PIGGYBACK META-DATA PROPAGATION IN DISTRIBUTED HASH TABLES

Erik Buchmann, Sven Apel, Gunter Saake
University of Magdeburg, Germany

Keywords: Peer-to-Peer data structures, meta-data propagation, Mixin Layers, Aspect-oriented Programming.

Abstract: Distributed Hashtables (DHT) are intended to provide Internet-scale data management. By following the peer-to-peer paradigm, DHT consist of independent peers and operate without central coordinators. Consequentially, global knowledge is not available and any information have to be exchanged by local interactions between the peers. Beneath data management operations, a lot of meta-data have to be exchanged between the nodes, e.g., status updates, feedback for reputation management or application-specific information. Because of the large scale of the DHT, it would be expensive to disseminate meta-data by peculiar messages. In this article we investigate in a lazy dissemination protocol that piggybacks attachments to messages the peers send out anyhow. We present a software engineering approach based on mixin layers and aspect-oriented programming to cope with the extremely differing application-specific requirements. The applicability of our protocol is confirmed by means of experiments with a CAN implementation.

1 INTRODUCTION

Distributed hashtables (DHT) are able to manage huge sets of (key,value)-pairs and cope with very high numbers of parallel transactions. DHTs are a variant of distributed data structures which follow the peer-to-peer (P2P) paradigm. In particular, they consist of many autonomous nodes, there is no central coordinator and global knowledge is not available. Examples of DHTs are Content-Addressable Networks (CAN) (Ratnasamy et al., 2001a), Chord (Stoica et al., 2001), Pastry (Rowstron and Druschel, 2001) or P-Grid (Aberer, 2001). Characteristic DHT applications are distributed search engines, repositories for the semantic web or dictionaries for web services; cf. (Gribble et al., 2001) for an exhaustive list. However, DHTs work well only if some meta-information are present on proper peers, e.g., are adjacent nodes alive and which tuples are managed by them. Furthermore, some information belong to management or optimization issues of the DHT. For example, load balancing mechanisms may depend on data about the average load of all peers. Finally, application-specific knowledge have to be spread over the network, e.g., information about the synchronization of work in distributed web-crawlers. Most meta-data items have a small footprint, are not needed in time, old ones becomes obsolete, and the number of data items is really large. In addition, the importance of data items for a certain node depends from various factors, e.g., the workload of the node that issued it, or the distance (Kleinberg, 2000) to the addressee. Summing up, propagation by peculiar messages would lead to a very high network traffic. Therefore, related techniques known from distributed databases (epidemic protocols etc.) can not be applied.

In this article we present a generic protocol that piggybacks meta-data attachments to existing messages that the peers send out for data management purposes, i.e., we use a variant of *lazy dissemination* (Rodrigues and Pereira, 2004; Boyd et al., 2005). Based on the peer a message is forwarded to, each peer selects a limited number of data items it deems to be important for the receiving peer, and attaches them to the message. On the other hand, each peer decides to keep data items from incoming messages or not. If the number of items exceeds the limit, a replacement algorithm removes the most dispensable data items. The broad spectrum of potential application scenarios leads to a wide range of requirements for meta-data dissemination protocols, e.g., small footprint, selective forwarding or time-constraints (cf. Sec. 2). Different strategies for item caching and dissemination as well as their combinations lead to different results regarding the behavior of DHTs (cf. Sec. 5). Furthermore, we have observed that strategies sometimes differ only in some details, e.g., a threshold or some additional constraints. Thus, our implementation is based on modern software engineering methods, namely *Aspect-oriented Programming (AOP)* (Kiczales et al., 1997) and *mixin layers* (Smaragdakis and Batory, 2002). Furthermore, we discuss their advantages and disadvantages in terms of reusability, configurability and extensibility especially for the domain of DHTs. We evaluate our protocol on top of a CAN implementation by experiments. The experiments exemplarily compare different dissemination strategies. Our results verify that the software engineering approach allows to build variants of the protocol which meet the requirements of many different applications.

The remainder of this paper has the following structure: After investigating in the characteristics of meta-data in Section 2, we describe our piggyback meta-data propagation protocol in Section 3. Section 4 presents our implementation based on mixin layers and Aspect-oriented Programming, and Section 5 features an experimental evaluation. This is followed by a conclusion in Section 6.

2 CHARACTERISTICS OF META-DATA IN DHTs

DHTs provide the data management operations of a hashtable, in particular *put* and *get*. All operations are addressed by a certain key. Each peer is responsible for a zone of the key space, and knows some other peers. Variants of DHTs primarily differ in the topology of the key space and the contact selection. Each peer forwards any operation it cannot perform on its own zone to a contact whose zone is closer to the key. The way a node determines a closer peer depends on the topology of the key space. DHTs similar to Chord (Stoica et al., 2001) or CAN (Ratnasamy et al., 2001a) organize as a circular key space in one or multiple dimensions, which is distributed among all nodes. Here, the peer forwards a message depending on the Euclidean distance of the zones to the key of the operation. In contrast, DHTs based on search trees like P-Grid (Aberer, 2001) map each node to a certain subtree-ID, and query forwarding follows subtrees with corresponding ID-prefixes. Usually, all DHT operations are performed by invoking $log(n)$ peers.

The absence of global knowledge makes it challenging to design DHT applications. There are two straightforward ways to bypass the problem: (1) setting up all necessary information at startup-time, and (2) disseminating information by flooding the DHT. But while (1) limits the range of applications, (2) limits the performance of the DHT. Before presenting our dissemination scheme, we want to investigate common characteristics[1] of meta-data items used in DHTs. Meta-data are used on three levels:

Meta-data of the DHT itself. DHTs forward messages between peers responsible for certain zones. Therefore each peer needs to know a proper set of contacts that changes whenever peers join or leave. Thus, status information regarding zone boundaries and contact information have to be updated. The source of information is the new peer or the peer that takes over the zone of a retiring peer. The addressee can be neighbors of that peer (CAN, Chord), nodes which follow a certain distribution (cf. (Kleinberg, 2000)) or other peers (Chord, P-Grid). Status updates need to *reach the addressee in time*, but they are *infrequent* and *irregularly generated*.

Data related to enhancements of the DHT. There are many different enhancements to the standard DHTs which require different meta-data. These information are *not needed in time*, and have *various creators and receivers*. For example, load balancing (Rao et al., 2003) may take place in DHTs by forwarding messages along idle paths, reorganizing the key space or changing the number of replicas of under- or overloaded zones. Here, each peer requires information about the average load (which may be rather old) and more recent data about the load of all of its contacts. Load information are disseminated at steady rates.

Application-specific meta-data. The application on top of the DHT may use the meta-data distribution scheme as well. For example, in a distributed web crawler application the peers notify others about the partitions of the WWW which they have already crawled. A further example are keep-alive statements in distributed groupware scenarios.

Summing up, nearly any application comes with different requirements for a meta-data dissemination protocol. The differences concern the origin of the data item, the addressee and the frequency the data items are generated. Some applications require meta-data in time while others do not. This leads to two important insights: (1) There is a strong demand for a *cheap protocol* that satisfies the needs of many different applications, and works at smaller costs than the traditional flooding protocols. With cost we refer to the amount of network traffic, execution time and memory consumption. (2) The number of required protocol variants is very large. However, the variants

[1]Each application comes with its very own set of requirements. Thus we cannot come up with a comprehensive list of features.

only differ in some details. From an algorithmic perspective, shipping information for load balancing is similar to the dissemination of contact information. Thus we require a *flexible architecture* which allows the developers to assemble the protocol from a set of well-defined, disjunct and reusable software fragments.

3 PIGGYBACK META-DATA PROPAGATION IN DHTs

This section describes our dissemination protocol. The protocol is intended to save resources by attaching meta-data to the messages a peer sends out anyhow. This promises to be advantageous:

- The ratio of *user data / protocol overhead* will be improved, in comparison to the use of dedicated messages. The data items are attached to existing messages, thus additional protocol headers, routing addresses etc. are omitted. This is important for information with a small footprint.
- Routing and handling efforts can be reduced by bundling multiple small data items to a single message, that can be forwarded as a whole. By using existing messages as container for meta-data, there is no additional routing overhead at all.

Data structures. We now shortly introduce the required data structures. We use Java for our implementation. Each node is represented by a class *Peer*. Each peer has the ability to send and receive messages by invoking the instance methods *send(Msg, Peer)* and *recv(Msg)*. Each message implements the interface *Msg*. Messages contain the addresser and addressee of the message, and type-specific information like query results. Each peer maintains a *Cache* containing the meta-data items it is aware of. The cache offers methods for updating the content with entries attached to incoming messages, replacing surplus meta-data items and attaching items to messages. Meta-data items are administered in a class *MetaData*. Each MetaData-object contains the user data, and information about the creator and the creation time. In addition, it can carry further information depending on the protocol variant which is actually used.

Overview. The piggyback protocol works as follows: each peer stores all data items it has generated in its own cache. As soon as a message containing a set of attachments arrives, all attachments are appended to the cache. If the number of meta-data items in the cache exceeds the limit, a certain algorithm removes the most dispensable items. In the following, we will refer to that group of algorithms as *caching strategies*.

When a peer is about to forward a message, it determines a subset of meta-data items from the cache which contains the items that are approximately most important for the receiver of the message, or should be forwarded in the direction of the receiver, respectively. This class of decision algorithms will be referred as *dissemination strategies*.

There are various possible strategies to decide which item have to be removed from the cache or which ones have piggybacked to messages. Tables 1 and 2 show some prominent examples. Note that not all of them are meaningful in any DHT variant and in any application scenario. We will come up with a showcase evaluation of some of the strategies implemented in a CAN in Section 5.

Table 1: Descriptions of the implemented caching strategies.

Caching Strategy	**Description**
Random	The peer replaces its cache entries randomly.
FIFO	The entries will be replaced which joined the cache at first. This is useful in settings where little numbers of entries should be forwarded to many nodes.
Oldest First	The cache entries with the oldest creation date will be replaced first. The strategy disseminates the most recent items if its number exceeds the transport capacity of the nodes.
Kleinberg-replacement	(Kleinberg, 2000) tells us that a well-chosen set of contacts improves the routing performance on a large scale. Here, a node which is close in units of the key space (e.g., a neighbor) is in the contact list of a certain other with a higher probability than a distant node. The caching strategy reflects this: The entries are replaced so that the receivers of the remaining ones follow a Gaussian distribution.
Most Attached	Here, the peer counts how many times a certain cache entry is transferred. Often-attached entries are replaced first.

Discussion. Our meta-data dissemination scheme is intended to serve as a cheap lazy dissemination layer on top of a DHT. It does not send out new messages; thus the number of messages the data items can be attached to is limited. Further limits are the maximum number c of attachments carried by a single message, and the size of the cache s. On the other hand, the protocol is stressed by the rate r new data items are generated.

Assume the query processing take place in rounds. In each round each of the n peers in the DHT issues and answers one query, and forwards f messages on average. The query points are distributed randomly

Table 2: Descriptions of the implemented dissemination strategies.

Dissemination Strategy	Description
Simple	Entries are attached randomly.
LIFO	The entries which are appended last will be transferred first. The strategy strongly prefers recent cache entries, and send old ones only if there is free transport capacity.
Remember Receiver	The peer ships each entry exactly one time to each of its neighbors, except for peers which have already sent the entry to the peer. This strategy reduces the number of duplicate transmissions.
Directed Forwarding	The peer attaches items so that the distance to its originator always increases. This is an adoption of the greedy routing strategy used in the CAN.
Kleinberg distribution	Entries are selected according to the distance between the peer which has created the item and the receiver of the message (Kleinberg, 2000).
CAN-flooding	The strategy exploits the structure of the CAN to transfer a cache entry to each node with a minimum of entries sent repeatedly to the same node (Ratnasamy et al., 2001b). A node that receives a message from a neighbor in dimension i forwards it in the opposite direction of that neighbor, and in dimensions $1 \cdots (i-1)$.
P-Grid flooding	This strategy adapts the idea of CAN-flooding to the P-Grid. Here, entries are forwarded along the edges of the routing tree.

over the key space. In this settings, the overall transport capacity of our protocol is $T = \frac{n \cdot f \cdot c}{r}$. Suppose each peer maintains a set of k contacts, and the dissemination strategy is intended to ship status updates the peer has generated to each of them. Then the following holds: $f \cdot c = k \cdot r$. Now assume we use a CAN (Ratnasamy et al., 2001a) implementation with a d-dimensional key space. In CANs, each node maintains at least[2] $k = 2 \cdot d$ contacts, and each node forwards $f = \frac{d}{4} \cdot n^{\frac{1}{4}}$ messages. Here, the maximum number of status updates a peer can disseminate in each round is $r = \frac{c}{8} \cdot n^{\frac{1}{4}}$. Thus, in CANs consisting of at least 4096 nodes, a single attachment piggybacked on each message allows for disseminating one status update per round for free to all of the neighbors. Calculations for other DHT variants yield similar results.

4 ARCHITECTURE, DESIGN AND IMPLEMENTATION

The spectrum of potential applications and target platforms as well as their requirements and characteristics is extremely broad and diverse. Furthermore, we have observed that strategies often differ only in details, e.g., a threshold or some additional constraints. Out of these considerations we derived following requirements on the implementation and integration of strategies:

1. The implementation and integration of (new) caching and dissemination strategies must be as easy as possible.
2. It should be possible to integrate more than one caching as well as dissemination strategy.
3. Different caching strategies must be combinable with different dissemination strategies.
4. Strategy code implements a special-purpose concern and should therefore be modularized and separated from the core DHT implementation.
5. A developer should be able to create new strategies out of existing ones with minimal effort.

To meet these requirements we have investigated in a novel architecture. It makes use of the software-engineering concepts of AOP and mixin layers.

Aspect-oriented Programming. AOP was first introduced by (Kiczales et al., 1997). The aim of AOP is to separate crosscutting concerns. It is well known that the *separation of concerns* leads to maintainable, comprehensible, reusable and configurable software (Kiczales et al., 1997). Common object-oriented methods fail in this context (Kiczales et al., 1997; Czarnecki and Eisenecker, 2000). The idea behind AOP is to implement so called orthogonal features as *aspects* separately. The core features are implemented as components, as with common design and implementation methods. An aspect weaver brings aspects and components together.

For our implementation we have used *AspectJ*[3], a language extension to Java. Mainly, we have used AspectJ to separate the code for the caching and dissemination strategies from the core implementation of the DHT. The left side in Figure 1 depicts a schematic design of several dissemination and caching strategies implemented as aspects. It can be seen that the aspects are separated clearly. This allows us to design a generalized protocol implementation, which does not depend on the DHT which is actually used, and increases the reusability and the ability of plugging strategies.

Mixin Layers. During the design phase we have observed that we need often only small modifications of the caching and dissemination strategies to specify different protocol characteristics. For example, from the algorithmic point of view the caching strategy random differs from FIFO only in the order the

[2]In a CAN where peers join and leave frequently, the join-protocol results in a irregularly fragmented key space.

[3]www.aspectj.org

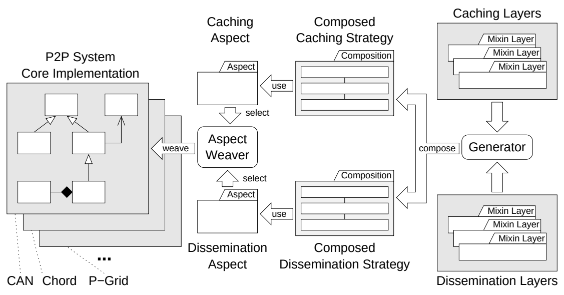

Figure 1: Generating and applying strategies using mixin layers and AOP.

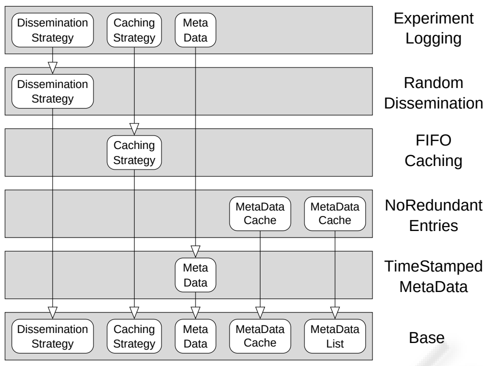

Figure 2: Stack of mixin layers for FIFO Caching and Random Dissemination.

items are removed. Furthermore, some features are common for many strategies, e.g., to keep track of the peer a certain item was already sent to. Implementing a new strategy in form of a separate aspect for each variant would be inefficiently and errorprone, because approved code is not reused. Therefore, we have decided to combine the approach of Aspect-oriented strategies with a mixin layer-based implementation.

A mixin layer is a static component encapsulating fragments of several different classes (mixins) so that all fragments are composed consistently. Mixin layers are an approved implementation technique for component-based layered designs. Advantages are the high degree of modularity and the easy composition (Smaragdakis and Batory, 2002).

We have used mixin layers to implement the variants of the different caching and dissemination strategies and the *AHEAD Tool Suite*[4] to automatize the configuration and generation process. Doing, so we had to decompose the mechanisms for caching and dissemination into fine-grained layers. Based on this design we are able to compose these layers to generate different strategy variants. AOP is then used to apply the configured strategies to the core of the DHT. Figure 1 depicts the schematic overview of configuring and applying strategies to the DHT: At first, a generator (here the *AHEAD Composer*) composes the strategies according to a (user) specification. Then, the composed strategies are applied to the DHT core using aspects and the *AspectJ Weaver*. Using mixin layers, we do not need multiple aspects. Instead, we need only two of them: one for applying the caching strategies and the other for applying the dissemination strategies. Figure 2 depicts the stack of mixin layers implemented for random dissemination and FIFO caching. The enclosing grey boxes are the mixin layers. The included rounded boxes are the inner mixin classes. The figure depicts only the inheritance relations between the inner classes. Other relations (e.g. associations) as well as references to external classes are omitted. The *Base* layer includes the needed data structures: *MetaData*, to provide a base for application specific data, *MetaDataCache* for storing the cached meta-data, *MetaDataList* to attach meta-data to messages, as well as the strategy base classes (interfaces) *CachingStrategy* and *DisseminationStrategy*. Tangible strategies (in our example random dissemination and FIFO caching) implement these interfaces to provide the desired algorithm. The layer *TimeStampedMetaData* refines the *MetaData* base by adding time stamps to each meta-data object. The layer *NoRedundantEntries* prevents the meta-data cache and the messages from storing equal meta-data objects. *ExperimentLogging* creates the logging dump which we used to analyze our experiments.

The advantage of using mixin layers is that we can easily derive and combine new variants of strategies. To introduce a new caching strategy we have to implement the interface *CachingStrategy*. Doing so, we can remove the currently used strategy and apply the new one. In this way we can easily combine caching strategies with dissemination strategies. Furthermore, we are able to reuse the base data structures as well as existing strategy implementations. Imagine a proximity FIFO caching strategy which combines FIFO caching with the contact information of the peer. This might be useful in order to limit the range of meta-data to the contacts of a peer. If a cache entry was generated by an immediate contact, the caching strategy is FIFO. If not, the cache entries are dropped. The proximity FIFO caching strategy reuses the overall code of the simple FIFO strategy.

[4]http://www.cs.utexas.edu/users/schwartz/Hello.html

5 EVALUATION

Our evaluation is divided into two parts: The first one is intended to show that our protocol framework and our software engineering approach is working as envisioned. Therefore, we have used a CAN implementation of our own, and have extended it by several caching and dissemination strategies; and we will come up with a detailed description of a simple one. The second part proves that the protocol itself meets the requirements for a meta-data dissemination scheme. Here we assembled all meaningful protocol variants from the strategies we have realized for the first part, and evaluated it by the means of experiments.

Evaluating the software design. The CAN implementation we have used for evaluation implements all features of the originally proposed protocol (Ratnasamy et al., 2001a) as well as some proprietary features (Buchmann and Böhm, 2004a; Buchmann and Böhm, 2003). For our experiments we have implemented the dissemination strategies *Random*, *Directed Forwarding* and *CAN-flooding*. In addition, we realized the caching strategies *Simple* and *FIFO*. Table 1 and 2 gives a short description. We expect that AOP will be useful for the separation and the easy plugging and combination of strategy code. A feature-oriented view and mixin layers should help to implement strategies highly configurable.

We start with the caching strategy. Here, the strategy ”Random”, which is shown in Figure 3, serves as a simple example. It collects the meta-data of incoming messages to update the peers cache (*MetaDataCache*). Thereby it chooses the message entries as well as replaces older cache entries randomly. The random caching aspect (Line 1) introduces for each *Peer* a cache for storing meta-data objects (Line 2,3). The aspect intercepts calls to the method *Peer.recv* to get access to incoming messages (Line 5-7). The attachments of each incoming message are used to update the peers meta-data cache (Line 11). The actual implementation of the caching strategies is encapsulated in the *put* method. We omit the detailed discussion of the random caching algorithm because its implementation is straight forward.

Figure 4 depicts a dissemination aspect (Line 1) for the dissemination strategy ”Simple”. It uses the cached meta-data to disseminate them through the DHT by attaching them to outgoing messages randomly (cf. Tab. 2). At first a list for attaching meta-data objects is introduced to all messages (Line 2,3). Moreover, the aspect intercepts all calls to the method *Peer.send* which sends all outgoing messages (Line 5-7). Doing so, the aspect has access to these messages and can add the peers cached meta-data objects (Line 8). The method *piggyBack* encapsulates the actual dissemination strategies (Line 10). In our case the strategy chooses a random set of cached meta-data (Line 14) and attach them to the outgoing messages (Line 11,12).

```
1   aspect RandomCachingAspect {
2       MetaDataCache Peer.m_meta_data =
3               new MetaDataCache();
4
5       before(Peer peer, Msg msg) :
6               target(peer) && args(msg) &&
7               call(* Peer.recv(Msg)) {
8                   put.(peer.m_meta_data,
9                            msg.m_meta_data)); }
10
11      void update(Peer peer, MetaDataList mlist)
             {
12          // replaces the entries of a peers
                cache
13          /* ... by a random list */ }
14  }
```

Figure 3: The random caching aspect

```
1   aspect RandomDisseminationAspect {
2       MetaDataList Msg.m_meta_data =
3               new MetaDataList();
4
5       before(Peer peer, Msg msg) :
6               target(peer) && args(msg) &&
7               call(* Peer.send(Msg)) {
8                   piggyBack(peer, msg);   }
9
10      void piggyBack(Peer peer, Msg msg) {
11          msg.attach(getRandomList(msg,
12                  peer.m_meta_data)); }
13
14      LinkedList getRandomList(Msg msg,
15          MetaDataCache cache) {
16          /* returns a random list of items */ }
17  }
```

Figure 4: The random dissemination aspect.

The advantages of modularizing and separating caching and dissemination strategies from each other and from the core implementation of the underlying CAN is that they can easily be applied to and removed from the CAN. Furthermore, one is able to combine different variations of caching and dissemination strategies (cf. Sec. 5). Finally, the maintainability and comprehensibility of the CAN core code and the separated caching and dissemination strategies is preserved.

We have implemented all mentioned caching and dissemination mechanisms using the combined approach (AOP + mixin layers). Mainly, we have investigated in this approach to get insight how modern software engineering methods can be exploited to achieve modular, reusable, extensible and customizable software component especially for distributed (P2P) Systems. Our results are positive: We have observed that the separation of special concerns like the caching and dissemination of meta-data leads to maintainable and comprehensible code. AOP and AspectJ are appropriate to separate code of special crosscutting concerns. Mixin layers allow to decompose the strategies functionalities into modular combinable building blocks (cf. Fig. 2). They support configurable, reusable and extensible implementations. The combined approach reduces the complexity of implementing new strategy variants and combinations and therefor the expenditure of time.

The implemented mixin layers and aspects are reusable in other DHTs because the rely only on a simple interface containing the methods *send* and *receive*. These are available in most DHTs and other unstructured P2P systems.

Evaluating the protocol We now want to show that our meta-data dissemination protocol is working, and is applicable to various settings. Due to the lack of space, we limit our evaluation to three showcase experiments investigating in the characteristics of the dissemination strategies *Random*, *Directed Forwarding* and *CAN-flooding* and the caching strategies *Simple* and *FIFO*.

All experiments are performed on a cluster of 32 loosely coupled Linux workstations equipped with 2 GHz CPU, 2 GB RAM and Fast Ethernet. We run experiments with a four-dimensional CAN consisting of 10,000 peers issuing 1,000,000 randomly distributed queries (see (Buchmann and Böhm, 2004b) for an exhaustive description of our experimental setup.) All evaluations are based on rounds. In one round each peer issues and answers one query. The cache size is set to 100 elements, and each message piggybacks up to 10 meta-data items. At the beginning, each peer owns one initial meta-data item.

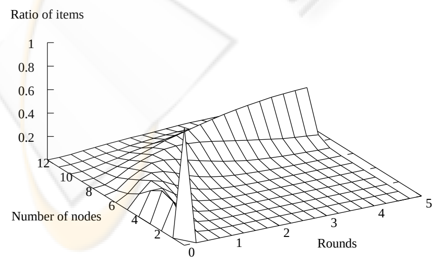

Figure 5: Item dispersion among the contacts.

At first we investigate in the suitability of our protocol for status updating. Here it is important to publish information about the state of a node to its contacts (cf. Sec. 2). We have used the CAN-flooding dissemination strategy and the cache strategy FIFO. See (Ratnasamy et al., 2001b) for a detailed description of CAN-flooding. In CANs, the number of contacts approximately follows a χ^2-distribution; in our four-dimensional setting the most nodes have 9 contacts. Figure 5 shows the distribution of meta-data items. The z-axis shows the ratio of meta-data items which are known by a certain number of nodes at a given time. For example, all items are known by one node at round 0, and approximately the half of the items are known by 9 nodes at round 5. The figure indicates that the majority of the items have reached the contact after at most 2 rounds. By considering that the protocol ships such information for free, this is a very positive result.

The next experiment examines the number of duplicates sent by different dissemination strategies. Obviously, due to the limited transport capacity shipping many duplicates decreases the effectiveness of the protocol. On the other hand, it would improve the fault tolerance and the time to reach an adjacent (in terms of hops between the nodes) node. Consider the CAN-flooding dissemination strategy for example. Here, having a single failure near the creator of an item could prevent a large fraction of the CAN from obtaining it. Figure 6 shows a comparison of the dissemination strategies random, directed forwarding and CAN-flooding in respect to the number of duplicates sent within 5 rounds. For example, after 5 rounds directed forwarding has transferred 1% of all items more than times 500 to peers which already received them. It is obvious that the CAN-flooding strategy comes with the lowest number of duplicates. In a CAN where all zones have the same size, the number of duplicates would be zero. In reality, the join-protocol leads to zones with different sizes. Here, a few items are sent more than 500 times to the same nodes twice. In contrast, it is surprising that the random dissemination strategy results in less duplicates than directed forwarding. The explanation is that directed forwarding prefers a few items for each direction.

This leads to the question: how long does it take to send items to a certain number of nodes? We address this issue in the next experiment. Again, we use the three different dissemination strategies. But now we measure the time the items need to reach an arbitrarily chosen number of 20 different nodes. We would expect that the majority of items will be transferred to the other nodes within one or two rounds, while a few number of items will need a very long time. The result of our experiment is shown in Figure 7. Because it utilizes the underlying CAN forwarding scheme to obtain a complete covering of all zones, CAN-flooding is the only strategy we have measured,

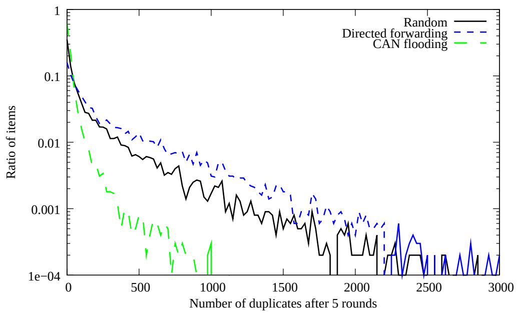

Figure 6: Duplicate items transferred within 5 rounds.

that disseminates all items to at least 20 nodes within less than 5 rounds. However, by considering the beginning of the curve, random and directed forwarding are much faster. These strategies ship $1/3$ of all items to 20 nodes within the first $1/4$ round, while CAN-flooding requires a half round. Thus, in applications where items would be useless if they do not reach its destinations very quick, directed forwarding promises to be an applicable strategy.

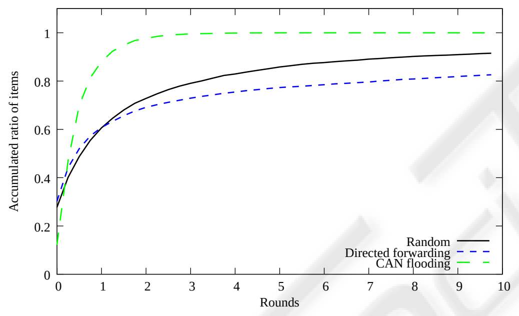

Figure 7: Time to disseminate an item to 20 nodes.

6 CONCLUSIONS

This article investigates in a novel lazy dissemination protocol for various DHT variants. The protocol allows to disseminate information free of charge, if they are not needed in time and have a small footprint. Our approach of combining mixin layers and AOP leads to highly configurable, modularized and separated strategy implementations. This combination makes it possible to cope with the wide spectrum of application requirements. Because of the limited space, we have shown only a few showcase experiments. However, all of our experiments indicate that our protocol is working as envisioned. In the future we want to examine our protocol in-depth on the basis of a specific application.

REFERENCES

Aberer, K. (2001). P-Grid: A Self-Organizing Access Structure for P2P Information Systems. *LNCS*, 2172.

Boyd, S. et al. (2005). Gossip Algorithms: Design, Analysis and Applications. In *Proceedings of the 24th Infocom*.

Buchmann, E. and Böhm, K. (2003). Effizientes Routing in verteilten skalierbaren Datenstrukturen. In *Proc. of the 10th BTW*.

Buchmann, E. and Böhm, K. (2004a). FairNet - How to Counter Free Riding in Peer-to-Peer Data Structures. In *Proc. of the 12th CoopIS*.

Buchmann, E. and Böhm, K. (2004b). How to Run Experiments with Large Peer-to-Peer Data Structures. In *Proc. of the 18th IPDPS*.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison Wesley.

Gribble, S. D. et al. (2001). The Ninja Architecture for Robust Internet-Scale Systems and Services. *Computer Networks*, 35(4).

Kiczales, G. et al. (1997). Aspect-Oriented Programming. In *Proc. of the ECOOP97*.

Kleinberg, J. (2000). The Small-World Phenomenon: An Algorithmic Perspective. In *Proc. of the 32th STOC*.

Rao, A. et al. (2003). Load Balancing in Structured P2P Systems. In *2nd Int. Workshop on Peer-to-Peer Systems*.

Ratnasamy, S. et al. (2001a). A Scalable Content-Addressable Network. In *Proc. of the ACM SIGCOMM Conf.*

Ratnasamy, S. et al. (2001b). Application-Level Multicast Using Content-Addressable Networks. *LNCS*, 2233.

Rodrigues, L. and Pereira, J. (2004). Self-Adapting Epidemic Broadcast Algorithms. In *FuDiCo II: S.O.S. Survivability*, Italy.

Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Int. Conf. on Distributed Systems Platforms*.

Smaragdakis, Y. and Batory, D. (2002). Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering Methodology*, 11(2).

Stoica, I. et al. (2001). Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proc. of the SIGCOMM*.