Abstract—An essential issue in peer-to-peer data management is to keep data highly available all the time. A common idea is to replicate data hoping that at least one replica is available when needed. However, due to churns, the number of created replicas could be not sufficient for guaranteeing the intended data availability. If the number of replicas is computed according to the lowest expected peer availability (a classical case), but the expectation were too high, then the peer availability after a churn could be too low, and the system could not be able to recover the requested data availability.

The paper is a continuation of previous work [1] and presents an replication protocol that delivers a configured data availability guarantee, and is resistant to, or recovers fast from churns. The protocol is based on a Distributed Hash Table (DHT), measurement of peer online probability in the system, and adjustment of the number of replicas accordingly. The evaluation shows that we are able to maintain or recover the requested data availability during or shortly after stronger or weaker churns, and at the same time the storage overhead is close to the theoretical minimum.

I. INTRODUCTION

Distributed Hash Tables provide an efficient way for storing and finding data within a peer-to-peer network, but they do not offer any guarantees about data availability. When a peer goes offline, the locally stored data become inaccessible as well. A standard way for achieving the requested data availability is to predict the system behavior in advanced, and to determine the sufficient number of replicas according to the minimal peer online probability. However, most of the time the system behaves much better, and therefore the generated storage costs (the number of replicas) will be higher than needed. On the other hand, if the peer availability decreases below the anticipated value, the requested data availability will be no longer guaranteed.

It would be much better if peers could measure the peer availability during the run-time, and adjust the number of replicas accordingly. By doing so, they would be able to detect churn periods, react and recover the requested data availability. Such a self-adapting replication approach could handle unpredictable system states, optimize the storage overhead, and would be independent of the initial settings.

The work presented in this paper extends existing Distributed Hash Tables (DHTs) by delivering a configurable data availability. The protocol is fully decentralized and has the following features:

- measuring current average peer online probability in the community with a high precision even during churns
- computing the number of replicas that every stored object needs to have in order to achieve the requested average data availability locally
- adjusting the number of needed replicas for locally stored items so that the requested data availability is achieved and kept

The protocol is fully decentralized, and is realized as a wrapper around an existing DHT implementation, exposing the common API [2] to applications. The wrapper communicates with the underlying DHT via the common DHT API as well. Therefore, high data availability feature can be added to any DHT that exposes the common DHT API.

The protocol has been evaluated at first in peer-to-peer networks without churns [1]. The results show that DHTs are able to measure the average peer availability with a high precision, and to adjust the number of replicas accordingly, no matter how the DHT was pre-configured. Some preliminary tests have shown that the protocol does recover the requested data availability after a churn.

The recovery time has been quiet long, but based on the obtained results and lessons learned, we have seen enough space for making the recovery shorter or even almost none existing. The main contribution of this paper is an improved version of the method for measuring the average peer availability. Churns are detected much faster, and the measurement error during a churn is minimized. The improved version of the replication protocol has been evaluated in DHT networks under stronger or weaker churns, and in networks under "negative" churn (i.e. a situation when the average peer availability increases). Both the obtained data availability and generated costs have been observed.

The paper is organized in the following way. The next Section gives a brief overview on common DHT API and properties. Section III gives details about applied replication and how it adapts on changes in the community. The approach is evaluated by using custom-made simulator and various results are presented in Section IV. Related work to the
II. DISTRIBUTED HASH TABLES

As suggested in [2], a common DHT API should contain at least the following methods:

- **route(Key, Message)** - routes deterministically a message according to the given key to the final destination. The operation is the basis for the store and lookup operations.
- **store(Key, Value)** - stores a value under the given key in DHT.
- **lookup(Key)** - returns the value associated with the key.

Both synchronous store and lookup operation are realized using asynchronous route method. A lookup is initiated by routing a LookupRequest message with the given key. The peer that receives it responds with a LookupResponse message. Similarly, a store operation introduces StoreRequest and StoreResponse messages.

Every peer is responsible for a portion of the key space, so whenever a peer issues a store or lookup request, it will end up on the peer responsible for that key. Although implementation specific, a peer is responsible for a part of the keyspace nearest to its ID. The distance function can be defined in many ways, but usually it is simple arithmetic distance. When the system topology is changed, i.e. a peer goes offline, some other online peers will be now responsible for the keyspace that has belonged to the offline peer. The part of the keyspace belonging to the currently offline peer will be split and taken over by peers that are currently nearest to keys in the part of the keyspace. Also, peers joining the system will take responsibility for a part of the keyspace that has been under control of other peers until that moment.

All DHT implementations do not guarantee data availability, i.e. when a peer is offline, then locally stored data are offline too. Therefore, enabling high data availability can be achieved by wrapping a DHT and adding the implementation of a replication protocol on top of it, and at the same time being transparent to the users, i.e. providing the same API to upper layers as a DHT does.

III. APPROACH

A. Decentralized Directory

Every replication protocol needs to have access to a directory that maintains information about replicas and their location. Usually, in a distributed environment, the directory is managed on a central, known, and highly available location. Unfortunately, peer-to-peer communities in general lack such high available peers that could play a role of such a directory manager. Therefore, before implementing any replication protocol, a solution for providing directory operations must be found.

A decentralized directory could be implemented on the top of an existing DHT. Thanks to the deterministic DHT routing, a key used in store or lookup is treated as a replica location, whereas a value is the replica itself. To avoid collisions, all replicas of all managed objects in a DHT must be stored using different, unique key, and every peer must be in a position to generate locally the list of all replica’s keys of the given object.

Therefore, the first replica key is equal to the object ID and is generated using a random number generator. All other replica keys are correlated with the first one, i.e. they are derived from it by using the following rule:

\[
\text{replicaKey} = \begin{cases} 
\text{hash(replicaKey(1) + ron)} & \text{for } ron = 1 \\ 2^{\text{hash(replicaKey(1) + ron)}} & \text{for } ron \geq 2 
\end{cases}
\]

where \( ron \) is a replica ordinary number, \( c \) is a random byte array, \( \text{hash} \) is a hash function with a low collision probability. \( \text{replicaKey} \) and \( i \) are observed as byte arrays, and \(+\) is an array concatenation function.

The above key generation schema ensures that all replicas are stored under different keys in the DHT. Further, two consecutive replica keys are usually very distant in the key space (a consequence of using a hash function) meaning that replicas should be stored on different peers in a fairly large network. Finally, it is sufficient to know only the first replica key for a given object; the remaining replica keys can be computed. Deriving the key of the \( ron^{\text{th}} \) replica requires access to the first replica key and the replica ordinary number \( (ron) \). Therefore, this information must be attached to the stored replica itself. To summarize, an entry containing a replica will be wrapped in an instance of the following class:

```java
class Entry {
    Key first;
    int ron;
    Object value;
}
```

B. Modified DHT Operations

Since the wrapper around DHT implements the common DHT API, store and lookup operations must be re-implemented.

**store(Key, Value)** When a value is created, it is wrapped in a number of instances of Entry class and appropriate keys are generated. The constructed objects are stored using the basic DHT store operation. In cases when a replica already exists on destination peer, its old value is overwritten.

**lookup(Key)** When a peer wants to get a value, it is sufficient to return any available replica. Therefore, the keys for the assumed number of replicas are generated and the basic DHT lookup operation is applied until the corresponding Entry object is retrieved.

When a peer rejoins the community, it does not change its ID, and corollary the peer will be now responsible for a part of the keyspace that intersects with the previously managed keyspace. Therefore, the peer keeps previously stored data, but no explicit data synchronization with other peers is required. Replicas, whose keys are not anymore in the part of the keyspace managed by the rejoined peer, can be removed and may be sent to peers that should manage them.

C. Number of Replicas

In order to add the data availability feature to an existing DHT, stored values must be replicated \( R \) number of times.
The number of replicas \( R \) is independent of objects, i.e. the same data availability is requested for all objects. Joining the community for the first time, a peer can assume an initial value for \( R \), or can obtain it from its neighbors.

The following analysis models the current data availability in the system, when objects are replicated \( R \) times. Let us denote with \( p_r \) the probability of a replica being online. Since both keys and peer IDs are generated randomly, we can assume that after a store operation, every replica will be placed on a different peer, i.e. \( p_r = p \), where \( p \) is the average peer availability.

Let us denote with \( Y \) a random variable that represents the number of replicas being online for a given object, and \( P(Y \geq y) \) the probability that at least \( y \) replicas are online. Under an assumption that peers are independent, the probability that \( y \) copies are online at any point in time can be given as:

\[
P(Y = y) = \binom{R}{y} p^y (1-p)^{R-y}
\]  

(2)

The probability \( a \) that a DHT value is available is equal to the probability that at least one replica is online \( (P(Y \geq 1))\):

\[
a = P(Y \geq 1) = 1 - P(Y = 0)
\]  

(3)

Therefore, the number of needed replicas \( R \) is

\[
R = \left\lceil \frac{\log(1-a)}{\log(1-p)} \right\rceil
\]

(4)

Determining the needed number of replicas \( R \) that delivers the requested average data availability \( a \) requires knowing average peer availability. Guaranteeing the requested data availability over time would require measuring average peer availability, and adjusting the number of replicas of locally stored objects accordingly.

Every peer should measure at least once per online session the current average peer online probability, and knowing the requested data availability, calculates the new value for the number of replicas \( R \). By knowing the previous value \( R' \), a peer removes the replicas with ordinary number \( r \) greater than \( R \) from its local storage replicas, if it turns out that less replicas are needed than before \( (R < R') \). In case a higher number of replicas is needed \( (R > R') \), a peer creates new replicas of the data in its local storage under the keys \( replicaKey(j) \), \( j = R'+1, \ldots, R \).

If more replicas of a particular object are available at some point in time, peers that have them could decide to create \( R-R' \) new replicas. Thanks to the DHT properties and our key generation mechanism (Formula 1), they all generate replicas with only \( R-R' \) different keys. Such situation causes multiple store requests with the same key (i.e. higher traffic), but they all end up at the same peer, increasing the number of replicas for every different key.

### D. Measuring Average Peer Availability

Measuring the average peer availability is based on probing peers and determining the peer online probability as a ratio between the number of positive and total probes. However, the measuring cannot be done directly (i.e. pinging peers) because we do not know anything about the peer community, i.e. peer IDs, and/or their IP addresses. Our wrapper uses the underlying DHT via the common DHT API, i.e. its routing operation for probing the availability of replicas. According to the key generation properties (Formula 1) stated before, we assume that every replica of a particular object is stored on a different peer within a fairly large network. With such an assumption, the peer online probability can be measured indirectly by measuring replica online probability.

Be aware that this condition is only valid if each replica of an object is stored in a single online or offline peer. Having several replicas with the same key distributed over several online or offline peers results in peer online probability measurement errors, because the availability of a replica does no longer equal the availability of a single, but of several peers. Having only one replica under a key is supported by the implemented replication, and therefore the measurement of peer online probability can be realized as follows. Every peer has some replicas in its local store and based on that and Formula 1, all keys of all other replicas corresponding to the same objects can be generated, and their availability can be checked.

#### 1) Probing with Requested Confidence:

Of course, we are not in a position to check all replicas stored in the system. In order to optimize the number of probes, we consider the confidence interval theory [3] to find out what is the minimal number of replicas that has to be checked, so the computed average replica online probability is accurate with some degree of confidence.

We check \( n \) replicas and compute average online probability \( \hat{p}_e \) from:

\[
\hat{p}_e = \frac{1}{n} (X_1 + \cdots + X_n)
\]

(5)

where \( X_i = 0 \) when a replica is offline, or \( X_i = 1 \) for being online.

Then the expectation value and variance are:

\[
E(\hat{p}_e) = p_e \quad \text{and} \quad \text{Var}(\hat{p}_e) = \frac{p_e(1-p_e)}{n} \leq \frac{1}{4n}
\]

If the community is very large, then we can approximate the distribution of \( \hat{p}_e(X) \) by Normal distribution

\[
N\left(\hat{p}_e, \frac{p_e(1-p_e)}{n}\right)
\]

Then we have approximately

\[
\frac{\hat{p}_e - p_e}{\sqrt{\frac{p_e(1-p_e)}{n}}} \sim N(0, 1)
\]
For a medium and small-size network, variance of $\hat{p}_e$ depends on the network size $N$

$$\text{Var}(\hat{p}_e) = \frac{(N - n) p_e (1 - p_e)}{n}$$

Based on this approximation, we are able to calculate the number of probes $n$ we need to make, in order that the computed average probability $p_e$ falls into the interval $(\hat{p}_e - \delta, \hat{p}_e + \delta)$ with a given probability $C$

$$P(\hat{p}_e - \delta \leq p_e \leq \hat{p}_e + \delta) \geq C$$

Be aware that the number of probes $n$ is independent of the network size for very large networks, but it depends on the size of the interval $2\delta$ and the probability $C$.

Unfortunately, the direct usage of the confidence interval theory is not so promising for practical deployment. Namely, by requesting a small absolute error ($\delta = 0.03$), and high probability of having the measurement in the given interval ($C = 95\%$), a peer should make at least 1068 random replica availability probes, generating high communication costs. Even if the network is not so large (e.g. 400 peers), the number of probes $n$ drops only to 290.

Therefore, we need to modify slightly our probing strategy to achieve the requested precision. After probing some replicas using their keys, and calculating the peer availability, the peer uses the same replica keys to ask other peers about measured values that they are aware of. Finally, the peer computes the average peer online availability by averaging the received values.

Getting the measurements done by other peers requires introducing two additional DHT message types. A peer that needs to find out what peer availability has been measured by another peer sends a PeerAvailabilityRequest message using the route DHT method with a replica keys from the probe set, and receives the answer via PeerAvailabilityResponse messages.

Computing the average peer online availability generates $n$ PeerAvailabilityRequest and $n$ PeerAvailabilityResponse messages, but a peer gets back measurements that has been able to check up to $n(n + 1)$ replicas. The peer itself probes $n$ random replicas, and receives as well measured values from up to $n$ peers that have been able to probe $n$ random replicas as well. Such a two-step approach produces low communication costs: already with $n = 33$, and 66 messages we are able to check up to 1112 replicas, and achieve good precision in a network of any size. This is a significant gain, because the initial idea requires 1068 messages for obtaining the same precision.

2) Better Accuracy under Churn: The proposed measurement technique produces accurate results all the time, if a DHT is quiet stable, i.e. the average peer availability is constant or changes slowly over time. In reality, the assumption cannot hold; the average peer availability varies significantly during the run-time, and it can depend on many parameters, like user behavior, time of the day, or content popularity.

Since the collected measurements are just averaged, the computed value is precise only if all measurements have been made since the system has entered in a stable state. During and some time after churns, peers measure much higher peer availability than the actual one, and do not create enough replicas immediately. However, as time passes, measurements gets more precise, and peers create the number of replicas needed for restoring the requested data availability.

Testing the protocol presented in [1] under churns proves the above statement: it is able to restore the given data availability after the system stabilizes again, but the recovery period is pretty long.

Our goal is to make the protocol more reactive, i.e. recovery should be quick, even nonexistent under a weak churn. Measured average peer availability should be as precise as possible, even during churns. As one can see, just averaging received values is not a good way to achieve that, because it does not explore the information about the time when a measurement is made. Recent values should be more precise, but older values are also important for recognizing churn trends.

Therefore, every measurement is accompanied with a time stamp. Since peers do not have synchronized clock and communities are usually spread across multiple time-zones. The time stamps that are traveling with the measurements must be converted from the absolute time stamp into a relative one. A relative time stamp describes how many time units before the current moment an event has happened.

Received values are not averaged anymore; instead every peer places them in a history. Peers exchange gathered values via within PeerAvailabilityResponse message, and received values are used for building more complete histories.

Finally, after receiving all $n$ PeerAvailabilityResponse messages, the peer’s history contains a number of measurements done in the past. A simple way of estimating the current peer availability is to find a curve that fits to the measured data with the lowest error. Such technique is called regression analysis and its simplest form is known as linear regression analysis, where a linear curve $y = a + bx$ is fitted to the data. In our approach $y$ is the measured peer availability $p$, and $x$ is time $t$. Therefore, our aim is to determine the curve $p(t) = a + bt$, i.e. its coefficients $a$ and $b$. Having the curve, the estimation of the current average peer availability is the value of $p(0)$.

There are plenty of techniques for computing the values of coefficients $a$ and $b$, and one of them is Least Squares method, which attempts to minimize the sum of the squares of the ordinate differences (called residuals) between points generated by the curve and corresponding points in the data.

Please notice that the methodology presented in this Section allows fitting to any another type of curve, or fitting can be done using another optimization techniques. Finding the optimal fitting strategy (i.e. a curve with the smallest error) is out of the scope of this paper.

The history size at every peer can keep only values of limited age, so that the computed curve can closely represent the current situation in the community. Otherwise, as the system run-time increases, the computed curve would become more and more insensitive to churns. The value $p(0)$ would
converge to the average peer availability in total, but peers would not be able to detect any increase or decrease of peer availability in longer periods of time and react properly.

The fitted curve is accurate only if data points have low variance. As demonstrated in Section III-D.1, the low variance can be obtained by averaging the received values. Averaging values with the same timestamp would be optimal, but chance that more measurements have been done at the very same moment is very low. Thus, in order to produce data points with lower variance, a tradeoff must be made. The history is divided into clusters of defined durations, and every cluster is represented with an average - an average of all values within the clusters. The major assumption here is that during the time period equal to the cluster duration, the average peer availability changes insignificantly.

To summarize, here is a list of steps needed for computing the current average peer availability:

1) Init: history has a length of $T_H$ time units, cluster length is equal to $T_c$ time units, where $T_c < T_H$
2) From the set of locally stored replicas, and knowing the last needed number of replicas $R$, a peer generates randomly $n$ replica keys using Formula 1
3) Peer issues $n$ PeerAvailabilityRequest messages
4) Peer receives $n$ PeerAvailabilityResponse messages, that correspond up to $n$ different peers; received measurements are placed in the local history
5) Peer iterates over the history, groups measurements into clusters, calculates the average values per cluster, and uses them in linear regression analysis
6) The computed average peer availability is equal to $p(t)$, where $p(t) = a + bt$

E. Costs

In general, the total costs consists of two major parts: communication and storage costs. The presented approach is self-adaptable; it tries to reach and keep the requested data availability with minimum costs at any point in time. If the initial number of replicas is not sufficient to ensure the requested data availability, new replicas will be created, i.e. storage costs will be increased. On the other hand, if there are more replicas than needed, peers will remove some of them, reducing storage costs.

Let us denote with $S(t)$ average storage costs per peer in a system with $M$ objects and $N$ peers that are online with probability $p$ at a point in time $t$. The minimum storage costs per peer $S_{min}$ that guarantee the requested data availability $a$ is:

$$S_{min} = \frac{M}{N} R$$

(6)

where $R$ is computed according to in Formula 4.

In addition to the storage load, every store operation generates $R$ StoreRequest and $R$ StoreResponse messages. A lookup operation does not have fixed costs, it stops sending LookupRequest messages when a replica is found. Sometimes, already $1^{st}$ replica is available, and only two messages (LookupRequest and LookupResponse) are generated. In an extreme case, $R$ request and response messages must be generated in order to figure out if the requested object is available or not. It can be shown that on average a lookup needs to generate $R$ messages, before finding a replica of the requested object.

The proposed approach introduces additional communication costs generated by measuring peer online probability $p$ and adjusting the number of replicas. Every measurement generates $n$ PeerAvailabilityRequest, and $n$ PeerAvailabilityResponse messages, where $n$ is the number of replicas to probe. As stated in Section III-D, $n$ depends on the absolute error $\delta$ we want to allow, and the probability that the measured value is within the given interval $(p - \delta, p + \delta)$. If the computed number of replicas $R$ is greater than the previous one $R'$, the peer will create additionally $(R - R')S(t)$ StoreRequest messages on average.

IV. Evaluation

The evaluation has been done using a custom-made simulator. The protocol implementation is based on FreePastry [4], an open-source DHT Pastry [5] implementation. The obtained results match the expectation, i.e. DHTs are able to maintain or recover the requested data availability during or shortly after churn.

A. Simulator

Every simulation consists of the following steps: (1) DHT creation, (2) Populating the DHT with a number of entries (initially replicated a number of times), and (3) Running a defined scenario

The simulator is aware of time, but it does not use a real time source. Instead, time is calculated in abstract time units that could be later extrapolated to the real time scale.

Every scenario last a number of time units, and during each of them, the following actions are performed:

- Changing DHT topology
  At the beginning of a time unit, some peers, whose session time is over, go offline, and some other peers with the probability $p$ come online and stay so for some time. Session length is generated by using Poisson distribution, with the given average session length $\lambda$. In order to simulate dynamic (churns), both the average peer online probability $p$ and the session length $\lambda$ can increase or decrease linearly. The duration of such a process, its start and stop value, or triggering mechanism can be configured separately.

Each time when a peer comes online, it measures the average peer availability, determines the number of required replicas, and adjusts the number of replicas for locally stored data according to the mechanism defined in Section III-D.

- Measuring actual data availability
  At the end of every time unit, the simulator measures the actual data availability by trying to access all stored
entries. It iterates over the entry keys, picks up a random online peer, which then issues a \texttt{lookup} request.

### B. Settings

The main configuration parameters of the simulator are the community size $N$, the requested data availability $a$, the average peer online probability $p$, the average session time $\lambda$, the initial number of replicas $R$, and the number of replicas to probe $n$.

These parameters offer a huge number of possible scenarios to simulate, and since the simulations are time-consuming, only the most relevant ones have been chosen. Delivering a high data availability is only relevant in real-world applications, and thus the simulations take into account only the requested data availability of 99%.

The community size $N$ has been fixed to 400 peers, the average session time $\lambda$ to three time units, and the number of replicas to probe $n$ to 30. As stated in Section III-D, for the network of a such size, and the measurement precision of $\pm 0.03$, $n = 17$ would be enough, but we wanted to be on the safe side at the first glance.

A major assumption of the proposed protocol is that all replicas of an object are stored on different peers within the network. This assumption mainly influences the measurement of the peer online probability. If this assumption is not fulfilled the correlation of the peer online probability and the replica availability is corrupted influencing the correctness of the approach. As stated in Section III-D, this assumption holds in a fairly large network.

Getting accurate measurements requires that peers have enough replicas for probing. Thus, the system is populated with $M = 6000$ items, and if replication factor is greater than 1 ($R > 1$), every peer receives more that 30 replicas on average, and is able to generate good probing sets.

In order to test DHT’s behavior equipped with our protocol under a churn, all DHTs are pre-configured to deliver the requested data availability of 99% from the beginning of simulation. After 10 time units, the simulator forces a churn of a given duration. After the churn end, the system, i.e. its average peer availability, remains stable. Each of the following scenarios are executed 10 times, and the obtained values are averaged:

1) highly-available DHT (the average peer availability of 50%) is under a strong churn, i.e. a number of peers is going offline in a short period of 15 time units (average peer availability linearly decreases to 20%)  
2) the same highly-available DHT is under a weak churn, i.e. a number of peers is going offline in a long period of 100 time units  
3) low-available DHT (the average peer availability of 20%) is under a strong "negative" churn, i.e. a number of peers is coming online in a short period of 15 time units (average peer availability linearly increases to 50%)  
4) the same low-available DHT is under a weak "negative" churn, i.e. a number of peers is coming online in a long period of 100 time units

The first two scenarios check if the protocol is able to generate enough replicas and recover the initial data availability during or after the churn. The simulator logs the data availability at every time unit, and after all runs for the given scenario, average, minimum, and maximum value at every time unit is computed. The obtained results are evaluated through a graph showing the obtained cumulative absolute error between the requested and the achieved average data availability. An error value at the time unit $t$ represents the average absolute error from the beginning of simulation until the time unit $t$. When the data availability is recovered, the error should become low and remain on a such level.

The data availability in the last two scenarios does not change, because an increase of peer availability makes only more replicas available. However, so many replicas are not needed anymore for guaranteeing the same data availability, and they could be removed. The simulator tracks the storage load per peer at every time unit, and after all runs produces a curve that shows the average storage costs per peer in time. When the requested data availability is achieved, the storage costs should be close to the predicted one (Formula 6).

In addition to this, we observe generated communication costs. The simulator is able to track the total number of \texttt{PeerAvailabilityRequest} and \texttt{StoreRequest} messages per time unit, generated during peer availability measurements and replica adjustments. The measurement costs should be stable over time, whereas the number of \texttt{StoreRequest} messages should decrease when the requested data availability is reached.

### C. Data Availability Recovery

Figure 1 demonstrates that the applied replication protocol is able to recover the data availability, shortly after the strong churn is over. Three curves are displayed: the average, maximum, and minimum delivered data availability obtained in 10 runs. Markers "dynamic start" and "dynamic end" shows when the churn start, and stop respectively.

The error (Figure 1b) goes up during the churn phase, but already 35 time units after the churn end, the error drops below 0.025. Interestingly, the error peak happens out of the churn period (around 30 time units). The peer availability measurements rely upon previous measurements stored in local histories. Although peers try to detect the actual average peer availability, measured value are higher than the actual one, and as a consequence, a lower number of replicas will be created. Fortunately, the measurement error drops at the end, or shortly after the churn, and peers are able to create enough replicas for restoring the data availability.

After a successful recovery from the strong churn, it would be interesting to see what happens if the churn is weaker. The DHT has coped very good with it, and the requested data availability is practically maintained (Figure 2) throughout the simulation. The average absolute cumulative error (Figure 2b) reaches its maximum of 0.012 at the end of churn phase. Our protocol handles this kind of churn well thanks to the applied measurement method (Section III-D). During long chucks,
built regression curves have a very precise slope, and measured peer availability is close to the actual one. Thus, peers are able to react properly, and follow the churn.

D. Costs

As already discussed, the applied replication protocol introduces additional storage and communication costs. This Section presents the costs generated in the previously defined scenarios, and compare them with the projected costs as described in Section III-E.

Obtained results have been used for calculating the average storage cost (the "average" curve), the maximum obtained costs within 10 runs (the "maximum" curve), and the minimum generated costs within 10 runs (the "minimum" curve). The curve "theoretical minimum" corresponds to the costs predicted by the model (Formula 6) and it is used for estimating the actual storage overhead. The curves "dynamic start" and "dynamic end" mark the beginning and the end of the churn.

Figure 3 presents the generated costs in the DHT under the strong churn. As one can see, already during the churn, the DHT starts the data availability recovery by creating additional replicas (Figure 3a), and therefore increasing the average storage costs per peer until the data availability is recovered. After that, the storage costs remain close to the theoretical minimum. The DHT behaves the same (Figure 4a) under the weak churn.

Apart from the storage overhead, our replication protocol introduces additional communication costs that are related to measuring the average peer availability and optionally creating additional replicas to reach the requested data availability. As stated in Section III-E, the cost of measuring the average peer availability does not depend on the system properties like the actual peer availability, or DHT network size. It depends only on the measurement precision we want to achieve. As defined in Section IV-A, peers measure the average peer availability immediately after coming online. Therefore, the total number of PeerAvailabilityRequest messages per a time unit, used for calculating the average peer availability is related to the number of peers that measure peer availability per a time unit.
Figures 3b and 4b demonstrate this statement: the measurement costs (the "peeravailabilityreq" curve) are quite independent from the initial settings, and the obtained data availability; it depends only on peer arrival rate. The arrival rate is lower for the average peer availability of 20% compared to 50%, and therefore the number of PeerAvailabilityRequest messages follows the change of peer availability as well.

The number of StoreRequest messages is correlated to the average storage costs, since almost every store creates a new replica. When a churn occurs, the DHT needs to recover the maintained data availability, and therefore, the number of store messages goes up. If the churn is very strong (Figure 3b), the peak of the "storereq" curve happens when the churn is over, and the peer availability has already stabilized. This is because, the number of online peers has decreased drastically in a short period. During that period only a small number of peers could come online, and recognize a need for adding more replicas. Contrary, if the churn is slower (Figure 3b), many peers are able to follow the churn dynamic and react properly. Therefore, the number of StoreRequest messages gets its peak within the churn period.

The last two evaluated scenarios explore the possibility of having "negative churns" in a DHT, i.e. periods when the average peer availability increases. Such a situation does not affect the data availability guarantees, since the number of replicas were computed for a lower average peer availability. However, if the new peer availability remains stable over a longer period of time, there is no necessity to keep so many replicas in peer’s storages; the same data availability is achieved with less costs. Figure 6a and Figure 5a demonstrate the development of the storage costs in DHTs where the average peer availability increases over time. Both DHTs have been able to detect the new stable situation, and to reduce the number of replicas accordingly. At the same time, the number of measurement messages remains stable (Figure 6b and Figure 5b). From time to time, there are some "storereq" activities, but they are related to moving replicas from one peer to another, because the system topology changes, and some
peers are not responsible anymore for some locally stored replicas.

V. RELATED WORK

Current popular P2P file-sharing systems (e.g. KaZaA, eDonkey, or Gnutella [7]) do not have any built-in support for replication. A file will be replicated every time it is downloaded on a new peer. Therefore, the file availability depends on its popularity, and there is no mechanism for guaranteeing its availability. However, unlike in our approach, none of the peers knows how many times a particular file is replicated, and what is its availability, and consequently, they are not able to keep it on some predefined level.

P2P file-storing systems like CFS [8], PAST [9], or FARSITE [10] have recognized the need for replication. CFS splits every file in a number of blocks that are then replicated a fixed number of times. PAST applies the same protocol, without chunking files into blocks. The number of replicas must be carefully chosen for the given community, because systems are not able to correct it, if the requested availability cannot be obtained. Also, if the system behaves much better, local peer storages will contain unnecessary replicas. Compared to the settings where our protocol can operate, a major assumption of FARSITE replication protocol is that the majority of machines is up and accessible for the majority of the time. Even with such settings, the system is not able to detect how many replicas are really needed, and it replicates each file a decent number of time, occupying more storage than really required. Thus, the primary goal of the replication is to increase performances of FARSITE operations, without optimizing the storage costs, like our protocol does.

Similarly, aiming to achieve low access latencies, and good load balancing, the authors in [11] propose a replication protocol based on the top of Chord DHT implementation. Every peer is able to measure its own load (in terms of network bandwidth and CPU), and if it is above a given threshold, the peer creates more replicas somewhere else, reducing its load. Oceanstore [12] is a P2P file-storing system trying to
provide a wide-area storage system. Oceanstore is not fully decentralized, it makes a distinction between clients and powerful, highly available servers (i.e., super-peer network). Within the storage, the data are replicated by using erasure coding (Reed-Solomon codes). Erasure coding is also used in TotalRecall [13]. Erasure coding offers lower storage costs compared to replication, if managed data are large in size. However, our DHT layer handles smaller data items. Additionally, if locally stored data are encoded, then it is not possible to process them without retrieving all needed pieces from the network, which would decrease the system performances.

The protocol presented in [14] exploits as well the erasure coding for achieving data availability. Periodically, peers detect current file availability and if below the requested one, replicate randomly files attempting to increase their availability. The idea behind the protocol is similar to our approach, but requires existence of a global index and global directory, where all information about files (locations and availability) are managed. The protocol presented in our paper is fully decentralized, and does not need such data structure for proper functioning. The peer-to-peer file system Ivy [15] provides both read and write operations, and should manage data availability, but more details about applied replication mechanism are not given, nor simulation were performed with replication.

VI. CONCLUSION AND FUTURE WORK

The paper has presented a fully decentralized replication protocol suited for DHT networks. The evaluation has shown that DHTs are able to recover the requested data availability shortly after strong churns, whereas it is maintained during churns with low peer leaving rate. At the same time, the protocol keeps storage costs very close to the minimum. The clear advantage of a DHT network equipped with the presented protocol is deployment in an environment, whose parameters like peer availability, are not well-known or cannot be good predicted. The system itself will find the optimal number of replicas, and maintain the requested data availability. If the system properties change afterwards, the system will adapt automatically, i.e. no system restart and/or reconfiguration is needed.

Future work will evaluate data consistency issue. Since not all peers are always online, an update might not be able to change all replicas, leaving some of them unmodified. Also, uncoordinated concurrent updates of an object result in unpredictable values of object replicas. Thus, the main issues related to updates are (1) how to ensure that the correct value is read, (2) to synchronize offline replicas after going online again, and (3) to handle concurrent updates on the same data. The protocol presented in the paper has been already extended in order to support solving the above issues [16]. The analytical analysis shows that we are able to provide data consistency with arbitrary high probabilistic guarantees. Further research will evaluate the protocol under various system settings and against the above issues using the simulator and similar communities like in this paper.

REFERENCES