Bunshin: DHT for distributed applications

Rubén Mondéjar, Pedro García, Carles Pairot *

Dept. de Matemàtiques i Informàtica
Universitat Rovira i Virgili
43007 Tarragona
{ruben.mondejar, pedro.garcia, carles.pairot}@urv.net

Abstract

We present Bunshin, a distributed hash table, which is constructed on a p2p structured network and is characterised by its robustness and reliability thanks to an active system of replication and an adaptive system of caching.

In order to give to support to other middleware architectures or final applications, it offers the services of insertion and recovery of keys/values of determinist form, allows for multifield values for each key and manifold contexts for each application.

Upon these services the search engine is founded. Based on keywords, it allows insertions and consultations by means of this. In addition, it incorporates mechanisms to establish communications between keys and notifications of new connections on the keys that are of interest to us.

1. Introduction

Throughout the last years, Internet has been growing in number of users. The bandwidth of the nodes has been increased considerably and the appearance of many applications of global scope has popularised the use of the network of networks. On the other hand, the computers every time have a capacity of greater calculation and the possibilities of compartment of resources have happened to be more and more and more important.

A system based on the p2p architecture so does not have the figure of a central node that it controls to the others, and as it happens in the traditional client-server architectures. The nodes act all like equal, so that no of them assumes, a priori, a greater protagonist than the others. In this context, the p2p computation has unfolded a great number of applications oriented to the communication and collaboration, as well as distributed computation.

One of its basic characteristics is the high availability, thanks to the existence of many peers in a group, thing that it facilitates that at least peer of the group it can satisfy a request with a user.
In the present generation of p2p networks is tried to give solution to the commented problem, so that the location of the resources is determinist.

Consequently, if a resource in concrete is in the network, then we will be able to obtain it. For such intention, the nodes in the network are grouped of structured way, habitually, as ring or tree. The challenges nail of these systems are to avoid the bottlenecks that can be produced in certain nodes and therefore, similarly distribute the responsibilities between the existing nodes and to adapt to the continuous entrances and exits (and also fallen) of nodes.

The p2p overlay networks are efficient, resistant to failures, and self-organising. Existing examples of this type of systems are Chord[1], Symphony[2], Pastry[3] or Kademlia[4]. The decentralised p2p overlay networks also receive the name of networks or KBR (Key Based Routing) substrates, since the routing one of the messages is function of the key of the nodes.

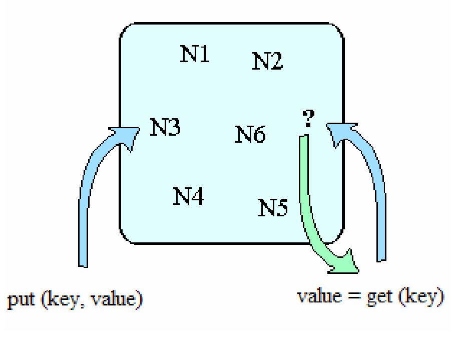


Figure 1. Basic interface of a DHT

* Quick Translation by Reinout van Schouwen, reinout@cs.vu.nl, www.iids.org

KBR networks provide services like distributed hash tables (DHTs). The abstraction which facilitates a DHT with the standard interface *put (key, value)* and *get (key)*, that we can see in figure 1, is similar to the use of the classic hash tables, but in this case, *buckets* are the physical nodes of the network, relating the rank of keys of these to the identifiers of the nodes to which they belong.

The DHT uses replication to assure that the kept data survive the falls of the nodes and use caching to make transitory copies and to balance the request load on these.

In this article we will begin describing the scene where we located ourselves and next we will expose the architecture of Bunshin, the obtained services that offer and results. Ultimately, we will finalise with a discussion of works related to the subject, of the future work and the conclusions.

2. Scenario

The scene where Bunshin is located is the Planet project [5]. In which we took of the same one we have developed to a series of middleware architectures for distributed decentralised environments. Bunshin was born of the necessity to create persistence services in Dermi[6]. Dermi is middleware of distributed objects constructed upon Pastry, that in first instance used the DHT of this one, PAST[7], for its decentralised directory service. Because PAST did not have a reliable implementation and that it was not easy to make modifications, since there was to re-adapt them to each new update, we decided to implement our own DHT. Thanks to the experience like users of a DHT when using PAST and to the requirements that we saw that we needed, implemented Bunshin (which this constructed on Pastry and like this one benefits from the services that offer). Not only Dermi uses Bunshin, but that from then began to arise a series from applications in the Planet project that made use of it. In figure 2 we can see an example of the typical architecture of these applications.

Of these we can emphasise **PlanetDR**, a service of decentralised repository that uses the DHT to keep the information on the communities and to make searches on keywords.

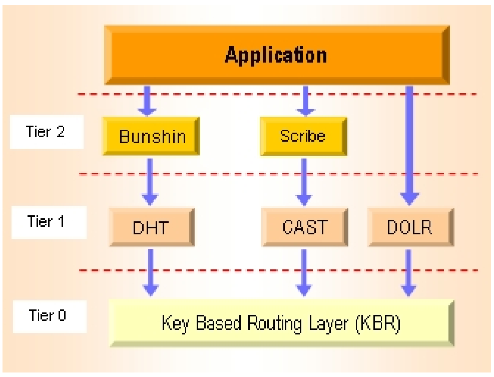


Figure 2. Three tiers architecture (overlay networks)

3. Architecture

The architecture of Bunshin can be divided in two layers. The DHT application layer itself and the layer of services that are constructed upon it. In a following section the services of the upper layer will be analysed, but first we will take a detailed look at the design of the primary layer, commenting on the following characteristics one by one:

- Active system of replication
- Adaptive caching scheme
- Different ways of persistence
- Multifield values
- Multicontext structure

3.1. Replication

To be able to guarantee the reliability of the data that are stored in environments as dynamic as p2p networks, these need not only to be in the responsible node, but there must exist more nodes that contain the data, so that if the responsible node leaves the network for any reason, their data will not disappear. In order to obtain to this objective the information it is replicated and actively managed as will we see now.

In order to be able to decide who will be the nodes that keep the replica of a concrete key/value, the responsible node chooses their nearer neighbours, obtaining the list of successors of the network overlay. With this it guarantees that if the responsible node fails, his immediate successor already will have the data that will need. In addition, if the overlay that we used groups the neighbours by proximity, as it is the case with Pastry, the operations with these will be more efficient.

The candidates to maintain the replicas are obtained by means of the method *replicaSet()*, that the overlay network provides us, which returns the list of ordered successive nodes of a particular key. Thanks to her also we are warned of the entrance or exit of the nodes by means of the method *update()*. In addition, to prevent lost with messages or the events they make the following verifications:

1. To verify if all the keys at the moment kept by a node still belong to him, by means of the verification of if the present node is the first candidate returned by the method *replicaSet()*. If it discovers that some key/value pair no longer is its responsibility, it will reinsert it in the network, thus arriving at its new owner.
2. To verify if the nodes in which there are replicas still kept follow assets, and in addition if the number of replicas of each key is equal to the wished number. But some of the two conditions is fulfilled means that there is to make more you talk back and therefore the necessary candidates will choose themselves and the replicas were made that lack.
3. Verification of which the nodes of which we have replicas even follow assets. But it is thus, will be made the same procedure that when we discovered that a key/value pair no longer is ours. The difference in this case is that with a much more high probability, at the new owner it arrived to him more than a copy of that key/value, since all the replicas will realise it. For this case, the criterion to follow by the new owner is to keep the key/value pair with newer version, being very important in this point the **version control** that is made in the update of the values.

The version control follows a non-temporal policy, that is to say, it does not pay attention to the date at which the value was updated, since we cannot assume that all *peers* are time-synchronised. For it the technique that is followed is to thus assign a number of update on an alive copy in the system, being easy to recognise if one replica sends a value to us previous to that which we already have. If it concerns a copy without version number we will assume that it is a new update.

3.2. Caching

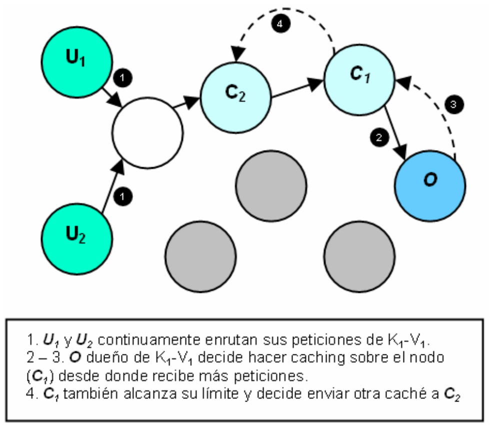
The keys/values copy provides two important characteristics, being the reliability and the yield. Since we have already commented, the reliability and the tolerance to failures are provided by the replication system, so to improve the yield the technique of caching is used.

In this case, and specially in wide-area environments, some nodes can begin to temporarily undergo request overload by near clients. In order to solve this problem, the **adaptive system of caching** is used, which we now will explain. The use of these caches is dynamic and adaptive according to the number of requests in a period of time on a certain key/value, thus temporary copies being created that do not maintain the state under demand.

In order to diminish the number of accesses to a single node and a load balance takes place, the owner of the key/value will send a copy so that [it will be cached?] to the immediately previous node from where the maximum number of requests is received. In this manner, a temporary copy will stay in cache and the owner will let receive requests of that node during the time that the cache lasts.

If the same happened to the neighbouring node that now has this cache, past the limit, this one would activate a cache in the interested neighbouring node. The idea is that the copy is propagated in caches them of the nodes more interested then, so that a node is not saturated being owner of a very popular key/value, but that activates copies near the nodes that make requests.

In order to be able to intercept requests routed to each node, we make use of the method *forward()*. This method that the network overlay provides serves to decide if the message must continue to be routed or not. So we in addition it is here where we verify if the key of the request has value in cache in the present node. If the consultation is satisfactory then the message will stop being routed and the result will be sent directly.



1. ***U₁*** y ***U₂*** continuamente enrutan sus peticiones de K_1-V_1.
2 – 3. ***O*** dueño de K_1-V_1 decide hacer caching sobre el nodo (***C_1***) desde donde recibe más peticiones.
4. ***C_1*** también alcanza su límite y decide enviar otra caché a ***C_2***

Figure 3. Adaptive activation of caches.

In figure 3, we can see a summary of this adaptive system of caching. Also it is possible to be summarised saying that back-pressure is applied to a technique where we obtained that the cache propagates towards the center of interest in a determined temporary space. Spent a time caches them expire and if the request bursts already have stopped, caches them they will be deactivated.

3.3. Persistence modes

Bunshin have been designed with the extensibility in mind and therefore it supports different ways of persistence:

- Memory
- Disc
- *FileMapping*

The way of **persistence in memory** is appropriate for the replicated data or in cache and also for the data of applications of test.

The standard way would be the way of **disc persistence**, keeping all the data from a context in an assigned directory. Most interesting it has to emphasise is the **FileMapping** that being based on the previous one, this mechanisms allow to map the files in the original state in which they were inserted in Bunshin.

This way of persistence is very useful for applications that they want to accede directly to the files. We have for example the case of **Snap**, that it is a decentralised portal constructed upon a p2p structured network, that gives to support to Web applications, where one of them, a collaboration space uses Bunshin like persistence service of the Web server and where it inserts documents, which interests to him that they are accessible by means of a Web client.

3.4. Multicontext and Multifield

Bunshin is not a conventional DHT, in the key/value sense of structuring, but in addition it provides, optionally, the possibility of treating the data as if they were a hash table too, that is to say, that stops a key we can have so many values as we want, identified with a subkey that we called field.

The other possibility, at a higher level of abstraction, is to be able to see a node not only like the container of one bucket, but of so many as we need. Then for us bucket this within a context, and thus in case the different application needed to have buckets will be forced to instantiate a Bunshin application for each one of them, but that exclusively it will have to indicate in the context where it wants to work.

In summary, we arrange if N buckets with values of M fields are necessary, depending on the necessities of the applications.

4. Search engine

Since already we have commented in the previous section, the architecture of Bunshin provides a superior layer where are the services to part of the basic ones of DHT. BunshinSearch is constructed upon Bunshin and uses the explained services previously to provide his:

- Services by keywords
- Connections between keys
- Notification of connections

Next we will detail each one of these services and the interface that each one provides.

For the services by **keywords** indexing by means of tables of distributed inverted indices is used. The interface that provides east service is similar to the basic interface of a DHT but introducing the concept of key word. The method *insert(keywords, key, value),* is in charge to insert the key/value and to update the tables of inverted indices corresponding to the keywords. On the other hand, the method query(keywords) is used to recover the keys that are bound at least to one of the keywords of the consultation. As a result, all the keys will be given back to us that fulfil the condition, ordered by the number of coincidences of keywords. Finally we have the method *remove(keywords, id),* that eliminates the key/value and in addition it modifies the tables of indices of each keyword.

Aside from PlanetDR, we have other implemented applications that make use exclusive of this service, as the case of our decentralised search application. This application makes periodic verifications in search of pages Web and its contents, inserting the meta-information generated in Bunshin. The criterion of selection of keywords is similar to the one of a conventional search application, selecting the keywords between the repeated words more in the document or also between most significant of the metadata, if it has of labels with meta-information.

Thanks to the multifield values we can diversify the key/value concept so that it allows us to offer the **link service**. This service adds two extra fields to a concrete key. These two new fields have as value the lists of the incoming links and outgoing links for their corresponding key.

In order to add the links we have the method *addLinks (key, links)* and *removeLinks(key, links)* to eliminate them, automatically process the both types of links (incoming or outgoing). To query links we have the methods *getIncomingLinks(key)* and g*etOutgoingLinks(key).*

Following with the previous example of the search application, also from the reading of the anchor of the Web, BunshinSearch can connect the keys of the inserted metadata. At the time of returning results thanks to be able to consult the information of hyperlinks of each Web, it will be able in addition to modify adding them the criterion of number of hyperlinks or important hyperlinks, making a considered calculation previously similar to the PageRank from Google[8].

Finally and complementing the previous service, we have the **link notification service**. This service offers to the applications the possibility of being notified when a key is connected. As a result, first the notification request is made on key A, and next, when adding the link to the key B towards the key A, is made the notification corresponding to this event.

Following the model of standard delegation of Java, the methods of this service are, *setLinkListener(id, listener)* to assign to a notifier and *removeLinkListener(id, listener)* to eliminate it. Such fact allows us to be informed on the entrance connections that are added to a key in individual.

This service is useful for the applications where it interests to us to have the news on connections to concrete values, like for example an application similar to OverCite[9], an application based on DHT where the focus is centred in the publication of scientific articles.

5. Validation

Bunshin is a quite mature system since it has been proven in multitude of applications and we have made as much simulations as experimental tests in PlanetLab[10].

In order to validate the reliability that provides the system of replication of Bunshin, we have simulated the creation of a key/value network of 1000 nodes and inserted 1000 pairs randomly, as much the key as the node from where we inserted. For this simulation we have used our own PlanetSim[11] simulator using a Chord overlay network. For each key/value six replicas are made (including the main copy in the successive node).

After insertions, we cause that a fraction of nodes fails and falls of the network, without possibility of warning nor of making the movement of keys. We can see in figure 4 that these results are very similar to those of CFS[12], validating our approach.

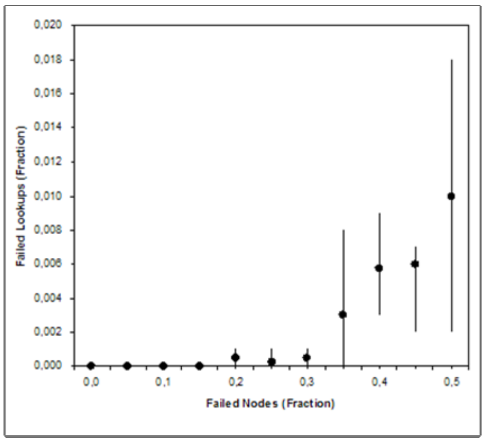


Figure 4. Percentage of requests no satisfied as opposed to the percentage of fallen nodes. Indicating the error bar the minimum and the maximum, and marking the average, of the 5 made tests.

We can observe that they do not begin to have failures in the consultations until 20% of the nodes have fallen and also that the failures are very few until the fallen nodes do not ascend to 35%. The failures begin to be considerable when the six made copies are lost. For example, if 50% of the nodes fall, the possibility of losing a key/value completely is very low, 0,56 = 0,0016, seemed to the results obtained in the test, the average of which, it is of 0.010.

6. Related work

CFS is a system of cooperative file constructed upon the network overlay Chord. It offers the basic services of a DHT, and maintains replicas and caches. CFS this oriented to the storage of blocks and upon is constructed a conventional system of files similar to the UNIX system. Each block is kept in multiple focused Chord nodes that this in storing blocks of data. It does not offer other services like could be a search engine, although if that uses a system of authentication of public key.

PAST follows a philosophy similar to CFS, but in the Pastry overlay. It tries to improve the system of CFS in storage of files and caching, helping itself of the services that Pastry overlay offers, as they are a greater efficiency and proximity of the keys. It offers a system of certificate files for the basic operations. It does not offer another type of services either.

Main the difference with CFS and PAST is that Bunshin so is not focused in offering a service of distributed file systems, but to offer more variety of services than they are interesting at the time of using a DHT from the point of view of the application programmer.

7. Future work and conclusions

In this article we have presented Bunshin, which gives an approach different from the conventional DHT, with the characteristics of an active system of replication and adaptive system of caching, equipping it with more flexibility, as they are the characteristics of multicontext and multifield, and with a service of searches of keywords, of connected of keys, and notification of new connections.

Aside from the services offered by Bunshin that extend the classic interface of the DHTs, it is possible to be extended or also to be extended with other future functionalities. An example could be that for the service of notification of connections if were complemented with a service of propagation of events, like for example Scribe[13], could cause that a single event of this service propagated efficiently in a group of peers interested.

Another remark is the security service that Bunshin offers at the time of conducting the basic operations. This service is suitable to as much equip with privacy the pairs keys/values in the insertions, as in the modifications or the consultations of these. At the moment, we used a system of verification with symmetrical key but shortly one will begin has to use a scheme of asymmetric key stops to encrypt and to decrypt these values with the public and private keys of the users.

In addition we have made, and we continue to make, numerous tests in PlanetLab to correctly purify and to prove all the functionalities and to thus support its good operation. Let us think that Bunshin provides interesting and novel services to the decentralised application programmer p2p.

Acknowledges

Financed for the Ministry of Science and Technology of Spain, TIC2003-09288-C02-00

References

[1] Stoica I., Morris R., Karger D., Kassshoek M.F. y Balakrishanan H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. ACM SIGCOMM, pp. 149-160. Agosto 2001.

[2] Manku G.S., Bawa M. y Raghavan P. Symphony: Distributed Hashing in a Small World. USENIX Symposium on Internet Technologies and Systems (USITS). Marzo 2003.

[3] Rowstron A. y Druschel P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pp 329-350. Noviembre 2001.

[4] Maymounkov P. y Mazières D. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. International Workshop on Peer-to-Peer Systems (IPTPS), pp 53-65. Marzo 2002.

[5] Pairot .C, García P., Rallo R., Blat J. y Skarmeta A.F. The Planet Project: Collaborative Educational Content Repositories on Structured Peer-to-Peer Grids. ACM/IEEE CCGrid 2005. Second International Workshop on Collaborative and Learning Applications of Grid Technology and Grid Education (CLAG). Mayo 2005.

[6] Pairot C., García P. y Skarmeta A.F. Dermi: A New Distributed Hash Table-based Middleware Framework. IEEE Internet Computing. Vol 8, No. 3, pp. 74-84. Mayo/Junio 2004.

[7] Rowstron A. y Druschel P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. ACM Symposium on Operating Systems Principles (SOSP). Octubre 2001.

[8] Brin S. y Page L. The anatomy of a large-scale hypertextual web search engine. International World Wide Web Conference (WWW). Abril 1998.

[9] Stribling J., Councill I. Li J., Kaashoek M.F., Karger D.R., Morris R. y Shenker S. OverCite: A Cooperative Digital Research Library. International Workshop on Peer-to-Peer Systems (IPTPS). Febrary 2005.

[10] PlanetLab Website, http://www.planet-lab.org/

[11] García P., Pairot C., Mondéjar R., Pujol J., Tejedor H. y Rallo R. PlanetSim: A New Overlay Network Simulation Framework. Lecture Notes in Computer Science (LNCS), Software Engineering and Middleware (SEM).Vol. 3437, pp. 123-137. Marzo 2005.

[12] Dabek F., Kaashoek M. F., Karger D., Morris R. y Stoica I. Wide-area cooperative storage with CFS. Proceedings ACM Symposium on Operating Systems Principles (SOSP). Octubre 2001.

[13] Castro M., Druschel P., Kermarrec A.M. y Rowstron A. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. IEEE Journal on Selected Areas in Communication (JSAC), Vol. 20, No 8. Octubre 2002.

[14] Bunshin Website, http://ants.etse.urv.es/bunshin