Simple and flexible DHTs

Luis Galárraga
Recent Advances in Computer Systems, Saarland University, Winter Semester 2010/11
Saarland University

February 9, 2011

Abstract

Distributed Hash tables are nowadays commonly used as the storage underlying infrastructure for many applications because of their decentralized design, scalability and fault tolerance. Unlike traditional storage systems, they offer an amazingly simple interface to store and retrieve streams of bytes, leaving the entire responsibility of the data semantics and manipulation to the application layer. This brief report suggests a series of improvements to the existing DHT design in order to provide type awareness and extended access semantics to such systems, transferring a significant part of the data management logic to the storage layer and relieving applications from the complexity derived from the nature of the data. The major goals of this approach are to impact the traditional programming style when relying on DHTs as well as to facilitate the sharing of a single storage system by multiple applications with diverse requirements due to the new level of flexibility introduced.

1 Introduction

DHTs (Distributed Hash Tables) are nowadays commonly used as the storage underlying infrastructure for many distributed applications, mainly because of their decentralized nature, scalability and fault tolerance [1]. Unlike traditional storage systems (e.g relational databases), they offer an amazingly simple interface to store and retrieve information which makes them an attractive solution for applications like distributed file systems, content distribution networks, peer to peer file sharing systems, cloud computing storage services among others. As a brief description, DHTs offer a simple key-value storage system with a minimalist get/put interface where the key determines in which node of the network, the value is actually stored. The key can be seen as a point in a huge identifier space and is normally generated by applying a hash function to the value or part of it. On the other hand, the value is a raw string of bytes whose structure and semantics depend on the scenario.
The upcoming sections of this report are aimed to propose a series of extensions to the original DHT design by incorporating *type awareness* and *extended access semantics* without compromising their inherent simplicity or incurring in a considerable performance penalty. I claim that providing such awareness creates an outstanding opportunity to optimize resources like space, network bandwidth and processing time for several applications. It also propose a change to the traditional programming style used in client applications as well as an opportunity to share a single storage system by multiple applications with diverse (maybe conflicting) requirements.

2 Motivation

Applications using database systems have benefited from this idea for years, delegating part of the data semantics logic to the storage-layer through mechanisms like table views, functions, procedures and triggers. In most cases, it releases the application from processing time and design complexity, since the storage system is normally well optimized to process the data.

3 Type awareness

From a very simplistic view, DHTs are storage systems not much different from a file system or a database, however so far they are totally oblivious to the structure of the data, leaving the entire responsibility of this semantic interpretation to the application layer. Relational databases use schemas to describe the organization and structure of the relations which are mainly used to specify integrity constraints, but also for efficiency reasons like finding the best query plans or join order schemes. These optimizations rely on metrics about the data; examples are the selectivity of a relation and the range or probability distribution of an attribute. In the second example, that information cannot be inferred without any knowledge about the nature of the data. In the context of our targeted applications, a key-value pair could store a big integer, a fragment of a file system table or an image stream.

4 Active storage

The concept of active storage systems is quite old. Active disks [2], active networks [3], active objects [4] and database triggers execute pieces of code in response to events associated to the data, like a read or write access. In all cases, such schemes are aimed to relieve some other component from processing load derived from the nature or semantics of the data, in a transparent way without altering the traditional interface between the storage and application layer. If we consider the type awareness introduced in previous section, a DHT could use that information to associate behavior based on the data type of the stored value. Using our previous example, suppose a DHT-based file system, using a

key-value pair to store a fragment of a log wants to add an entry. By treating the value as a raw sequence of bytes, the application will have to overwrite the whole fragment plus the new entry whenever it is updating the log. Even though one might argue this can be easily circumvented with other techniques like batching updates or small size fragments, the application can certainly do better by just avoiding the transmission of repeated information. As a second example, consider a SVG image storage system which uses a key value pair to store an image or set of images. A modification to the image might imply a change in a single line of the markup; whereas for the DHT it means a normal put operation. The design complexity to implement these kind of operations in an efficient way relies entirely on the application layer.

5 The idea

The general idea is to implement data type awareness and active storage in an existing DHT solution with minimal impact in performance and without compromising the simplicity of the traditional interface. Applications should be able to define their own data types for the information they manage and extend their access semantics.

Again, databases are a good example of type awareness. They normally define a set of data types that meet the needs of most applications. There also exist several initiatives that extend standard SQL types tailored for specific applications [5–8], however in the general case standard types consist of scalars, strings and time structures which are too limited for the foregoing examples. Furthermore, since the active storage features will allow to extend the semantics of the access operations, it facilitates the definition of arbitrary user-defined data types (e.g log entry in the context of a distributed file system). Some of the aforementioned applications may benefit from the inclusion of common MIME types [9] as part of the data type set.

For the extended access semantics, many approaches have been deployed. Comet [4] introduces the concept of active objects where the data and its behavior (in the form of code handlers) are stored inside the value in the key-value pair. Other solutions associate data to a piece of code located somewhere in the environment, which is triggered in response to every access controlling the result of the operation. For example, active disks [2] are stream-based solutions which means code, referred as disklets in the original proposal, is associated to streams of data whose source can be a file or set of files as well as the output of another disklet. The aim of this solution was to reduce the central processor load and bus usage by running disklets on the hard disk processing unit. Other approaches like Watchdogs [10] assign code at file level extending the semantics of common file system operations like lookup, open, read and write.

Based on the approach followed by Watchdogs, I propose to extend the semantics of put and get operations based on the data type of the key-value pair. One feature is required to achieve the intended level of flexibility: the ability to provide arguments to the access operations. Suppose our key-value pair stores a versioned text file. We can extend the semantics of the *get* operation by including the version as an argument in the request:

```
dht.get(key, "version=5")
```

which would execute the handler associated to the data type (if any) specified for this key [1]. The handler would just take the version number and retrieve the state of that file at version 5 using his knowledge about the data organization. The data type should be part of the retrieved information.

On the other hand a *put* operation might be extended, so that creating a new version of the file can be achieved by applying a patch:

[1] The application could define a custom type "versioned text file"

```
dht.put(key, "2a3,4 \n> Jean JRS@pollux.ucs.co", "mode=patch")
```

where "mode=patch" redefines the normal semantics of *put* so that the sent text does not override the value but it is merged. Note that the opportunity of sending a patch instead of the whole file, might represent a substantial save of bandwidth resources.

The data type policy addresses a wide range of applications, but nevertheless it may not be sufficient in some cases. Think of an application that uses key-value pairs to store caches and relies on the storage system for automatic entry eviction. Even though they all have the same internal structure, every cache might need to implement a different replacement policy, justifying the presence of a per key policy. Code associated according to this policy should override any existing type policy; however the system might provide a way to reuse data type level behavior.

Based on the previous examples, our programming model leads us to the definition of code handlers, triggered in response to *get* and *put* operations.

```
<result, mimetype> onGet(key, arguments);
result onPut(key, value, mimetype, arguments);
```

The returned values for *onGet* and *onPut* define what is replied and what is stored as new value respectively.

5.1 Possible applications

5.1.1 Image storage solutions

Image storage repositories can be benefited from this approach because it could be used to save resources like bandwidth and storage space. A key value-pair may store an image or group of images using any format or compression scheme and extend the access semantics to apply several operators. Thumbnail generation, image compression/decompression and composition as well as customized filters could be implemented in the handlers.

5.1.2 DHT-based file systems

There exist several distributed file systems implementations relying on DHTs as storage layer [11–13]. Even though their capabilities, properties and guarantees are different, they have at least one challenge in common: use the storage system as efficiently as possible. Backup schemes, usage and monitoring measurements, automatic log purging, access control, virtual files, encryption among other applications can be easily constructed by leveraging the extended access semantics without dedicated (possibly external) software components.

5.1.3 Version control systems

If a single key-value pair is able to store all the versions of a single file or object like in our first example, the implementation of distributed version control systems becomes much easier. Repository hooks, data encoding considerations and patch management can be implemented in a very straightforward way.

5.1.4 Delta encoding

Applications like content distribution networks could benefit from a repository-level implementation of delta encoding. Suppose we have a content distribution network where several proxies get a request from an expired element. The origin server could just transmit the differences between the newest and the expired version. Proxies could also repeat this approach to communicate with each other.

5.1.5 Public suscribe schemes

An application can implement a public suscribe scheme by creating a key with an associated handler that somehow [2] reports the suscriber as soon as a *put* operation happens.

5.1.6 Many applications, one instance

So far, to provide every single feature mentioned in this section would imply to implement a customized version of a based DHT system, tailored exclusively for that need. With this approach many applications can rely on a single instance of a DHT, which in general supposes a save in development effort as well as less metadata overhead [2].

5.2 Implementation clues and challenges

Even though the goal of this report is not to propose a concrete implementation for the proposed ideas, I think it is worth providing some insight about the development implications, in order to convince the reader about its feasibility and the great benefit obtained in return from a moderate effort in terms of design and development.

Since our assumption is platform-agnostic, we assume for simplicity that our DHT provides capabilities like huge key space, simple put/get/delete interface, ability to specify the level of replication and logarithmic sized routing tables per node. Note however, that this modifications are totally independent of the routing layer or metric space defined.

5.2.1 Changes in access interface

The signature for put and get operations must be extended to allow the inclusion of parameters. The URI syntax for parameters could be used:

```
get(key, arguments)
put(key, value, datatype, arguments)
```

where arguments is a string of the form $name1 = value1\&name2 = value2...$ like in URLs.

5.2.2 Security considerations

The introduction of the active storage capability certainly poses a series of challenges in terms of security. Some of the distributed systems mentioned so far, run in out of the firewall environments where the presence of untrusted nodes introduces a considerable amount of complexity to the goal of designing a secure solution. Additionally, it enforces a level of centralization in terms of administration because the data type policy implies somebody has to explicitly define this association.

5.2.3 Metadata management

Metadata consists of supported data types, the association between a key-value pair and its handler and the handlers code. The naive approach is to store this information in the DHT. Since these key-value pairs are vital for the system, they have to be highly available. Consistency becomes less urgent under the assumption that these associations will not change frequently in a system. An interface for applications to manage this metadata has also to be designed.

[2] The exact implementation will depend on the defined security policies

[2] It would be nice to compare the overhead in metadata between this approach and certain number of standard DHT instances when used by the same applications

5.2.4 Runtime environment for code handlers

This is the factor that requires more care at design time for two reasons:

1. It is the major hazard. Therefore a set of security policies about resource consumption, isolation and external interaction must be defined in order to minimize the impact of malicious code handlers.

2. Replication introduces extra challenges in relation to error semantics and consistency since a put operation now triggers the handlers in all replicas and of course the result of their execution should be the same, thus determinism guarantees need to be enforced.

5.2.5 Programming interface

So far we have sketched the interface the new DHT provides to client applications, but not the programming interface for handlers whose design has to match our goals as well as the security policies.

6 Conclusion

The proposed set of changes is aimed to bring existing concepts implemented in other architectures to the DHTs' world, in order to combine the inherent decentralization and fault tolerance of these systems with the expressivity of database systems and resource oriented architectures caring about the semantics of the data. I claim this is not any sort of panacea for all DHT based applications, but for many important ones which makes it a path worth exploring despite the series of design challenges in the field of systems security.

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, pp. 17–32, February 2003.

[2] A. Acharya, M. Uysal, and J. Saltz, "Active disks: programming model, algorithms and evaluation," *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pp. 81–91, 1998.

[3] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 26, pp. 5–17, April 1996.

[4] R. Geambasu, A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, "Comet: An active distributed key/value store," in *Proc. of OSDI*, 2010.

[5] D. Guliato, E. de Melo, R. Rangayyan, and R. Soares, "Postgresql-ie: An image-handling extension for postgresql," *Journal of Digital Imaging*, vol. 22, pp. 149–165, 2009. 10.1007/s10278-007-9097-5.

[6] "What is postgis?." http://www.postgis.org/.

[7] "Oracle spatial & oracle locator: Location features for oracle database 11g." http://www.oracle.com/technetwork/database/options/spatial/index.html.

[8] "Oracle multimedia." http://www.oracle.com/technetwork/database/multimedia/overview/index.html.

[9] "Mime media types." http://www.iana.org/assignments/media-types/.

[10] B. N. Bershad and C. B. Pinkerton, "Watchdogs: Extending the unix file system," in *USENIX Winter*, pp. 267–275, 1988.

[11] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: a read/write peer-to-peer file system," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 31–44, December 2002.

[12] J.-M. Busca, F. Picconi, and P. Sens, "Pastis: a highly-scalable multi-user peer-to-peer file systems," in *Euro-Par'05 - Parallel Processing*, Lecture Notes in Computer Science, (Lisboa, Portugal), Springer-Verlag, Aug. 2005.

[13] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, (Chateau Lake Louise, Banff, Canada), October 2001.