DhtFlex: A Flexible Approach to Enable Efficient Atomic Data Management Tailored for Structured Peer-to-Peer Overlays

Udo Bartlang
Siemens AG
Corporate Technology
Information & Communications
Otto-Hahn-Ring 6, D-81739 Munich
udo.bartlang.ext@siemens.com

Jörg P. Müller
Clausthal University of Technology
Department of Informatics
Julius-Albert-Str.4, D-38678 Clausthal-Zellerfeld
joerg.mueller@tu-clausthal.de

Abstract—Distributed hash tables (DHTs) build on structured peer-to-peer overlays because of their inherent high scalability and resilience against node failures. Combined with additional replication strategies such systems promise high availability for published data resources. However, regarding the support for atomic data operations replication comes at the cost of maintaining consistency. Thus, most DHT like systems either focus on immutable data resources or if supported at all they reinvent the wheel for their storage needs strictly focussing on mutable data and their own narrow application domain. What lacks is a generic but efficient solution to enable flexible consistent data operations for replicated data trimmed for highly concurrent and fluctuating environments. In this paper we introduce DhtFlex, a fault-tolerant distributed algorithm tailored for the needs of a DHT and optimized for the consistent management of replicated data resources in such environments. DhtFlex is supposed to serve as a generic building block to any underlying structured peer-to-peer overlay. It imposes an annotated resource concept to typify replicated data. DhtFlex enables efficient support for immutable as well as for optimized atomic operations on mutable data resources.

I. INTRODUCTION

The peer-to-peer (P2P) communication approach has proven to be practically successful in various domains of distributed applications, ranging from file sharing [1] across voice over IP [2] to IPTV [3]. Interestingly, although “advanced data management” seems to be an obvious domain when thinking about the power of P2P computing, there lack breakthrough applications in real life. Existing solutions that offer advanced distributed data management equipped with strong consistency guarantees like distributed file systems [4], [5], typically follow the classical client-server principle. Here, structured P2P overlays have the potential to close the gap by providing a powerful foundation to build high scalable distributed hash tables (DHTs). Combined with replication strategies, a DHT system promises high data availability, scalability and resilience against node failures. As a common approach to enhance fault tolerance, a certain data resource instance is stored replicated at different physically located peers, called its replication group. However, regarding atomic data operations replication comes at the cost of maintaining consistency, i.e., an atomic data operation on a certain resource has to be consistently applied to all of its replicas. In this case, the existence of several replicas is crucial and raises the challenge if a data resource could be concurrently modified. So, most DHT like systems either avoid the difficulty and focus on immutable data resources [6], [7], or rather limit concurrent data resource modifications allowing only one dedicated modifier, the resource’s owner [8]. Those rare P2P systems that allow concurrent atomic data operations [9] are usually monolithic reinventing the wheel for their storage needs with focus on their specific application domain, overlay, and strictly on mutable data resources. One the one hand, this hampers the adoption by other applications; on the other, applications may support both explicit immutable and mutable data resources. However, given the vantages of a structured P2P overlay and the requirements of a DHT reveals potential worth profiting. What lacks is a generic but efficient solution to enable flexible consistent data operations for replicated data trimmed for such a highly concurrent and fluctuating environment.

In this paper we introduce DhtFlex, a full-blown fault-tolerant distributed algorithm tweaked for the needs of a DHT and optimized for the consistent management of replicated data resources. DhtFlex may serve as a generic building block to every underlying structured P2P Overlay. It annotates resources using a type concept to efficiently cope with immutable, as well as mutable data resources. Especially for the latter, DhtFlex is able to provide strong consistency guarantees enabling atomic put and get operations. Therefore, it exploits techniques of Leslie Lamport’s famous Paxos algorithm to coordinate the recast process of a resource’s replication group. As peers within a replication group may come and go, it is necessary to preserve the consistent formation of a single replication group. To accelerate concurrent put and get requests DhtFlex serializes them over the master of a replication group. Hence, DhtFlex is optimized for get, put and recast operations on mutable data resources, in that sequence. These operations are even more efficiently supported for immutable data re-
sources, increasing the overall performance in an employed system.

The rest of the paper is structured as follows. Section II provides rudimentary background knowledge. Next, Section III compares DhtFlex to available related work. The system model and architecture of DhtFlex is defined in Section IV. Its functionality is explained in Section V. In Section VI we sentimentiously discuss our approach regarding safety, liveness and performance. Finally, Section VII concludes the paper and gives an outlook on future work.

II. BACKGROUND

The starting point of the work described in this paper is our JSR-170 compliant P2P content repository [10]. JSR-170 [11] specifies a standard API to access content repositories in Java™ independently of a concrete implementation, fostering interoperability, portability, and adaptability. The repository employs a DHT background and combines the flexibility of its distributed nature with the uniformity of the involved standard, facilitating its adoption. In addition to basic storage capabilities like read and write operations, it integrates advanced services commonly required by content-centric applications, e.g., versioning, locking, or rich queries. On a logical level, the repository consists of an unlimited set of workspaces, each establishing a hierarchical, n-ary tree-based view of items, i.e., data resources. In order to uniquely identify an item, it is always referenceable through some universally unique identifier (UUID), allowing a direct way to address it. JSR-170 allows the typecast of items, providing further information to raw data. The DhtFlex approach is motivated by the observation that it is typical for such an advanced distributed data management system to deal with mutable, as well as immutable data, e.g., once it is defined, a certain version of a content item within a version chain remains unchanged forever. Hence, DhtFlex is optimized by explicitly differentiating both data resource types and presents an efficient building block to cope with replicated data in P2P systems.

A distributed hash table (DHT) [7] offers a key-value based distributed storage for data resources. The major benefits are its inherent large scalability and robustness using replication techniques. A DHT builds on structured P2P overlay networks [12], [13], [14], [15], [16] which impose a graph structure on the participating peers assigning key space responsibilities per peer identifier, i.e., peer id. Hence, an overlay composed of N different peers allows to locate a data resource of a certain key within O(log N) communication steps while maintaining logarithmic state information at each peer. The drawback of DHTs is their typical focus on immutable data resources when using replication. Although there has been a lot of research, it lacks a generic building block to support both, immutable as well as mutable data within a structured P2P overlay. DhtFlex is intended to close this gap.

One option to enforce strong consistency guarantees for operations on mutable, replicated data resources is to serialize all operations using distributed consensus techniques. However, asynchronous distributed systems, e.g., today’s Internet, are faced with the FLP impossibility [17]. Lamport’s Paxos algorithm [18], [19] was one of the first practical solutions to cope with this issues, assuming a benign crash-recovery failure model. Paxos is a three-phase commit protocol and uses a quorum based approach to reach distributed consensus between participating nodes. In simple terms, as long as a quorum of nodes is correctly working progress is guaranteed.

Figure 1 sketches the basic order of interaction events between three nodes executing Paxos. In the first phase, the so called “read phase”, a potentially master tries to collect information about already existing proposal values, e.g., from an old master. In the second, the “write phase” the master propagates a consistent value. This is either the learned value from the previous phase, if existing, or an own proposal value. If a quorum of positive acknowledgements is achieved, consensus is finally reached and the master can announce the decision in phase three. The whole process roughly requires five message transmission delays. Paxos guarantees that if phase three is reached by multiple nodes, all consistently commit on the same proposal value. DhtFlex builds on the Paxos principle to reach a total-ordering of the replication group castings for a certain mutable data resource. However, for ordinary put and get operations this overhead is not necessary, resulting in a faster execution. Operations on immutable data resources can be processed even faster by DhtFlex.

III. RELATED WORK

In part, DhtFlex revisits a problem approached by Etna [20]. Etna claims to support atomic mutable data within a Chord-based DHT [12]. As our solution it builds on Paxos to ensure the consistency of a data resource’s replication group. Although its basic ideas are sketched no results have been published yet. Unfortunately, Etna is not practicable in real world P2P systems because of its relying on a crash-stop failure model; in practice, it is highly preferable that peers which temporarily crash are allowed to recover with the identifier, i.e., peer id. We also discovered its shortcoming treating an initial put operation, resulting in concurrency inconsistencies. This way, DhtFlex can be seen as a consequent advancement to build a practical solution allowing crash-recovery and a consistent handling of put operations. DhtFlex even outperforms Etna’s recast handling by reducing the communication steps, or message transmission delays, from seven to five or rather three, in ideal situations (cf. Section VI). Because of our annotated
resource concept, we are actually more high-performance for immutable data resources. Contrary to Etna’s attempt of a system-wide replication factor DhtFlex is able to define the size of a resource’s replication group dynamically at runtime allowing even more flexible storage policies. In addition, DhtFlex is designed as a generic building block on top of any structured P2P overlay, not just supporting Chord. Lynch et al. present an atomic data access extension for DHTs using a state machine replication approach to enforce replica consistency [21]. The focus is more on providing atomicity in the case of peers joining or leaving the system. Contrary to DhtFlex the approach heavily interferes in the working of the underlying structured overlay, drawing no clear interface. Moreover, peers are expected not to fail, which is not feasible on real distributed systems, e.g., the Internet, where churn, i.e., the dynamics of peer membership participation, can occur. Regarding performance no evaluation is given, but implementing a DHT node as fully replicated state machine is likely to be worse than our solution. It even may undermine the overall performance of a DHT regarding scalability. As opposed to DhtFlex which has its focus on the exploitation of the benefits offered by a P2P overlay boosting atomic DHT operations.

IV. SYSTEM

A. Model

DhtFlex assumes a system composed of a dynamic set of non-malicious peers \(\{p_1, p_2, \ldots, p_n\}\) which follow a benign failure model. Each peer is uniquely identified by a UUID, its peer id. A running peer correctly works at its own speed obeying its specification. Any peer might fail by crashing, i.e., it stops executing. It is able to join or leave the system at any time. A peer can even recover with its original peer id by executing some recovery procedure. Recovery requires essential resource information to be recorded to stable storage (e.g., redundant disks). Therefore, a peer is equipped with volatile and stable memory. The latter is assumed not to be affected by a crash and can always be recovered. An unstable peer might crash and recover “infinitely” many times. For message transfer we build on a partial synchronous environment. Messages might be lost and there is no upper bound on message transmission delays. In order to use timeout mechanisms a peer has access to some local clock.

B. Architecture

DhtFlex is supposed to bridge the gap between the requirements of a DHT and the benefits offered by a structured key-based routing overlay. Therefore, it is designed bundling up its functionality to act as a generic building block in a modular environment, illustrated in figure 2. It communicates with the other components over clearly defined interface functions (cf. figure 3), simplifying its adoption and implementation.

DhtFlex assumes an underlying communication system which offers basic send and receive primitives. It can work on top of any structured P2P overlay. In order to guarantee interoperability we adopted the general key-based routing API [22]. In fact, we were able to further simplify the interface. An underlying key-based routing overlay only has to support one method replicateSet(). Hence, DhtFlex uses the routing overlay to perform the peer lookup processing. All data and replica management is controlled by DhtFlex separating responsibilities. DhtFlex primarily uses replicateSet() to build the replication group for a certain resource, i.e., to obtain information about appropriate peers. But it does not rely on the received set of peers as its configuration can rapidly change regarding churn. DhtFlex is able to deal with such inconsistency to allow the correct adjustment of replication groups.

V. FUNCTIONALITY OF DHTFLEX

This section introduces the distributed fault-tolerant DhtFlex algorithm. Our approach is tweaked using the possibilities of a DHT overlay and supports flexible data operations. The algorithm is fault-tolerant offering high data availability and efficient regarding concurrent operations on mutable or immutable data resources. Subsequently, the major building blocks of DhtFlex are presented.

A. Annotated Data Resources

The key idea in order to support efficient, flexible put, get, and recast operations is the utilization of P2P overlay peculiarities by the concept of annotated data resources. For

![Fig. 2. Interactions of the Major Building Blocks of DhtFlex’s Environment](image)

![Fig. 3. Interface Functions Provided by the Several Components of DhtFlex’s Environment](image)
example, DhtFlex serializes atomic operations on mutable data resources over a certain peer of its replication group, in contrast to its behaviour on immutable data resources. This "replication knowledge" is persistently attached to a data resource to allow fine-grained policies at resource level. Hence, a data resource for a certain key or \(id\), as illustrated in figure 4, does not only represent the pure value, as the usual key-value matching DHT approach suggests. Additionally, essential procedure information of DhtFlex is annotated to a resource allowing flexible operations per instance.

We employ a simple type concept per resource, determined by the field \(type_{id}\). Thus, DhtFlex is able to distinguish mutable from immutable data resources. Once defined by an application, the type of an immutable data resource may not be changed to a mutable data resource as immutable data may be cached arbitrarily in the P2P network. However, this beehoves the application domain. The differentiation of the \(type_{id}\) provides the foundation to employ the flexible data operations of DhtFlex. The latest successfully written \(val_{id}\) is identified by an incremented \(seqNr_{id}\). Of course, an immutable data resource always preserves its initial value.

In order to avoid data loss tolerating node failures, a resource is replicated at different nodes determined by its replication factor \(replicaSize_{id}\). Thus, very flexible replication strategies can be defined individually per resource instance. All peers belonging to the replication group of the resource, thus being responsible for it, are captured in the set \(replicas_{id}\), as well as the special peer \(replicaMaster_{id}\) enforcing operation serialization for mutable data. Relying on a structured P2P overlay, the peer responsible for the administration of the key \(id\) in the overlay should be its \(replicaMaster_{id}\). This is optimal for the overall system performance, since all requests for this \(id\) are being routed to it anyway. DhtFlex uses this knowledge when maintaining a resource’s replication group configuration. As replicas may fail, new nodes need to be canvassed to buoy the replication factor. For the safety of the algorithm treating mutable data it is crucial to keep the \(replicas_{id}\) set consistent. Therefore, each \(replicas_{id}\) set is uniquely identified by a certain \(replicasNr_{id}\) incremented in a total-order style for each new composition. DhtFlex uses more lax procedures for immutable data resources.

**B. Recast Case**

**Algorithm 1** Trigger Recast Operation for peer \(p_i\)

**Require:** \(id\) data resource for \(id\)

1: \(\text{procedure checkReplicas}(id)\)  
2: \(\text{nextReplicas} \leftarrow \text{replicas}(id, \text{replicaSize}_{id})\)  
3: \(\text{if} \ \text{replicas}_{id} \neq \text{nextReplicas} \text{then}\)  
4: \(\text{if} \ \text{type}_{id} = \text{immutable} \text{then}\)  
5: \(\forall \text{peers} \in \text{nextReplicas} \\text{do}\)  
6: \(\text{send} \{\text{PUT}, \text{id}, \text{nextReplicas}, \text{value}_{id}\}\)  
7: \(\text{else if} \ \text{type}_{id} = \text{mutable} \text{then}\)  
8: \(\text{send} \{\text{RECAST-REQ}, \text{id}, \text{replicas}_{id}, \text{replicas}_{id}, \text{nextReplicas}\} \text{to} \text{nextReplicas}.\text{replicaMaster}\)

In order to preserve the replication factor \(replicaSize_{id}\) for a certain locally stored data resource, each replica periodically invokes procedure \(\text{checkReplicas}()\) (cf. algorithm 1) to check its replication group’s relevance to the current situation in the P2P overlay. This is necessary in potentially high churn environments, as peers may join, leave, or crash at arbitrary time. DhtFlex adopts the replication group of a resource to cover those peers being closest to its \(id\) in the P2P overlay (line 2). The force of a recast operation is simple for an immutable resource, e.g., only a PUT operation is required to transfer a

<table>
<thead>
<tr>
<th>(id)</th>
<th>The key under which the data resource is published and uniquely identified in the P2P overlay network, initially (0). E.g., the UUID of an item regarding our repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>(type_{id})</td>
<td>Flag variable to determine the type and the currently allowed operations on the data resource. Allowed values are immutable to represent read-only resources, mutable to represent modifiable resources allowing atomic put and get operations, or (FREEZE) to indicate an ongoing recast process</td>
</tr>
<tr>
<td>(seqNr_{id})</td>
<td>Counter variable (\in \mathbb{N}_0) to mark the sequence number for the latest successfully executed put operation, initially 0. In this context, the term successfully stands for &quot;accepted by a quorum of replicas&quot;</td>
</tr>
<tr>
<td>(seqNr_{id})</td>
<td>Counter variable (\in \mathbb{N}<em>0) marking the sequence number of the latest atomic put, i.e., modify effort of a certain (replicaMaster</em>{id}), initially (0). Only relevant for a mutable data resource</td>
</tr>
<tr>
<td>(replicaSize_{id})</td>
<td>Determines the replication factor (\in \mathbb{N}_0) of the resource, i.e., the size of its replication group, initially 4</td>
</tr>
<tr>
<td>(replicasNr_{id})</td>
<td>Counter variable (\in \mathbb{N}_0) to label the current replication group of the data resource, initially (0). Hence, it marks the “epoch” of a replication group configuration</td>
</tr>
<tr>
<td>(replicaMaster_{id})</td>
<td>ID of the current responsible peer of the data resource, i.e., the master of (replica_{id}), e.g., the immediate successor of (id) on a Chord ring. Usually this is a link to the first member of (replicas_{id}). Initially (\bot)</td>
</tr>
<tr>
<td>(replicas_{id})</td>
<td>Ordered set of peers forming the current replication group of the data resource – always corresponding to a certain (replicasNr_{id}), initially (\emptyset). Per convenience, the first peer is the (replicaMaster_{id}). The size should conform to (replicaSize_{id})</td>
</tr>
<tr>
<td>(replicasNr_{id})</td>
<td>Backup counter variable (\in \mathbb{N}_0) to identify a replication group that needs to be recasted; only relevant for the potentially new master of the new replication group of a mutable resource. As an illustration, this variable corresponds to a certain execution of Paxos</td>
</tr>
<tr>
<td>(replicas_{id})</td>
<td>Set of peers corresponding to a certain (replicasNr_{id}) forming a replication group configuration for a mutable resource; only relevant for the potentially &quot;new master&quot;</td>
</tr>
<tr>
<td>(recastNr_{id})</td>
<td>A sequence number (\in \mathbb{N}_0) to mark the proposal of an ongoing recast operation. This would correspond to a &quot;proposal number&quot; within Paxos and therefore has to be unique per execution effort. The Paxos algorithm requires this guarantee of uniqueness to work correctly. Only relevant for a mutable data resource</td>
</tr>
<tr>
<td>(recastNr_{id})</td>
<td>Set of peers building a potentially new replication group configuration in a recast process for mutable resources. Corresponds to a certain (recastNr_{id}) representing the &quot;proposal value&quot; within Paxos. Only relevant for a mutable data resource</td>
</tr>
<tr>
<td>(recastReplicas_{id})</td>
<td>Represents the latest accepted &quot;proposal number&quot; in the &quot;write phase&quot; of Paxos, i.e., the highest received (recastNr_{id}) of a RECAST-PROPOSE message. Only relevant for a mutable data resource</td>
</tr>
</tbody>
</table>

Fig. 4. Essential Fields of an Annotated Data Resource for a Certain ID per Peer State
copy to new members of its replication group (line 4). Hence, a peer no longer needed as replica for an immutable resource may simply delete its copy. For a mutable data resource it is essential for DhtFlex to assure consistent replication group configurations. For such resource types (line 7), it builds on the already addressed Paxos distributed consensus protocol to consistently determine its next replication group composition. To trigger a recast process, a current replica informs the master of a potentially new replication group configuration to initiate it (line 8). Peer nextReplicas.replicaMaster is typically the peer directly responsible for the id of the affected resource in the overlay network. The recast process for the replication group of a mutable data resource is typified in algorithm (line 2). A suspected master of a new replication group configuration gets informed with all necessary information about current (old) and new replicas by receiving a RECAST-REQ message (line 1). In short, the potentially master backs these information up and initiates the recast with a unique “proposal number”, recastNrId. Hence, the Paxos inspired procedure would be applied on the old replication group, but only if its “epoch” is still valid (line 2). The flag variable recasting marks an ongoing recast process to avoid unnecessary parallel execution, preventing operating resource waste. It is typically reset after expiration of some timer, so that a master can retry with a higher “proposal number” to support liveness. The aim of the whole recast process is that the current (old) replicas agree on a new composition and at least the latest consistently, i.e., to a quorum of replicas written value for the resource is preserved. For instance, a quorum might be represented by a simple majority, as in our case.

A RECAST message launches the “read phase” of Paxos, a RECAST-PROPOSE message the “write phase” and a RECAST-RES message the “commit phase”. In the “read phase” a master wants to get informed about a previously accepted replication group set.

A replica receiving a RECAST message (line 12) checks if the request actually targets the current replication group configuration (line 13) and does not conflict with concurrent recast efforts (line 14); if valid, the replica sets the typeid of the resource to freezed preventing put and get operations for it, till the recast process is finished (line 15). It accepts the request (line 16) and informs the master sending a RECAST-ACK message about previously proposed replication group configuration, if available (line 17).

For reasons of simplicity, algorithm 2 omits “NACK messages” being sent by a replica if a master tries to force a recast with an old replicasNr or a recastNr less than recastNrId or recastNrβ. If a NACK would be received by a “wannabe master”, it is assumed to stop executing the recast procedure, or may try a higher recastNr after a certain time.

A master receiving a RECAST-ACK checks if the message addresses the right replicasNrβ (line 19) and recastNrId (line 20). If a master is able to establish a quorum of positive feedback (line 21) it continuous and is allowed to enter the “write phase”. Here, the master selects a consistent replication group configuration (line 22) and tries to promote it as the next

**Algorithm 2** Recast Protocol for Mutable Data of Peer p_i

```
1: task wait until receive [RECAST-REQ, id, replicasNr, replicas, nextReplicas] from peer p_i { p_i may be p_j }
2: if replicasNrα < replicasNr then
3: if recasting = false then
4: typeid = freezed
5: recasting = true
6: replicasNrα = replicasNr
7: replicasIdα = replicas
8: recastNrId = recastNrId + replicasSizeId
9: replicasIdβ = nextReplicas
10: ∀ peers ∈ replicas do
11: send [RECAST, id, replicasNr, recastNrId] from p_i
12: send [RECAST, id, replicasNr, recastNrId] from p_i
13: if replicasNrβ = replicasNr then
14: if recastNrβ < recastNr & recastNrβ < recastNr then
15: typeid = freezed
16: recastNrId = recastNr
17: send [RECAST-ACK, id, replicasNr, recastNr, recastReplicas, α, β, recastNrId] to p_i
18: task wait until receive [RECAST-ACK, id, replicasNr, recastNr, recastReplicas] from p_i
19: if recastNrβ = recastNr then
20: if recastNrβ = recastNr then
21: if received [RECAST-ACK, id, replicasNr, recastNrId, replicasIdα, seqNrId] then
22: select [RECAST-ACK, id, replicasNr, recastNrId, recastReplicas] with highest recastNrβ
23: if recastReplicas ≠ 2 then
24: replicasIdβ = recastReplicas
25: ∀ peers ∈ replicasIdβ do
26: send [RECAST-PROPOSE, id, replicasNr, recastNr, seqNrId] to p_i
27: task wait until receive [RECAST-PROPOSE, id, replicasNr, recastNr, seqNrId] from p_i
28: if replicasIdβ = recastNr then
29: if recastNrId ≤ recastNr & recastNrβ < recastNr then
30: typeid = freezed
31: recastNrId = recastNr
32: recastReplicasId = recastReplicas
33: if seqNrId < seqNr then
34: send [RECAST-PROPOSE-ACK, id, replicasNr, recastNr, seqNrId, replicasIdβ, seqNrId] to p_i
35: else
36: send [RECAST-PROPOSE-ACK, id, replicasNr, seqNrId, valueId] to p_i
37: task wait until receive [RECAST-PROPOSE-ACK, id, replicasNr, seqNrId, valueId] from p_i
38: if replicasIdβ = recastNr then
39: if recastNrId = recastNr then
40: if seqNrId < seqNr then
41: seqNrId = seqNr
42: valueId = value
43: if received [RECAST-PROPOSE-ACK, id, replicasNr, recastNrId, replicasIdβ, seqNrId, valueId] from p_i
44: ∀ peers ∈ replicasIdβ ∪ replicasIdβ do
45: send [RECAST-RES, id, replicasNr, seqNrId, replicasIdβ, seqNrId, valueId] to p_i
46: task wait until receive [RECAST-RES, id, replicasNr, seqNrId, valueId] from peer p_i
47: if replicasNrβ < recastNr then
48: if replicasIdβ = recastNr then
49: seqNrId = seqNr
50: valueId = value
51: replicasIdβ = recastNr
52: replicasIdβ = recastNr
53: replicasIdβ = recastNr
54: replicasIdβ = recastNr
55: replicasSizeId = [replicas]
56: typeid = suitable
57: else delete data resource for id
```

381
replication group. By sending a RECAST-PROPOSE message it also requests at least the latest consistently written resource value and its corresponding seqNr in order to preserve it for the next replication group composition.

If a RECAST-PROPOSE request is accepted by a replica it compares the passed seqNr with its stored one. To save bandwidth, a replica only sends a real value within a RECAST-PROPOSE-ACK if it is relevant, i.e., it has a later value locally stored than the master (line 33).

If the master receives enough RECAST-PROPOSE-ACK messages to achieve a quorum within the “write phase” (line 43) it selects the latest consistently written value (line 40) and may propagate the results, so that the new replication group can take effect (line 56). Therefore, it increments the old replicasNr to label the new “epoch” (line 45). Here, the adoption of the Paxos principle guarantees that the new replication group is always chosen consistently reaching total-ordering.

C. Put Case

\[
\text{max}(v_1, v_2) \quad \text{return the maximum of two values } v_1, v_2
\]
\[
\delta(v) \quad \text{Computes the digest of a value } v
\]

Fig. 5. Auxiliary Functions of a Peer \( p_i \)

The basic protocol to process a DHT put operation in DhtFlex is given in algorithm 3 differentiating the case if a data resource is already locally stored, or not; associated auxiliary functions are elucidated in figure 5. A “writer” \( p_j \) passes with a PUT-REQ message its value and an optional policy to the responsible peer in the P2P overlay, calculated by \( \text{replicaSet}(id, 1) \). It must be pointed out that no actual value is transferred during such pure overlay lookup procedure, reducing bandwidth. The policy defines at least the writing peer \( \text{policy.writer} \), the resource’s replication factor \( \text{policy.replicaSize} \), and its type \( \text{policy.type} \). The latter ones might be specified statically or dynamically at runtime; if no policy is given default values are supposed, e.g., the resource being of type “mutable”.

The case is rather simple for an immutable data resource of a certain \( id \). For a not locally stored resource, the assumed master calculates its replication group (line 2) and distributes copies sending along relevant replication information (line 8). Each replica simply stores an immutable resource (line 21). Once done, the “writer” peer \( p_j \) gets informed about which value has successfully been published by sending back its digest.

However, the initial put of a mutable data resource needs special treatment to satisfy consistency. The potentially master checks its right regarding the current overlay situation (line 11); then, the data resource “blank” is “frozen” and pre-initialized with an initial seqNr and the given value. Subsequently, the potentially master initiates a recast process to consistently set up the resource’s replication group and pass the stored value to it (line 15). If the recast has been successful, either the value was adopted and the “writer” can be informed (line 20), or the put is retried (line 18).

![Algorithm 3 Put Protocol of Peer \( p_i \)](image)

For a “regular” put operation affecting a mutable data resource (line 25), the peer in charge rechecks if it is the master of the corresponding replication group. If not, the put request might be forwarded by inspecting \( \text{replicaMaster}_id \). A master tries to propagate the new value for the next valid increased sequence number (line 28) to the remaining replicas. All put requests are serialized over the master. A peer receiving a PUT message (line 32) verifies that the resource for the given \( id \) is mutable and the “epoch” of the master is still valid (line 34). For each \( \text{replicasNr}_id \) there exists a unique master. A replica only accepts the passed value if it is newer than its locally stored one (line 36) in order to forbid old updates. Here, “NACK” messages might be used to stop an “old master”, e.g., after having received a message carrying an old \( \text{replicasNr} \).
A master only updates its $value_{id}$ if a quorum of all replicas for a certain $seqNr$ (line 44) is acquired. Again, an update for an old value is prevented (line 45). Once a new value was successfully adopted the put is done and the responsible “writer” is informed (line 48).

D. Get Case

The behaviour of DhtFlex to deal with a DHT get operation is depicted in algorithm 4. A ‘reader’ propagates its get request GET-REQ to the assumed responsible peer given by $replicaSet(id, 1)$. If the data resource is an immutable one, the request can be immediately answered, as no consistency scrupulosity exists (line 2). In that case, each immutable data resource might be cached arbitrarily often at reader side for even more optimized get operations requiring no additional remote lookups.

For a mutable data resource (line 4) the target master verifies its authority (line 5) and sends a GET message to all replicas to investigate that the replication group it is responsible for is still valid, i.e., no missed recast has happened (line 7). A replica that receives such GET request needs to check if no recast process is active (line 9) and acknowledges the request if it is in the same replication group configuration “epoch” (line 10). Again, after receiving a GET message a replica might send a ”NACK” to stop an old master. Remarkably, no value transmission within a GET-ACK message from a replica to the master is necessary, as a valid master always knows the latest value. If the resource has not been “freezed” in the meantime, i.e., a recast process is going on (line 13), and the master achieves a quorum of positive responses (line 16) the “reader” can be served by sending a response (line 17).

```
Algorithm 4 Get Protocol of Peer $p_i$

Require: $\exists$ local data resource for id
1: task wait until receive (GET-REQ, id) from peer $p_j$
2: if $type_{id} = \text{immutable}$ then
3: send (GET-RES, id, value_{id}) to $p_j$
4: else if $type_{id} = \text{mutable}$ then
5: if $p_i = \text{replicaMaster}_{id}$ then
6: [peers $\in$ replicaSet do]
7: send (GET, id, replicasNr_{id}, $p_j$)
8: task wait until receive (GET, id, replicasNr, reader) from $p_j$
9: if $type_{id} = \text{mutable}$ then
10: if replicasNr_{id} = replicasNr then
11: send (GET-ACK, id, replicasNr, $p_j$) to $p_i$
12: task wait until receive (GET-ACK, id, replicasNr, reader) from $p_j$
13: if $type_{id} = \text{mutable}$ then
14: if replicasNr_{id} = replicasNr then
15: if $p_i = \text{replicaMaster}_{id}$ then
16: [if received (GET-ACK, id, replicasNr, reader) from $\frac{\text{replicasNr}_{id}+1}{2}$ peers then]
17: send (GET-RES, id, seqNr_{id}, value_{id}) to reader
```

VI. DISCUSSION

We belief that three thinks are crucial for the adoption of the DhtFlex approach: its offered safety, liveness and performance. As an entire evaluation is out of the scope of this paper we sentientiely discuss our approach highlighting the benefits of DhtFlex with regard to these crucial points.

The problem of ensuring safety and liveness for atomic operations occurs in the case where mutable data resources are considered. For such one, DhtFlex ensures the latest consistently written value via a put operation to be the one returned by a following get operation. DhtFlex implements atomic put and get operations by serializing all such requests through one master of a replication group achieving total-ordering. This is save, as a master is unique per replication group composition and only one composition is valid at certain point of time by relying on the used quorum concept. The adoption of Paxos for recast operations ensures total-ordering on the compositions of a resource’s replication group thus total-ordering of replication group masters and the preservation of at least the latest consistently written resource value, i.e., with the highest $seqNr_{id}$, for the succeeding configuration, respectively. Hence, safety holds in even high churn environments. The only exception applies for the case of an intermittent P2P overlay break-up, where initial put operations for the same id may be propagated in different physically separated segments. For all “normal”, i.e., not initial, put operations, even such partitioning of an already set up replication group can be tolerated guaranteeing safety. Therefore, DhtFlex might (temporarily) block put and get operations if the current valid replication group is affected in a way no quorum can be achieved, but continues once the partitioning is over.

DhtFlex supports a crash-recovery model. Therefore, it is able to tolerate the failure of arbitrary many peers within the currently valid replication group of a certain data resource and to ensure the progress of put and get operations, i.e. liveness. At least, as long as sufficiently many replicas eventually recover to form a quorum. The failure of a master is compensated by the execution of a recast operation. Regarding the progress of a recast operation DhtFlex may use some timer mechanism to force a master of a potential new replication group composition to retry its effort with a higher $recastNr_{id}$. Regarding the performance of DhtFlex we take a brief look at the internal communication overhead $c$ introduced by the support of atomic operations for replicated data. That is to say overhead additional to the necessary P2P overlay lookups and interactions with a client. Therefore, a single communication step is assessed, distinguishing a simple message transmission latency $t$ from such one where the transmission of a value is involved $t_v$, the latter being usually more expensive. The transmission latency variance of parallel send messages is neglected. The possibility to differentiate between mutable and immutable data resources allows DhtFlex to treat the latter ones very efficiently, i.e., for the communication costs of an “ordinary” DHT. A get request can be immediately responded by a replica ($c = 0$). A put requires forwarding the value to the remaining replicas ($c = t_v$); though this could be avoided if a put request is initially propagated to all replicas. The costs of a recast operation are comparable ($c = t_v$). DhtFlex for mutable data resources is optimized for get, put, and recast operations, in that order. Once a get request arrives at the master of a certain resource only two communication steps are necessary to verify its qualification ($c = 2t$). The exchanged messages...
are rather small, as the master holds the data and no value has to be transmitted. An “usual” put operation requires just as many steps, though the transmitted data is larger due to the need for replication \((c = t_u + t)\). An “initial” put usually requires as well as a recast five communication steps starting from a master, representing all phases of Paxos \((c = 3t + 2t_v)\). But even these “expensive” recast operations can be optimized by the assumption that it is not likely that concurrent initial puts or recasts occur. This is supported by the working of the algorithm as not every replica arbitrarily conducts such operations. All requests are always sent to the responsible peer appointed by the current P2P overlay situation. Hence, the circle of candidates is naturally kept small. The further optimized DhtFlex encourages a master to start with a RECAST-PROPOSE message, reducing the communication steps from five to three \((c = t + 2t_v)\). But, a replica may only acknowledge such an effort if it has not previously received a RECAST or RECAST-PROPOSE message from another peer for the same \(replicasN\). Otherwise, the master is informed about another concurrently operating master peer and reverts to the original, full recast process. Hence, in the worst case of the optimized variant seven communication steps are needed \((c = 5t + 2t_v)\).

VII. CONCLUSIONS AND FUTURE WORK

This paper presented DhtFlex, a flexible, fault-tolerant algorithm to support atomic data operations tailored for structured P2P overlays. DhtFlex is supposed to act as a generic building block offering very simple interfaces. It uses the concept of annotated resources to obey the peculiarities of P2P systems thus being able to distinguish immutable from mutable data. In this respect, DhtFlex is optimized for get, put and recast operations on mutable data resources. These operations are even more efficiently supported for immutable resources, increasing the overall performance in an employed system. Our approach is discussed regarding safety, liveness and performance. As a result, the communication costs of DhtFlex are comparable with that of non-atomic DHTs, in most of the cases.

Regarding future work, the evaluation of the different DhtFlex operations is currently under development to perform more sophisticated analyses. Especially, we plan to integrate a “King” data set [23] into our emulation environment. This approach allows us to gain estimations of DhtFlex’s communication costs based on real measurement data of thousands of Internet hosts. Hence, the overhead introduced by atomic operations on mutable data resources will be evaluated from a more real-life point of view.

REFERENCES