}_{u,i} \prec \mathrm{ENQ}_{w,k} \prec \mathrm{DEQ}_{x,l} \prec \mathrm{DEQ}_{v,j}$ or $\mathrm{ENQ}_{w,k} \prec \mathrm{ENQ}_{u,i} \prec \mathrm{DEQ}_{v,j} \prec \mathrm{DEQ}_{x,l}$, and*
4) *for all $v \in V$ and $i \in \mathbb{N}$: $\mathrm{OP}_{v,i} \prec \mathrm{OP}_{v,i+1}$*

Intuitively, the four properties have the following meaning: The first property means that an element has to be enqueued before it can be dequeued. The second property means that each DEQUEUE() request returns a value if there is one in the queue and that each element passed as a parameter of an ENQUEUE() request will be added to the queue. The third property means elements are dequeued in the order they have been added to the queue. Finally, the fourth property is the *local consistency* property: It means that for each single process, the requests performed by this process have to come up in $\prec$ in the order they were executed by that process.

Note that if there is only a single process in the system, then the ENQUEUE() and DEQUEUE() operations of the Distributed Queue have exactly the same semantics as a classical queue.

B. Model

The distributed queue consists of multiple processes that are interconnected by some overlay network. We model the overlay network as a directed graph $G = (V, E)$, where V represents the set of processes and an edge (v, w) indicates that v knows w and can therefore send messages to w. Each process v can be identified by a unique identifier $v.id \in \mathbb{N}$.

We consider the asynchronous message passing model where every process v has a set $v.Ch$ for all incoming messages called its *channel*. That is, if a process u sends a message m to process v, then m is put into $v.Ch$. A channel can hold an arbitrary finite number of messages and messages never get duplicated or lost.

Processes may execute *actions*: An action is just a standard procedure that consists of a name, a (possibly empty) set of parameters, and a sequence of statements that are executed when calling that action. It may be called locally or remotely, i.e., every message that is sent to a process contains the name and the parameters of the action to be called. We will only consider messages that are remote action calls. An action in a process v is *enabled* if there is a request for calling it in $v.Ch$. Once the request is processed, it is removed from $v.Ch$. We assume fair message receipt, i.e., every request in a channel is eventually processed. Additionally, there is an action that

is not triggered by messages but is executed periodically by each process. We call this action TIMEOUT.

We define the *system state* to be an assignment of a value to all protocol-specific variables in the processes and a set of messages to each channel. A *computation* is a potentially infinite sequence of system states, where the state s_{i+1} can be reached from its previous state s_i by executing an action that is enabled in s_i.

We place no bounds on the message propagation delay or the relative process execution speed, i.e., we allow fully asynchronous computations and non-FIFO message delivery.

For the runtime analysis, we assume the standard synchronous message passing model, where time proceeds in *rounds* and all messages that are sent out in round i will be processed in round $i + 1$. Additionally, we assume that each process executes its TIMEOUT action once in each round.

C. Related Work

The most important type of distributed data structure is the distributed hash table, for which seminal work has been done by Plaxton et al. [2] and Karger et al. [3]. Distributed hash tables have a wide range of practical realizations, such as Chord [4], Pastry [5], Tapestry [6] or Cassandra [7]. Our queue protocol makes use of a distributed hash table through consistent hashing.

Distributed hash tables do not support range queries, so distributed trees were proposed, e.g. in [8], [9], to overcome this.

There is a wealth of literature on *concurrent* data structures. Consider, for example, [10] for a queue, [11] for a stack, [12] for a priority queue or [13] for a general survey. These structures allow multiple processes to send requests to a data structure that is stored in shared memory. Hendler et al. [14] present a scalable synchronous concurrent queue, where they used a parallel flat-combining algorithm similar to the aggregation technique used in this work: A single 'combiner' thread gets to know requests of other threads and then executes these requests on the queue. However, they do not provide any guarantees on the semantics, as their queue is considered to be *unfair*, meaning that it does not impose an order on the servicing of requests. Shavit and Taubenfeld formulated some (relaxed) semantics for concurrent queues and stacks in [15]. The main difference of concurrent data structures compared to distributed data structures is that there has to be a single instance that stores the data, whereas distributed data structures are fully decentralized.

A scalable distributed heap called *SHELL* has been presented by Scheideler and Schmid in [16]. *SHELL*'s topology resembles the De Bruijn graph and is shown to be very resilient against Sybil attacks. Our protocol uses the virtual De Bruijn graph from Richa et al. [17], which is based on [18], where Naor and Wieder showed how to construct P2P systems in the continuous space.

Plenty of work has also been done on *distributed queuing*, but this is very different from our approach. Distributed queuing is all about the participants of the system forming

a queue: Every process introduces itself to its predecessor and (depending on its position) knows its successor in the queue. Distributed queuing is not about inserting elements into a distributed data structure that is maintained by multiple processes, which can generate requests to the data structure. See, for example, the Arrow protocol in [19], which was made self-stabilizing in [20], or a protocol for dynamic networks in [21].

D. Our Contribution

We propose a protocol for a distributed queue which guarantees sequential consistency (Definition 1). Requests can be handled very effectively due to the aggregation of multiple requests to a batch. This fact makes our queue highly scalable for both, a large number of processes and a high load of queue requests. More precisely, when assuming synchronous message passing, our ENQUEUE() and DEQUEUE() operations are processed in $\mathcal{O}(\log n)$ rounds w.h.p. Furthermore we show that we can process n JOIN() or $n/2$ LEAVE() operations in $\mathcal{O}(\log n)$ rounds. Through the usage of a distributed hash table, our distributed queue allocates its elements equally among all processes, such that no process stores significantly more elements than the rest.

The paper is structured as follows: In Section II we describe the linearized De Bruijn network topology, into which we embed a distributed hash table. The general ideas for our protocol are presented in Section III along with descriptions for ENQUEUE() and DEQUEUE() operations. In Section IV we extend the protocol in order to support JOIN() and LEAVE() operations. We explain how to modify SKUEUE in order to work as a distributed stack and present experimental results for both the queue and the stack (Sections VI and **??**). Before we conclude the paper in Section VIII, we analyze the most important properties of our protocol in Section V.

II. PRELIMINARIES

A. Linearized De Bruijn Network

We adapt a dynamic version of the De Bruijn graph from [17], which is based on [18], for our network topology:

Definition 2. *The* Linearized De Bruijn network *(LDB) is a directed graph $G = (V, E)$, where each process v emulates 3 (virtual) nodes: A left virtual node $l(v) \in V$, a middle virtual node $m(v) \in V$ and a right virtual node $r(v) \in V$. The middle virtual node $m(v)$ has a real-valued* label[1] *in the interval $[0, 1)$. The label of $l(v)$ is defined as $m(v)/2$ and the label of $r(v)$ is defined as $(m(v) + 1)/2$. The collection of all virtual nodes $v \in V$ is arranged in a sorted cycle ordered by node labels, and $(v, w) \in E$ if and only if v and w are consecutive in this ordering (*linear edges*) or v and w are emulated by the same process (*virtual edges*).*

We will assume that the label of a middle node $m(v)$ is determined by applying a publicly known pseudorandom hash

---

[1]We may indistinctively use v to denote a node or its label, when clear from the context.

function on the identifier $v.id$. We say that a node v is *right* (resp. *left*) of a node w if the label of v is greater (resp. smaller) than the label of w, i.e., $v > w$ (resp. $v < w$). If v and w are consecutive in the linear ordering and $v < w$ (resp. $v > w$), we say that w is v's *successor* (resp. *predecessor*) and denote it by $succ(v)$ (resp. $pred(v)$). As a special case we define $pred(v_{min}) = v_{max}$ and $succ(v_{max}) = v_{min}$, where v_{min} is the node with minimal label value and v_{max} is the node with maximal label value. This guarantees that each node has a well defined predecessor and successor on the sorted cycle. More precisely, each node v maintains two variables $pred(v)$ and $succ(v)$ for storing its predecessor and successor nodes. Whenever a node v gets to know the reference of another node w, such that w is stored in either $pred(v)$ or $succ(v)$, we assume that v also gets to know whether w is a left, middle or right virtual node. This can be done easily by attaching the information to the message that contains the node reference. By adopting the result from [17], one can show that routing in the LDB can be done in $\mathcal{O}(\log n)$ rounds w.h.p.:

Lemma 3. *For any $p \in [0, 1)$, routing a message from a source node v to a node that is the predecessor of p in the LDB can be done in $\mathcal{O}(\log n)$ rounds w.h.p.*

B. Distributed Hash Table

In order to store the elements of our queue in a distributed fashion, we use a distributed hash table (DHT) that makes use of consistent hashing: Elements $e \in \mathcal{E}$ that should be stored in the DHT will be assigned a unique position $p(e) \in \mathbb{N}_0$ by SKUEUE. This position can then be hashed to a real-valued key $k(p(e)) \in [0, 1)$ via a publicly known pseudorandom hash function. A node v is responsible for storing all elements whose keys are within the interval $[v, succ(v))$. Thus, if we want to insert (resp. delete) an element $e \in \mathcal{E}$, we only have to search for the node v with $v \leq k(p(e)) < succ(v)$ and tell v to store e. The search for v can be performed in $\mathcal{O}(\log n)$ rounds according to Lemma 3. We will use the following operations in SKUEUE:

1) PUT(e, k) Inserts the element $e \in \mathcal{E}$ with key k into the DHT.
2) GET(k, v): Removes the element with key k from the DHT and delivers it to the initiator v of the request.

It is well known for consistent hashing that it is *fair*, meaning that each node stores the same amount of elements for the DHT on expectation.

Lemma 4. *Consistent Hashing is fair.*

III. ENQUEUE & DEQUEUE

Throughout this paper, a *queue operation* is either an ENQUEUE() or a DEQUEUE() request.

The main challenge to guarantee the sequential consistency from Definition 1 lies in the fact that messages may outrun each other, since we allow fully asynchronous computations and non-FIFO message delivery. In a synchronous environment, this would not be a problem. Another problem we have to solve is that the rate at which nodes issue queue requests

may be very high. As long as we process each single request one by one, scalability cannot be guaranteed.

The general idea behind SKUEUE is the following: First, we aggregate batches of queue operations to the leftmost node in the LDB, called *anchor*, by forwarding them to the leftmost neighbor at each hop. By doing so, every involved node implicitly becomes part of an *aggregation tree*. The anchor then assigns a position $p \in \mathbb{N}_0$ in the DHT for each queue operation and spreads all positions for the queue operations over the aggregation tree such that sequential consistency (Definition 1) is fulfilled. Nodes in the aggregation tree then generate PUT and GET requests for the respective positions in the DHT. We describe this approach in more detail now.

A. Operation Batch

Whenever a node initiates a queue operation, it has to buffer it in its local storage. We are going to represent the sequence of buffered queue operations by a *batch*:

Definition 5 (Batch). *A batch B (of queue operations) is a sequence $(op_1, \dots, op_k) \in \mathbb{N}_0^k$, for which it holds that for all odd i, $1 \leq i \leq k$, op_i represents the length of the i-th enqueue sequence. Similarly, for all even $i, 1 < i \leq k$, op_i represents the length of the i-th dequeue sequence. Denote the batch (0) as empty.*

We are able to *combine* two batches $(op_1, \dots, op_k)$ and $(op'_1, \dots, op'_l)$ by computing $B = (op''_1, \dots, op''_m)$ with $op''_i = op_i + op'_i$ and $m = \max\{k, l\}$ (we define $op_i = 0$ if $i > k$ and $op'_i = 0$ if $i > l$). If a batch B is the combination of batches $A_1, \dots, A_k$, then we denote $A_1, \dots, A_k$ as *sub-batches*. Each node may store two types of batches locally: One batch that is currently being processed and another batch that waits for being processed and acts as the buffer for newly generated queue operations. For a node v, we call the former batch $v.B$ and the latter one $v.W$. We denote v as the *owner* of the batch $v.B$.

Whenever a node v generates a queue operation op, we update the batch $v.W = (op_1, \dots, op_k)$ in the following way: If op is an ENQUEUE() request, then we increment op_k if k is odd, otherwise we add a 1 to the batch by setting $v.W = (op_1, \dots, op_k, 1)$. Similarly, if op is a DEQUEUE() request, we increment op_k, if k is even, otherwise we set $v.W = (op_1, \dots, op_k, 1)$. By doing so, the batch $v.W$ respects the local order in which queue operations are generated by v, which is important for guaranteeing sequential consistency.

B. Aggregation Tree

All (virtual) nodes in the LDB implicitly form an aggregation tree. In order to do this, a node v needs to know both, its parent and its child nodes in the tree. Both depend on whether v is a left, middle or right virtual node (see Figure 1 for an example).

The parent node $p(v)$ of some node v in the aggregation tree is always the node that is v's leftmost neighbor. More specifically, if v is a middle virtual node, then $p(v) = l(v)$. If

v is a left virtual node then $p(v) = pred(v)$. Finally, if v is a right virtual node, then $p(v) = m(v)$.

Next, we describe how a node v knows its child nodes (denoted by the set $C(v)$) in the aggregation tree, assuming that the node set is static (we describe how to handle JOIN() and LEAVE() requests in Section IV). If v is a middle virtual node, then either $C(v) = \{r(v), succ(v)\}$ (if $succ(v)$ is a left virtual node) or $C(v) = \{r(v)\}$ (otherwise). If v is a left virtual node, then either $C(v) = \{m(v), succ(v)\}$ (if $succ(v)$ is a left virtual node) or $C(v) = \{m(v)\}$ (otherwise). Last, if v is a right virtual node, then $C(v) = \emptyset$. Intuitively, each node has its next virtual node as a child and also its successor if that successor is a left node. A right virtual node cannot have a left virtual node as a right neighbor (as the id of a right virtual node is always at least 0.5 and the id of a left virtual node is always less than 0.5).

Fig. 1. A LDB consisting of 6 nodes (corresponding to 2 processes u and v). Bold linear/virtual edges define the corresponding aggregation tree.

Observe that nodes are able to find their connections in the tree by relying on local information only. Thus, for the rest of the paper we assume that every node knows its parent and child nodes in the aggregation tree at any time.

From Lemma 3, we directly obtain an upper bound for the height of the aggregation tree:

Corollary 6. *The aggregation tree based on the LDB has height $\mathcal{O}(\log n)$ w.h.p.*

We are now ready to describe our approach for queue operations in detail, dividing it into 4 stages.

C. Stage 1: Aggregating Batches

Every time a node v calls its TIMEOUT (see Algorithm 1) method, it checks whether its batch $v.B$ is empty and its batch $v.W$ contains the batches of all of its child nodes in the aggregation tree. If that is the case, then v transfers the data of $v.W$ to $v.B$ and sends out a message containing the contents of $v.B$ to $p(v)$. Additionally, v memorizes the sub-batches that are combined in $v.B$ such that it can determine the child node that sent the sub-batch to v. We proceed this way in a recursive manner, until the root node v_0 of the aggregation tree, denoted as *anchor* from now on, has received all batches from its child nodes. Then it combines these batches with its own batch $v_0.W$ into $v_0.B$ and switches to the next stage by locally calling ASSIGN (see Algorithm 2).

D. Stage 2: Assigning Positions

At the anchor v_0 we maintain two variables $v_0.first \in \mathbb{N}_0$ and $v_0.last \in \mathbb{N}_0$, such that the invariant $v_0.first \leq v_0.last + 1$ holds at any time. The interval $[v_0.first, v_0.last]$ represents

Algorithm 1 Stage 1 ▷ Executed by each node v

1: **procedure** TIMEOUT
2:     **if** $v.B = (0) \wedge v.W$ contains sub-batches from all
3:     $c \in C(v)$ **then**
4:         $v.B \leftarrow v.W$
5:         $v.W \leftarrow (0)$
6:         **if** v is the anchor node v_0 **then**
7:             ASSIGN($v_0.B$) ▷ Switch to Stage 2
8:         **else**
9:             $p(v) \leftarrow$ AGGREGATE($v.B$)
10: **procedure** AGGREGATE(B)
11:     $v.W \leftarrow v.W \cup B$

---

the positions that are currently occupied by elements of the queue, which implies that the current size of the queue is equal to $v_0.last - v_0.first + 1$.

Now we describe how the anchor processes its batch $v_0.B = (op_1, \ldots, op_k)$ at the start of this stage. Based on its variables $v_0.first, v_0.last$, v_0 computes intervals $[x_1, y_1], \ldots, [x_k, y_k]$ by processing each element in the batch $(op_1, \ldots, op_k)$ in ascending order of their indices i. If i is odd, then v_0 sets the interval $[x_i, y_i]$ to $[v_0.last + 1, v_0.last + op_i]$ and increases $v_0.last$ by op_i afterwards. Similarly, if i is even, then v_0 sets the interval $[x_i, y_i]$ to $[v_0.first, \min\{v_0.first + op_i - 1, v_0.last\}]$ and updates $v_0.first$ to $\min\{v_0.first + op_i, v_0.last + 1\}$ afterwards. By doing so, we assigned an interval to each sequence op_i of requests, implying that we can assign a position to each single queue operation of such a sequence (which is part of the next stage). Note that in case the queue is empty or does not hold sufficiently many elements and the anchor has to assign positions to some sequence of DEQUEUE() requests of length k, it either holds $x_i = y_i + 1$ (if the queue is empty) or $x_i - y_i < k$ for the computed interval $[x_i, y_i]$.

E. Stage 3: Decomposing Position Intervals

Once v_0 has computed all required position intervals $[x_1, y_1], \ldots, [x_k, y_k]$ for a batch, it starts broadcasting these intervals over the aggregation tree, by calling SERVE on its child nodes, see Algorithm 2. When a node v in the tree receives a collection $[x_1, y_1], \ldots, [x_{k'}, y_{k'}]$ of intervals, it decomposes the intervals with respect to each sub-batch $B_1, \ldots, B_l$ of $v.B$ (recall that v has memorized this combination). Consider a sub-batch $B_i = (op_1, \ldots, op_m)$ of $v.B$. We describe how v is able to assign a (sub-)interval to each op_i. Assume i is odd for op_i (corresponding to op_i many ENQUEUE() requests). Then v assigns the (sub-)interval $[x_i, x_i + op_i - 1]$ to op_i. Afterwards we update $[x_i, y_i]$ by setting $[x_i, y_i] = [x_i + op_i, y_i]$. This implies that every ENQUEUE() request is assigned a unique position.

Now assume i is even for op_i (corresponding to op_i many DEQUEUE() requests). Then v assigns the (sub-)interval $[x_i, \min\{x_i + op_i - 1, y_i\}]$ to op_i. Afterwards we set $[x_i, y_i] = [\min\{x_i + op_i, y_i + 1\}, y_i]$. This implies that DEQUEUE() requests are either assigned a position or immediately return

$\perp$ in case the interval is not large enough to assign a position to all DEQUEUE() requests.

Once each sub-batch of $v.B$ has been assigned to a collection of (sub-)intervals, we send out these intervals to the respective child nodes in $C(v)$. Applying this procedure in a recursive manner down the aggregation tree yields an assignment of a position to all ENQUEUE() and DEQUEUE() requests.

F. Stage 4: Updating the DHT

Now that a node v knows the exact position $p \in \mathbb{N}_0$ for each of its queue operations, it starts generating PUT and GET requests. For an request ENQUEUE(e) that got assigned to position p, v issues a PUT(e, $k(p)$) request to insert e into the DHT (recall that the key $k(p) \in [0, 1)$ is just the real-valued hash of p). This finishes the ENQUEUE(e) request. For a DEQUEUE() request that got assigned to position p, v issues a GET($k(p)$, v) request. Since in the asynchronous message passing model, it may happen that a GET request arrives at the correct node in the DHT *before* the corresponding PUT request, each GET request waits at the node responsible for the position k until the corresponding PUT request has arrived. This is guaranteed to happen, as we do not consider message loss.

Once a node has sent out all its DHT requests, it switches again to Stage 1, in order to process the next queue operations.

Algorithm 2 Stages 2-4

1: **procedure** ASSIGN(B) ▷ Executed by the anchor
2:     Compute intervals $I = [x_1, y_1], \ldots, [x_k, y_k]$ from B
3:     SERVE(I) ▷ Switch to Stage 3
4: **procedure** SERVE(I) ▷ Executed by each node v
5:     Decompose I depending on $C(v)$ and $v.B$
6:     **for all** $c \in C(v)$ **do**
7:         Forward sub-intervals $I_c \subset I$ to c via SERVE(I_c)
8:     Forward PUT/GET requests to the DHT
9:     $v.B \leftarrow (0)$ ▷ Return to Stage 1

We defer the analysis of the ENQUEUE() and DEQUEUE() requests to Section V.

IV. JOIN & LEAVE

When a process enters or leaves the system, this entails several changes to the system in order to get into the state assumed in Section III: The DHT has to be updated, which includes movement of data to joining or from leaving nodes, the LDB has to be updated and meanwhile the aggregation tree changes. To prevent chaos caused by the latter, we handle joins and leaves *lazily*. This means that a node v joining or leaving the network will be assigned a node u *responsible for v*. u then acts as a representative for v meaning that u takes over v's DHT data and emulates v in the case of v being a leaving node, or relays v's ENQUEUE() or DEQUEUE() requests in the case of v being a joining node. Only after a sufficiently large number of nodes has requested to join or leave the system (which is counted at the anchor), the system

enters a special state in which no further batches are sent out. During this state, joining nodes are fully integrated into the system (meaning they do no longer need a node responsible for them) and nodes that left can end being emulated. In the following, we will specify the details of this. Keep in mind that a node that requested to join the system and that is not yet fully integrated into the system is called a *joining node* and a node that requested to leave the system and that has not yet left is called a *leaving* node.

Note that if a process v wants to join or leave the network, we have to integrate or disconnect the three nodes $l(v), m(v), r(v) \in V$ into or from the system. Therefore, we generate a JOIN() or LEAVE() request for each of these three nodes separately. In the following we describe how one of these requests is handled.

A. Join

Assume a node v wants to join the system and further assume $v > v_0$ for now (we will consider the other case separately below). Then it sends a JOIN(v) request to a node w. We assume that if node v wants to join the system via JOIN(v) at node w, we route v from w to the node u such that $u < v < succ(u)$ or $succ(u) < u < v$ (in case the edge $(u, succ(u))$ closes the cycle) holds. We define u to be *responsible for* JOIN(v). u has the following tasks: First, it introduces itself to v. Second, it hands over to v all DHT data whose key is in v's interval. Any PUT or GET requests for data with keys in this interval u will forward to v from then on. Third, u considers v to be a child in its aggregation tree, meaning that v is able to send ENQUEUE() or DEQUEUE() requests via u. Fourth, u notifies the anchor that there is an additional node that has joined the system. For this, we extend the notion of a batch B from Definition 5, such that it stores an additional number $B.j \in \mathbb{N}_0$ representing the number of JOIN() requests that u is responsible for. Node u proceeds in the same manner as for the queue operations in Section III: It buffers the request in $u.W$ by adding 1 to $u.W.j$ and once $u.B$ is empty and u has received batches from every child, u transfers all ENQUEUE(), DEQUEUE() and JOIN() requests stored in $u.W$ to $u.B$ forwards the batch up in the aggregation tree. Any intermediate node, when combining batches $B_1, \ldots B_k$, calculates the sum of the $B_i.j$ values for the combined batch. This way the anchor learns a lower bound on the total number of joining nodes (note that additional nodes may have requested to join but knowledge of this has not yet reached the anchor).

Note that a node u may become responsible for several joining nodes $v_1, \ldots v_k$. In this case, everything written before still holds with one exception: Assume u is responsible for nodes $v_1, \ldots, v_k$ and becomes responsible for an additional node v' such that a node v_i is the closest predecessor of v'. Then u does not transfer the DHT from itself to v' but issues v_i to transfer the DHT data to v' and sends a reference of v' to v_i. Using this reference, v_i can forward any PUT or GET requests that fall within the remit of v'.

If the anchor can observe that the number of joining nodes exceeds the number of successfully integrated nodes when processing a batch, it sends the computed intervals down the aggregation tree as usual (c.f. Section III), but attaches a flag to the message indicating that the *update phase* should be entered (thus informing all nodes of this). In this phase, no node will send out a new batch until it has been informed that the update phase is over. Instead, nodes responsible for other nodes will fully integrate these nodes into the system. This works in the following way: When a node $u \neq v_0$ in the aggregation tree receives the intervals from its parent node, it proceeds as described in Section III, i.e., it splits the intervals, forwards intervals to its children and possibly sends out PUT and GET requests. Additionally, u stores the parent $p_{old}(u)$ in the aggregation tree it received the intervals from and all children $C_{old}(u)$ it forwards the intervals to. This is required because in the update phase the aggregation trees may change, but the acknowledgments that the joining nodes have been integrated successfully need to be aggregated via the old aggregation tree. That means that as soon as u has integrated all nodes it is responsible for (if any) and received acknowledgments from all nodes in $C_{old}(u)$ (if any), it sends an acknowledgment to $p_{old}(u)$ and forgets $C_{old}(u)$ and $p_{old}(u)$. v_0 behaves similar to any other node u, i.e., it also stores its old children, processes PUT and GET requests and also starts integrating nodes it is responsible for. However, when it has finished in doing so, and received all acknowledgments from the nodes in $C_{old}(v_0)$, it propagates down in the new aggregation tree a message indicating that the update phase is over (note that we consider the case of a joining node to the left of the anchor below). This is safe because it can be shown by induction that when v_0 has received acknowledgments from all its children, every node in the tree has finished integrating at least all joining nodes that were joining when the anchor entered the update phase. Once a node has received an indication that the update phase is over, it starts aggregating and sending out batches again. We now describe how integrating a joining node works.

Consider a node u that is responsible for $v_1, \ldots, v_k$. W.l.o.g., we assume $u < v_1 < \ldots < v_k < succ(u)$. u introduces v_i to v_{i+1} and vice versa for all $i \in \{1, \ldots, k-1\}$ and introduces $succ(u)$ to v_k and vice versa. Finally, u drops its connections to $v_2, \ldots, v_k$ and $succ(u)$.

Note that the nodes v_i already stored their corresponding DHT data from the point when u became responsible for them. Due to changes in the De Bruijn graph it may happen that PUT or GET requests do not need to be routed to the same target as before. However, if a PUT request is at a node v that is not responsible for storing the corresponding element e, v must have a neighbor that is closer to the node responsible for storing e. This is because whenever v removes an edge to a neighbor during join, it has learned to know a closer one in the same direction before. Thus v can forward it into the right direction. Similarly, if a GET request is at a node v that does not store the desired element e, v can wait until it either stores e or until it has learned to know a node that is closer to the target than itself. Since eventually our procedure forms the

correct De Bruijn topology, these requests will be answered.

a) Updating the Anchor: We now consider the special case, where at least one new node v's label is smaller than the label of the current anchor v_0. Then the node responsible for v is the node u with maximum label, i.e., $u = pred(v_0)$. u behaves as described before. However, when v_0 has received all acknowledgments from it children and integrated the nodes it is responsible for, it does not send out the message indicating that the update phase is over (note that v_0 can determine that a node $v < v_0$ has joined because its neighborhood to the left has changed). Instead, v_0 searches for the leftmost node v_0' and and then transfers its interval $[v_0.first, v_0.last]$ to v_0'. From that point on, v_0' will behave as the new anchor and send the message indicating that the update phase is over down in the new aggregation tree.

B. Leave

The general strategy for leaves is the following: For each leaving node v, the process emulating the left neighbor u of v creates a virtual node v' that acts as a replacement for v, i.e., v' will store v's DHT data, be responsible for the nodes v was responsible for and have the same connections as v had. As soon as this replacement has been created, the corresponding edges have been established, the edges to v have been removed, and all messages on their way to v have been delivered and successfully forwarded from v, v is safe to leave the system and does so. The challenge is to deal with neighboring leaving nodes: If v has a neighbor that is also leaving, then this neighbor does not want to establish a new edge, which might result in a deadlock situation. Thus, we have to prioritize leaves: Whenever two neighboring nodes u and v determine that they both want to leave, the one with the higher identifier postpones its attempt to leave until the other one has left the system. Since in any case there is a unique leftmost leaving node, there will always be a node that can leave the system, which inductively yields that all nodes eventually leave. To enable this, each node that calls LEAVE() first asks all its left neighbors if it is allowed to do so. Only if all of them acknowledge, it starts the actual procedure to leave. Note that a node u that acknowledged a right neighbor v that it may leave and becomes leaving afterwards has to wait with actually executing LEAVE() until that node has left (i.e., was replaced by a replacement).

One may ask how a leaving node v can determine that it has received and successfully forwarded all messages sent to it to v'. Therefore, we additionally assume that for each message sent via an edge in the system, an acknowledgment is sent back to the sending node (except for acknowledgments, for obvious reasons). Each node then stores, for each edge, the number of acknowledgments it is still waiting for. We assume also that a node knows all other nodes with incoming connections to it (this can, e.g., be achieved in that each node that establishes a new edge first introduces itself and waits for an acknowledgment before it uses the edge for any other messages). Then, v can ask all nodes with incoming connections to inform v once they have received all acknowledgments for messages sent to

v. Once v has received all responses, it knows that it does not receive any more messages. After forwarding the received messages to v' and receiving all acknowledgments for those, it knows it is safe to leave.

A left, middle, or right virtual node u that created a replacement v' for its right neighbor v is called the node *responsible for* v'. Note that v' may receive an additional LEAVE() request from a node w. In this case, the process emulating u would spawn an additional node w' and everything is carried out as though v' were a normal node. However, we say that u is also responsible for w'. This way a left, middle, or right virtual node may become responsible for a number of nodes. Similar to the joining of nodes, a node u responsible for at least another node sends an additional number $B.l \in \mathbb{N}_0$ in the batch B it sends out next, representing the number of LEAVE() requests that u has become responsible for since it last sent out a batch.

The rest is analogous to the join case: As soon as the number of leave requests falls below half of the number of nodes emulated, the anchor initiates the update phase during which each node u responsible for a set of nodes $v_1, \ldots, v_k$ deletes these nodes and updates the De Bruijn Graph accordingly. Once all acknowledgments for this have been propagated up in the tree, the update phase is left again. Note that both joins and leaves may be handled in the same update phase.

On a sidenote, one may ask what happens if a joining node v joins at some node w that is currently in the process of leaving. While w is alive and has edges to some non-leaving nodes, w can forward v such that v stays in the system. However, once w has left the system and is not alive anymore, v cannot join the system through w. Still, v can detect if w is not active anymore and then try joining the system from another node.

a) Updating the Anchor: When v_0 wants to leave, we proceed similar as for the join case: $pred(v_0)$ will become the node responsible for v_0 and perform the duties of the anchor and at the very end of the update phase, the anchor information is transferred to the node that then has the minimum identifier.

V. ANALYSIS

To prove that SKUEUE implements a distributed queue according to Definition 1, we define a total order on the ENQUEUE() and DEQUEUE() requests. To do so we specify an algorithm that assigns each request a unique value from $\mathbb{N}$: First, initialize a virtual counter c at the anchor with 1 as its initial value (this value is transferred if the anchor is changed due to a join or a leave). We assign a virtual counter to each ENQUEUE() or DEQUEUE() request op in the following way: Recall that when op is initiated, it causes the increase of an op_i value of one batch B. Virtually assign to $value(op)$ the new value of op_i. We also say that op belongs to B at index i. When B is combined with another batch on its way up in the aggregation tree, choose one of the batches as the first one and one as the second one. If B is the second one, let op_i' be the i-th entry of the other batch and add op_i' to $value(op)$. In any case, op belongs to the combined batch afterwards. Proceed in this way for every combination

of batches up to the anchor. When the anchor processes the batch $(op''_1, \ldots, op''_{k''})$ which op belongs to, add $c + \sum_{j=1}^{i-1} op_j$ to $value(op)$. Afterwards, the anchor updates c by $\sum_{j=1}^{k''} op_j$. Intuitively, imagine the anchor would process every request individually: Then it would first consider all op''_1 ENQUEUE() requests, then all op''_2 DEQUEUE() requests, and so on. The final value of op would then be the number of requests that the anchor has served up to (and including) op.

Observe that the values are unique. In the following, let $\prec$ be the order defined by the values given this way. The following lemmas follow from the protocol description (check the way we assigned values to the requests and how the intervals are assigned to the requests):

Lemma 7. *If, for two DEQUEUE() requests $\mathrm{DEQ}_{u,i}$, $\mathrm{DEQ}_{v,j}$ that get assigned positions, pos_a, pos_b, respectively, $\mathrm{DEQ}_{u,i} \prec \mathrm{DEQ}_{v,j}$, then $pos_a < pos_b$.*

Lemma 8. *If, for two ENQUEUE() requests $\mathrm{ENQ}_{u,i}$, $\mathrm{ENQ}_{v,j}$ that get assigned positions, pos_a, pos_b, respectively, $\mathrm{ENQ}_{u,i} \prec \mathrm{ENQ}_{v,j}$, then $pos_a < pos_b$.*

Lemma 9. *If a DEQUEUE() request $\mathrm{DEQ}_{u,i}$ gets assigned a position pos_a, then for every ENQUEUE() request $\mathrm{ENQ}_{v,j}$ with $value(\mathrm{DEQ}_{u,i}) < value(\mathrm{ENQ}_{v,j})$ the position pos_b assigned to it satisfies $pos_b > pos_a$. Likewise, if an ENQUEUE() request $\mathrm{ENQ}_{u,i}$ gets assigned a position pos_a, then for every DEQUEUE() request $\mathrm{DEQ}_{v,j}$ with $value(\mathrm{DEQ}_{v,j}) < value(\mathrm{ENQ}_{u,i})$ the position pos_b assigned to it satisfies $pos_b < pos_a$.*

Lemma 10. *Assume there is a sequence of DEQUEUE() requests $deq_1, \ldots, deq_k$ that belong to the same batch B and the same index l such that $value(deq_1) < \cdots < value(deq_k)$. If deq_i returns $\perp$ for some $i \in \{1, \ldots, k\}$, then all deq_j with $i < j \le k$ also return $\perp$ and $[v_0.first, v_0.last]$ is empty after index l of batch B has been processed in v_0 in Stage 2.*

This lemma directly implies:

Corollary 11. *If a DEQUEUE() request $\mathrm{DEQ}_{u,i}$ returns $\perp$ then every $\mathrm{DEQ}_{v,j}$ request with $value(\mathrm{DEQ}_{u,i}) < value(\mathrm{DEQ}_{v,j})$ does not return an element e added by an ENQUEUE() request $\mathrm{ENQ}_{w,k}$ with $value(\mathrm{ENQ}_{w,k}) < value(\mathrm{DEQ}_{u,i})$.*

Lemma 12. *If a DEQUEUE() request $\mathrm{DEQ}_{u,i}$ gets assigned a position pos, then for every ENQUEUE(e) request that received a position $pos' < pos$ there exists a $\mathrm{DEQ}_{v,j}$ request with $value(\mathrm{DEQ}_{v,j}) < value(\mathrm{DEQ}_{u,i})$ that returns e.*

The reason is that the dequeue intervals always start with the lowest possible value.

Lemma 13. *Every GET operation issued by any of the nodes is answered in finite time.*

a) Proof sketch: Note that the way SKUEUE deals with leave requests makes sure that no messages get lost during leave as was argued in Section IV. Furthermore, check in the protocol description that whenever a GET message is at a node u that is not responsible for storing the position corresponding with the GET message, u knows a node that is closer to the node responsible for storing the position. Thus, each GET message will eventually reach the node that is responsible for storing it (note that even if the node responsible for storing it changes meanwhile, then the old node responsible for storing it knows the new one and can forward the message accordingly). If that node already stores the element required by the GET message, it can be answered directly. Otherwise, check that the same we said about the GET message analogously applies to the corresponding PUT message. Thus, the element will eventually arrive at that node and the GET message can be answered.

We are now ready to prove the following theorem:

Theorem 14. SKUEUE *implements a data structure that is sequentially consistent.*

Proof. First of all note that due to the protocol description and Lemma 13, every DEQUEUE() request returns a value (i.e., either $\perp$ or some element $e \in \mathcal{E}$). We will consider all four requirements of Definition 1 individually.

First, consider an arbitrary DEQUEUE() request $\mathrm{DEQ}_{w,j}$ that returns a value $e \in \mathcal{E}$ that was added due to an ENQUEUE() request $\mathrm{ENQ}_{v,i}$. Since the position in the DHT is the same for both these requests, Lemma 9 implies that $value(\mathrm{ENQ}_{v,i}) < value(\mathrm{DEQ}_{w,j})$.

Second, again consider an arbitrary DEQUEUE() request $\mathrm{DEQ}_{w,j}$ that returns a value $e \in \mathcal{E}$ that was added due to an ENQUEUE() request $\mathrm{ENQ}_{v,i}$. For the first part, assume for contradiction that there is a $\mathrm{DEQ}_{u,k}$ that returns $\perp$ with $value(\mathrm{ENQ}_{v,i}) < value(\mathrm{DEQ}_{u,k}) < value(\mathrm{DEQ}_{w,j})$. Then, Corollary 11 implies that $\mathrm{DEQ}_{w,j}$ cannot return e, which is a contradiction. For the second part, assume for contradiction that there is an $\mathrm{ENQ}_{u,k}$ whose element $e' \in \mathcal{E}$ is never returned with $value(\mathrm{ENQ}_{u,k}) < value(\mathrm{ENQ}_{v,i}) < value(\mathrm{DEQ}_{w,j})$. Combining Lemma 8 with Lemma 12 yields the desired contradiction also here.

For the third requirement, consider an arbitrary DEQUEUE() request $\mathrm{DEQ}_{v,j}$ that returns a value $e \in \mathcal{E}$ that was added due to an ENQUEUE() request $\mathrm{ENQ}_{u,i}$ and an arbitrary DEQUEUE() request $\mathrm{DEQ}_{x,l}$ that returns a value $e' \in \mathcal{E}$ that was added due to an ENQUEUE() request $\mathrm{ENQ}_{w,k}$. For the first part, assume for contradiction that $value(\mathrm{ENQ}_{u,i}) < value(\mathrm{ENQ}_{w,k}) < value(\mathrm{DEQ}_{x,l}) < value(\mathrm{DEQ}_{v,j})$. Lemma 8 yields that for the positions pos_a and pos_b assigned to $\mathrm{ENQ}_{u,i}$ and $\mathrm{ENQ}_{w,k}$, respectively, $pos_a < pos_b$ holds. Note that pos_a is assigned to $\mathrm{DEQ}_{v,j}$ and pos_b is assigned to $\mathrm{DEQ}_{x,l}$. However, Lemma 7 would imply $pos_b < pos_a$, which yields a contradiction. The second part of the third requirement is analogous.

The fourth requirement is directly satisfied by the way we defined $\prec$. This completes the proof of the theorem. $\square$

In the following, we want to analyze the runtime of the operations ENQUEUE(), DEQUEUE(), JOIN() and LEAVE(). We start with ENQUEUE() and DEQUEUE() requests.

Theorem 15. *Each request* ENQUEUE() *or* DEQUEUE() *needs* $\mathcal{O}(\log n)$ *rounds w.h.p. until it is processed correctly on the distributed queue.*

Proof. Consider an arbitrary request $op \in \{$ENQUEUE(), DEQUEUE()$\}$. Assume an op is generated by some node $v \in V$. By Corollary 6 we need $\log n$ rounds w.h.p. to transfer op to the anchor node v_0 (Stage 1) as part of a batch. Thus it takes $\log n$ rounds w.h.p. to assign a position to each request (Stages 2 and 3). Finding the corresponding node u for the position in the DHT and transferring the PUT/GET operation for op takes again $\log n$ rounds w.h.p. by Lemma 3. Note that if $op = $ DEQUEUE(), then we only have a constant message overhead for GET, as u is able to send the result of GET to v in one round. Summing it all up, we need $\mathcal{O}(\log n)$ number of rounds w.h.p. □

We obtain the following corollary, which shows that our approach is indeed scalable for a large number of incoming requests.

Corollary 16. *Assume a node $v \in V$ has stored an arbitrary amount of queue requests in $v.W$. The number of rounds, needed to process all requests successfully is $\mathcal{O}(\log n)$ w.h.p.*

Proof. Follows from Theorem 15 and the fact that we process requests in batches. □

Corollary 16 emphasizes the advantages of processing multiple requests at once via batches: Imagine a node v that generates one queue request in each round. If a single queue request op takes $\mathcal{O}(\log n)$ rounds to finish and v is prohibited to process any further request before op is finished, v's local storage would eventually overflow. For SKUEUE however, v is able to flush all requests contained in $v.B$ after $\mathcal{O}(\log n)$ rounds w.h.p.

Theorem 17. *Assume that at the beginning of the update phase there are n joining nodes ($n/2$ node replacements). Then the update phase finishes after $\mathcal{O}(\log n)$ rounds w.h.p., if no node wants to join/leave the system in the meantime.*

Proof. By Corollary 6, we need $\mathcal{O}(\log n)$ rounds w.h.p. to propagate the start of the update phase to all nodes in the aggregation tree. It is easy to see that a node v responsible for multiple JOIN() or LEAVE() requests can process these requests in a constant amount of rounds once it got the permission from $p(v)$ in the update phase. The only case that may exceed the claimed upper bound is the case where the (old) anchor transfers its data to the new anchor, i.e., to the node with minimal label. However, with n nodes joining, each old node is only responsible for at most $\mathcal{O}(\log n)$ joining nodes w.h.p. This implies that there are only $\mathcal{O}(\log n)$ joining nodes w.h.p. with smaller label than the anchor. The same argumentation holds for leaving nodes. □

Now we want to analyze the size of messages that are sent over communication channels in the network. Obviously, the messages containing the most data are the ones containing a batch. Thus, we want to get an upper bound on the maximum batch size.

Theorem 18. *Batches representing* ENQUEUE(), DEQUEUE(), JOIN() *and* LEAVE() *requests have size $\mathcal{O}(\log n)$ w.h.p. if each node generates one such request per round.*

Proof. Note that JOIN() and LEAVE() requests a node v is responsible for are represented in a batch by a single constant. By Theorem 15, each batch $v.W$ containing ENQUEUE(), DEQUEUE(), JOIN() and LEAVE() requests needs $\mathcal{O}(\log n)$ rounds w.h.p. until it is processed. Therefore a batch $v.W$ of some node v can only have a size up to $\mathcal{O}(\log n)$ w.h.p. until it is sent out via $v.B$ assuming each node generates one request per round: The size of the batch increases only if the type (ENQUEUE() or DEQUEUE()) of the request generated in round s_i differs from the type of the request generated in round s_{i-1}. □

Finally we note that SKUEUE is fair regarding the number of elements that each node has to store. This immediately follows from the fairness property of the DHT (Lemma 4) and the fact that each joining or leaving node gets or transfers its DHT data.

Corollary 19. SKUEUE *is fair.*

VI. DISTRIBUTED STACK

In this section we propose some simple modifications to SKUEUE in order to realize a scalable distributed stack that fulfills sequential consistency. Instead of ENQUEUE() and DEQUEUE() requests, the stack provides requests PUSH() and POP() such that for a single process it resembles a LIFO data structure. Definition 1 can then be adjusted easily.

A natural approach would be to just change the way in which the anchor computes the position intervals for DEQUEUE() requests (see Stage 2 in Section III). Recall that the anchor computes the interval $[v_0.first, \min\{v_0.first + op_i - 1, v_0.last\}]$ in case there are op_i consecutive DEQUEUE() requests. For op_i consecutive POP() requests, we want the anchor to return the interval $[\max\{1, v_0.last - op_i + 1\}, v_0.last]$ and update $v_0.last$ to $\max\{0, v_0.last - op_i\}$ afterwards. Observe that we do not need the variable $v_0.first$ anymore. Processes decomposing their position intervals in stage 3 now have to take out the maximum position in the interval first. Unfortunately, this modification does not suffice on its own, because the assigned positions for inserted elements are not unique: For the operation sequence $($PUSH(x), POP$()$, PUSH$(y))$ both PUSH() requests are assigned to the same position by the anchor, leading to elements being replaced in the DHT. Therefore we have to make sure that the key under which elements are inserted into the DHT is unique: We introduce a variable $v_0.ticket \in \mathbb{N}$ at the anchor, which is increased by i every time $v_0.last$ is increased by i, but is never decreased, i.e., $v_0.ticket$ is monotonically increasing. Intuitively, $v_0.ticket$ represents the number of PUSH() requests ever processed at the anchor, whereas $v_0.last$ represents the current size of the stack. A request is now

assigned a pair $(position, ticket) \in \mathbb{N} \times \mathbb{N}$ instead of just a single position. For such a pair (p, t) that got assigned to a PUSH(x) request, we store (p, t) and x at the node that is responsible for position p in the DHT. A POP() request that got assigned to the pair (p', t') searches the DHT for the node v that is responsible for position p'. After arrival at v, we remove the element with ticket $t \le t'$ from v and return it to the initiator of the POP() request.

Nodes are able to locally combine generated requests in order to answer them immediately: For instance, if node v generates k PUSH() requests $p_1, \ldots, p_k$ followed by k POP() requests $po_1, \ldots, po_k$, then v can process all of these requests immediately by assigning the $k - i + 1$-th PUSH() request to the i-th POP() request for all $i \in \{1, \ldots, k\}$. This is particularly advantageous in scenarios where the rate at which nodes generate requests is very high. It is easy to see that we do not violate sequential consistency with this modification. Furthermore it follows that all batches which are sent upwards the aggregation tree have the form $B = (op_1, op_2)$ with $op_1 \in \mathbb{N}$ representing POP() operations and op_2 representing PUSH() operations. This immediately yields the following theorem on the size of a batch:

Theorem 20. *Batches representing* PUSH() *and* POP()*, requests have constant size.*

In contrast to Theorem 18, Theorem 20 holds for any rate in which nodes generate stack requests.

Since we consider the asynchronous message passing model, all that is left is to prevent the following scenario from happening: Consider the operation sequence (a, b, c, d) with $a = $ PUSH(x), $b = $ POP(), $c = $ PUSH(y) and $d = $ POP(). Then the anchor assigns the pair (p, t) to a, (p, t) to b, $(p, t + 1)$ to c and $(p, t + 1)$ to d. Due to asynchronicity in our system, the DHT requests representing a, b, c and d may arrive in the order (a, d, c, b) at the node responsible for position p. This leads to d returning the element x, as the ticket value for a is smaller than the ticket value for d. Request b does not find an element with ticket value smaller or equal than its own and consequently fails, violating sequential consistency. In order to fix this, we force all nodes v to wait in stage 4 before switching to stage 1 again, until all DHT-operations that v has generated in stage 4 have been finished (we just have to add this constraint to the clause in lines 2-3 of Algorithm 1). Reconsidering the above example, it follows that the order of arrival of the DHT operations will be either (a, b, c, d) or (a, c, b, d), because a and d are guaranteed to be in different batches than b and c when combining requests as described above. It is easy to see that both cases prevail sequential consistency. We obtain the main result of this section:

Theorem 21. *The modified* SKUEUE *protocol implements a stack that is sequentially consistent.*

JOIN() and LEAVE() requests are processed in the exact same manner on the stack as described in Section IV.

VII. EVALUATION

We implemented and evaluated SKUEUE as well as its stack adaptation (see Section VI) on different instances. In this section we present and interpret the most important results of these experiments.

A. Setup

We implemented the protocols for the synchronous message passing model and performed the following experiment for instances up to 100000 nodes: At the beginning of each (synchronous) round, we generate 10 queue requests and assign them to random nodes in the system. After 1000 rounds we stop the generation of requests and wait until all requests that are still being processed have finished successfully. For each finished request we measure the number of rounds it took the requests to finish. For the results presented in this section we always consider the average amount of rounds per requests. We tested instances with different ratios of EN-QUEUE()/DEQUEUE() requests, respectively, PUSH()/POP() requests.

B. Distributed Queue

Consider Figure 2 for results on the distributed queue.

Fig. 2. Average number of (synchronous) rounds per request on the distributed queue. The graphs represent the different probabilities p that a generated request is an ENQUEUE() operation, meaning that $1 - p$ is the probability that a generated request is a DEQUEUE() operation.

One can see that the average number of rounds for a request to finish scales logarithmically in the number of nodes n. As soon as the ENQUEUE() rate drops below 0.5 the queue performs better, because the queue is empty most of the times. This implies that DEQUEUE() operations do not have to search for a position in the DHT, as they can be processed immediately as soon as the requesting node receives the position intervals from the anchor. Interestingly, the curves for ENQUEUE() rates of 0.5 or higher are almost the same, which means that DEQUEUE() operations waiting for the corresponding ENQUEUE() operations in the DHT do not have a significant impact on the performance.

Roughly, these curves correspond to 3 times the height of the aggregation tree (denoted as $ATH \approx \log n$) plus the average number of rounds it takes for a DHT operation to finish: A queue request first has to wait after generation until the next aggregation phase begins (on average ATH rounds), then it is aggregated to the root (ATH rounds) and assigned a position (ATH rounds). Afterwards we process the corresponding DHT operation in approximately $\log n$ rounds.

C. Distributed Stack

Consider Figure 3 for results on the distributed stack.

Fig. 3. Average number of (synchronous) rounds per request on the distributed stack. The graphs represent the different probabilities p that a generated request is a PUSH() operation, meaning that $1 - p$ is the probability that a generated request is a POP() operation.

Same as for the queue, the average number of rounds for a request scales logarithmically in the number of nodes n. However, the stack performs a bit slower than the queue, because we wait at the end of stage 4 until all DHT operations have finished. This delays the start of the next aggregation phase and leads to all curves representing PUSH() ratios greater than 0 being roughly the same. Obviously the stack performs better if we only generate POP() operations. In fact, the curve for a PUSH() ratio of 0 is the same as the corresponding curve for the queue, which makes sense, since both data structures do not have to issue any DHT operations.

Unfortunately, we cannot see the impact of the local combination of operations in this setting, because the probability that more than one operation is generated at a node v in the same aggregation phase is very low. Therefore we perform an additional experiment: We consider an instance of $n = 10000$ nodes and generate requests at nodes with constant probability $p \in \{0.05, 0.1, 0.15, 0.2, 0.25, 0.5, 1\}$ at each round. For instance, if $p = 1$, we generate one request at each node in each round leading to $1000n = 10^7$ generated requests after 1000 rounds. The probability that a generated request is an ENQUEUE()/PUSH() operation is 0.5. Again we looked at the average number of rounds it takes a request to be processed successfully for both, the queue and the stack. The results

can be seen in Figure 4 (note that the horizontal axis now represents the different probabilities p mentioned above).

Fig. 4. Average number of (synchronous) rounds per request on the queue/stack with different request ratios and $n = 10000$.

Here we can see that the stack's performance gets even better if the rate at which requests are generated increases. This is due to nodes issuing multiple requests in the same aggregation phase, which leads to the stack being able to combine operations locally, such that they can be processed immediately.

VIII. CONCLUSION

We presented the protocol SKUEUE for a distributed queue that guarantees sequential consistency and is able to process requests fast even for a high rate of incoming requests.

A challenging task would be to make SKUEUE self-stabilizing, such that the network can recover itself from faulty states. However, due to the various amount of variables that have to be stored at each node and the fact that we are in an asynchronous environment, one will quickly have to weaken the queue semantics.

REFERENCES

[1] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.
[2] C. G. Plaxton, R. Rajaraman, and A. W. Richa, "Accessing nearby copies of replicated objects in a distributed environment," in *ACM SPAA*, 1997, pp. 311–320.
[3] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *STOC*, 1997, pp. 654–663.
[4] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM*, 2001, pp. 149–160.
[5] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware*, 2001, pp. 329–350.
[6] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, 2004.
[7] A. Lakshman and P. Malik, "Cassandra: structured storage system on a P2P network," in *PODC*, 2009, p. 5.

[8] S. Alaei, M. Toossi, and M. Ghodsi, "Skiptree: A scalable range-queryable distributed data structure for multidimensional data," in *ISAAC*, 2005, pp. 298–307.

[9] B. Kröll and P. Widmayer, "Distributing a search tree among a growing number of processors," in *SIGMOD*, 1994, pp. 265–276.

[10] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *PODC*, 1996, pp. 267–275.

[11] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," *J. Parallel Distrib. Comput.*, vol. 70, no. 1, pp. 1–12, 2010.

[12] N. Shavit and I. Lotan, "Skiplist-based concurrent priority queues," in *IPDPS*, 2000, pp. 263–268.

[13] M. Moir and N. Shavit, "Concurrent data structures," in *Handbook of Data Structures and Applications.*, 2004.

[14] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Scalable flat-combining based synchronous queues," in *DISC*, 2010, pp. 79–93.

[15] N. Shavit and G. Taubenfeld, "The computability of relaxed data structures: queues and stacks as examples," *Distributed Computing*, vol. 29, no. 5, pp. 395–407, 2016.

[16] C. Scheideler and S. Schmid, "A distributed and oblivious heap," in *ICALP*, 2009, pp. 571–582.

[17] A. W. Richa, C. Scheideler, and P. Stevens, "Self-stabilizing de bruijn networks," in *SSS*, ser. Lecture Notes in Computer Science, vol. 6976. Springer, 2011, pp. 416–430.

[18] M. Naor and U. Wieder, "Novel architectures for P2P applications: The continuous-discrete approach," *ACM Trans. Algorithms*, vol. 3, no. 3, p. 34, 2007.

[19] M. Herlihy, S. Tirthapura, and R. Wattenhofer, "Competitive concurrent distributed queuing," in *PODC*, 2001, pp. 127–133.

[20] S. Tirthapura and M. Herlihy, "Self-stabilizing distributed queuing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 7, pp. 646–655, 2006.

[21] G. Sharma and C. Busch, "Distributed queuing in dynamic networks," *Parallel Processing Letters*, vol. 25, no. 2, 2015.