Providing KBR Service for Multiple Applications

Pengfei Di and Kendy Kutzner
System Architecture Group,
Universität Karlsruhe (TH), Germany
{di|kutzner}@ira.uka.de

Thomas Fuhrmann
Computer Science Department,
TU Munich, Germany
fuhrmann@net.in.tum.de

Abstract

Key based routing (KBR) enables peer-to-peer applications to create and use distributed services. KBR is more flexible than distributed hash tables (DHT). However, the broader the application area, the more important become performance issues for a KBR service.

In this paper, we present a novel approach to provide a generic KBR service. Its key idea is to use a predictable address assignment scheme. This scheme allows peers to calculate the overlay address of the node that is responsible for a given key and application ID. A public DHT service such as OpenDHT can then resolve this overlay address to the transport address of the respective peer.

We compare our solution to alternative proposals such as ReDiR and Diminished Chord. We conclude that our solution has a better worst case complexity for some important KBR operations and the required state. In particular, unlike ReDiR, our solution can guarantee a low latency for KBR route operations.

1. Introduction

Over the last years, peer-to-peer (P2P) overlay networks have become more and more popular. At the beginning, file-sharing was the only major application of that new paradigm. Meanwhile, the P2P application landscape evolved into a rich set of various applications in multiple fields, such as games, content distribution and Internet telephony. Dabek et al. [1] found that many seemingly different P2P applications such as, e. g. , Distributed Hash Tables (DHT), Application Layer Multicast (ALM), and Decentralized Object Location and Routing (DOLR), can be built upon a common API, the *Key Based Routing* (KBR) interface. Recently, this KBR interface has been implemented as a network layer routing protocol [2] [3], too.

The basic KBR API primitive is `route(key, value)`, where the application dependent `key` is hashed into a virtual address space that is common to all applications using the same KBR system. Every node handles the received `value` in some manner, depending on the application type. These *endpoint operations* may be more complex than just the storage and retrieval of objects which are offered by a DHT. For example, a distributed database may evaluate a non-trivial query, an online game may perform a non-trivial update of its distributed state. So, generic DHTs such as OpenDHT [4] cannot directly support all potential KBR applications.

In this paper, we address the question: *How can we efficiently provide a generic KBR service to a potentially large set of applications?*

To this end, we first briefly discuss two naive solutions: (1) all applications straight forwardly share one KBR overlay, and (2) each application builds its own KBR overlay. We argue that the first solution is impractical because nodes and applications might be too heterogeneous. For example, a bandwidth constrained node could still run a small bandwidth KBR application. But it should not be bothered with high volume traffic. This could be the case when all applications naively used one KBR overlay. Furthermore, we argue that the second solution is inefficient in terms of bootstrapping and maintenance effort.

We then propose a new KBR overlay system that combines a basic DHT overlay with application specific KBR overlays. We show that each of these KBR overlays has a significantly lower state and maintenance complexity than a full-fledged KBR overlay.

This paper is structured as follows: We first state the problem this paper deals with in section 2. In section 3 we briefly review related work. Then, section 4 gives an overview over our design considerations. We present our solution in section 5 and compare it to other solutions in section 6. In section 7 we conclude with an outlook to future work.

2. Problem Statement

As said above, many applications can benefit from a KBR service. For the purpose of this paper, we assume that such a service shall forward any message of an application or service type A to that node that runs an instance of A and whose ID is closest to the message destination.

This requirement can be met most easily if each application builds up its own independent KBR overlay. Obviously, here, the required amount of state and communication effort scales linearly with the number of applications. Especially, the need to run the message intensive stabilization protocols independently for each application would lead to a significant waste of bandwidth.

Another simple solution would be one global KBR overlay where each application instance joins with its own identifier. For example, a node N with applications A_i could join with the concatenated IDs $(A_i : N)$. This means that the different applications populate different regions in the KBR address space. This is impractical for many purposes. Furthermore, each application instance would again need to stabilize on its own. In case of the concatenated IDs $(N : A_i)$ all applications of one node are virtual neighbors in the KBR overlay. Thus they can share their stabilization overhead. However, each node has to store and maintain additional state for each application that it does *not* run so that it can forward these messages to the node closest in the ID space that *does* run that application.

In this paper we discuss how a KBR overlay can efficiently provide seemingly the same service as independent per-application KBR overlays would do. We present two insights about KBR overlays whose combination yields an efficient solution to this problem. Of course, other authors have already addressed similar overlay designs. We will briefly review the relevant of them in section 3. We believe that our proposal has significant advantages over all of them.

3. Related Work

Several authors have already discussed how to provide KBR services for multiple applications. Depending on the authors the different application domains are called "namespace" [4], "subgroup" [5], or "cluster" [6].

In their work on OpenDHT [4] Rhea et al. also proposed ReDiR, a recursive distributed rendezvous scheme. It uses a shared DHT to construct a per-application (per-namespace) KBR system. ReDiR divides the DHT address space in *partitions* at different *levels*. Each partition in each level contains one rendezvous point (RP) for one application. A partition of level i covers b sub-partitions of level $i + 1$, where b is often set to 2. At the finest level, a RP stores the addresses of all the nodes in this partition that run the respective application. RPs in coarser levels just store the extremal node addresses for each of their sub-partitions.

For routing a lookup message for a given key in a given namespace, ReDiR first computes the RP at the finest partition level for the message destination. A node query message is sent to this RP. If there is no hit, ReDiR uses the RPs in coarser levels to recursively look up the node that the requested namespace belongs to.

Karger et al. [5] proposed *Diminished Chord* for the formation of subgroups in a Chord overlay. Each subgroup corresponds to a binary search tree that is embedded into the Chord ring. These trees are rooted at the hashed name of the subgroup. When a node joins a subgroup, it registers with one other node in the Chord ring. Note that this node does not necessarily belong to the same subgroup. But there is a well defined process to retrieve the stored information from the embedded tree. It is guaranteed that a node stores information about only up to one other node for each subgroup that exists in the entire system.

For routing a key within a subgroup, it is first routed according to the regular Chord rules. Then the binary search tree is traversed to find the actual destination node. This requires the Chord ring to be equipped with additional *pre-fingers*. Both steps take $O(\log N)$ overlay hops on average.

Ganesan et al. [7] proposed "Canon", a general technique to construct hierarchical DHTs. In Canon, each subdomain has its individual overlay. Nodes in one subdomain merge into a larger overlay network by adding links to nodes in the other DHT domains. Cyclone [6] uses an idea similar to Canon to merge individual Chord rings into one hierarchy. Unlike Canon, Cyclone uses a *suffix* which is appended to the node ID. This results in a more uniform distribution of the Cyclone nodes in the Chord ring. But as described in section 2 it requires additional state at the nodes.

Joung et al. [8] suggest a two tier Chord to reduce the overhead required by stabilizing the Chord ring. Stable nodes become super peers in the first tier Chord ring. All other nodes form the second tier Chord ring.

A binary search tree that is embedded in the second tier Chord ring enables each node to find its closest super node. Thereby, the complexity of join/leave operations is reduced from $O(\log^2 N)$ to $O(\log N)$.

4. Overview and Design Considerations

Our proposal is based on two observations:

1. In a KBR overlay such as Chord, each node stores $O(\log N)$ fingers where N is the total number of nodes. Thus, in total, there are $N \cdot O(\log N)$ fingers stored distributedly in the entire system. Each finger is a tuple that maps a KBR address I to a transport endpoint T in the underlying network, e. g. an IP address and port number. Thus, a KBR overlay is a distributed implementation of that mapping $I \rightarrow T$.

In fact, this implementation is quite redundant: In an overlay with N uniquely identified nodes there exist actually only N different fingers altogether. This redundancy is costly, not only in terms of space but more importantly in terms of maintenance overhead.

2. Assume an idealized Chord ring with address space $[0, 1[$. Assume further that the nodes join sequentially at 0, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{3}{4}$, $\frac{1}{8}$, $\frac{3}{8}$, ... Then, given the own address and the predecessor's address, a node can (almost exactly) calculate which node is responsible for any given KBR address. Here, 'almost exactly' means that in up to 50% of the cases the calculation may erroneously yield the responsible node's direct virtual neighbor. Summarizing this insight we can say that if the nodes join the KBR overlay in such a predictable pattern, we have a mapping $I \rightarrow I_{1,2}$ where either I_1 or I_2 is known to be responsible for I.

From these two observations we can construct an efficient KBR overlay for multiple applications. To see this, let us start with our design goal:

If it was not for the efficiency, we would like each application to have their own KBR overlay. In other words, we want to perform the lookup $(I, A) \rightarrow T$ where T denotes the transport address of the node running application A and whose ID is closest to I. Assume that the nodes that run an instance of application A are assigned their application specific identifiers I_A in a predictable pattern (as described above). Then we know that there is an application specific mapping $(I, A) \rightarrow \{I_1, I_2\}$ where either I_1 or I_2 are responsible for I with respect to application A. A global KBR overlay can then map $(I_{1,2}, A) \rightarrow T_{1,2}$. Note that this global KBR overlay needs only $O(\log N)$ state per node and can perform this lookup in $O(\log N)$ steps.

We elaborate on this idea and present an in-depth analysis of its scaling behavior in the following section.

5. A Light-weight Application Overlay

In order to accommodate the different application types we extend the KBR API to `route({key, app-id}, value)`. Upon joining the KBR overlay for an application A, the node obtains a KBR address I_A for that application. Thus, in general, the node has different IDs for its different applications. But unlike with the naive approach, the node does not need to join a KBR overlay for each application. It only needs to register with a DHT which stores the mapping of these application keys to transport addresses in the underlying network. Similar to ideas from Joung et al. [8] this DHT could be formed of the rather powerful and stable nodes, only. For the purpose of this paper, we propose to use OpenDHT here.

5.1. Addressing Scheme

Unlike many other KBR overlays where nodes are assigned random overlay addresses, here, the application overlay addresses are assigned sequentially according to the following pattern:

$$\begin{cases} addr_0 &= 0 \\ addr_i &= \frac{2^m}{2^k} \cdot (i - 2^k + \frac{1}{2}) \quad \text{for} \quad i \geq 1 \end{cases} \tag{1}$$

where m denotes the size of the application address space and $k = \lceil \log_2 i \rceil$ denotes the division depth of the application ring.

In fig. 1 the application ring can accommodate up to 16 nodes ($m = 4$). In the figure, 10 nodes have already been assigned. The allocation sequence is: {0, 8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15}. This is a pre-order walk through a binary tree (cf. fig. 2).

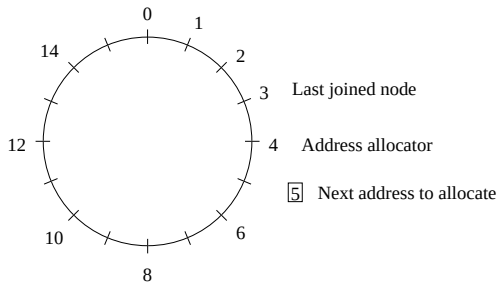

Fig. 1. Application overlay ring with 10 application instances already assigned

This addressing scheme distributes the application addresses uniformly in the address space. If there are

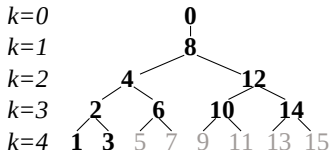

Fig. 2. Address allocation tree

exactly 2^k application instances in the system, this distribution is strictly uniform. In any case, we have the following lemma:

Lemma 1. *Let the* address density *at a given node be the average multiplicative inverse of the distance between that node and its neighbors in the application overlay ring. The ratio of all address densities in the ring will always be less than or equal to 2.*

We omit the formal proof because of the limited space. Just consider the example in fig. 1. There exist two address densities. For some nodes the density is $\frac{1}{2}$ (e. g. node 10), for some it is 1 (e. g. node 1). For node 0 and node 4 the density is $\frac{3}{4}$. This special property of node 0 and 4 (in this example) is used for the address allocation process.

5.2. Address Allocation

The particular sequential order of the address assignment requires coordination. Therefore, we assign the node that has non-equal distances to its neighbors in the address ring the role of an *address allocator*. If the ring contains exactly 2^k nodes, node 0 will assume that role. If the sequential order has been temporarily disturbed because nodes have left the ring, multiple nodes have non-equal distances to their neighbors, and thus multiple address allocators exist.

There are two ways to bring a joining node together with the address allocator: 1. The address allocator registers its transport address with the DHT using the hashed application ID as key. A newly joining node queries the DHT for the current address allocator and sends it a join request. 2. Newly arriving nodes register with the DHT, and the address allocator regularly polls the DHT.

In both cases, when the allocator receives a join request or when it retrieved a joining node's registration, the address allocator assigns that node its address and hands the allocator role over to the next node in the ring. The joining node registers its transport address with the DHT using the hash of the (application ID, node ID) tuple. It does not need to perform any further registration because the new node can calculate the addresses of its neighbors in the application ring from the allocator address and its assigned address:

Lemma 2. *Let $addr_{alloc}$ be the address allocator and $addr_{join}$ the address that it assigns to a joining node. Then $\delta = addr_{join} - addr_{alloc}$ is the distance between the joining node's address and that of its neighbors.*

The first mechanism is well suited for application overlays where nodes join only rarely. If the join frequency becomes too high, the DHT node that is responsible for the respective application ID might become overloaded.

With the second mechanism, there is a clear trade-off between the DHT traffic and the delay for joining the application overlay. If the polling interval is too large, a joining node will have to wait quite long before it can actually join the application overlay. If the interval is too small, the DHT is bothered with too much query traffic. The lower the join frequency in the respective application overlay, the more severe becomes this trade-off.

Thus, we recommend using the one or the other mechanism depending on the join frequency of the respective application.

With both mechanisms, we can exploit the DHT's aggregation properties and thereby further reduce the load in the DHT. Assume that a registration in the DHT is only slowly propagated to the root of the respective aggregation tree, i.e. the node in the DHT that is responsible for the respective application. In that case, the match between a joining node and the address allocator can be made early, in the sense that a request is answered before it has reached the root. If the DHT exploits proximity route selection, this match is likely to happen in the vicinity of the joining node.

The address allocator role is sequentially assumed by various nodes across the network. During this course the allocators pick up the joining nodes from their vicinity, thereby creating additional locality in the application ring.

5.3. Maintenance

In order to maintain the particular order of the address assignments in the application ring, the nodes need to regularly probe their neighbors and their parents in the address tree. If an inner node in the address tree detects the absence of its successor, it implicitly becomes address allocator for the respective address (again). If a node detects the absence of its parent, it registers its own transport address under its parent's address. That means, it (temporarily) assumes more

than one address in the application overlay ring to fill the gap. For a given time span that node becomes address allocator, and it will assign its original address to the joining node. If no joining node is found within this time, it will release its original address. As a consequence, the stable nodes tend to move up in the address tree. This reduces the probability of churn for the inner nodes of the tree.

In order to keep the address tree balanced even under heavy churn, the nodes regularly report the number of their children to their parents. When a node thereby detects that its two subtrees have become unbalanced, it notifies (some of) the leaves of the overpopulated subtree to additionally assume addresses in the underpopulated subtree. Similarly to the above, the resulting dual address assignments will be resolved the next time such a node becomes address allocator. If it does not become address allocator within a given time span, it may release one of its addresses, namely that address that belongs to the lower level in the address tree.

5.4. Routing

For each of its application instances, a node must only store the respective application overlay address and its current address density. From that it can calculate the addresses of its successor and predecessor in the application overlay ring. Furthermore, it can calculate the addresses of many other nodes in the application overlay ring.

Lemma 3. *Supposing there are N nodes in the application overlay ring, a node can calculate the application overlay addresses of at least $\frac{N}{2}$ other nodes in that overlay.*

For example, in fig. 1, node 10 knows the tree height propagated from the tree root. It knows which addresses have already been assigned in this application overlay — namely all the addresses allocated in the first 4 levels of the allocation tree. These are addresses $\{0, 8, 4, 12, 2, 6, 10, 14\}$. The node cannot decide whether the addresses $\{1, 3, 5, 7, 13, 15\}$ have been assigned yet. It can however exclude the existence of any other nodes.

In fig. 2 we see that this ignorance about the existence of some nodes is confined to the lowest level of the address allocation tree. Hence, each node can calculate more than half of the allocated addresses with the help of its own address and its height of allocation tree. (The formal proof is given in the long version of this paper.)

When routing a given key, the sending node calculates the application address that it knows to be closest to the key. With respect to the uncertain addresses ($\{1, 3, 5, ..., 15\}$ in the example above), it presumes their existence.

Lemma 4. *In 25% of the cases, the calculated node does not exist (assuming equal key distribution). The position of this node is exactly between two existing nodes.*

Again, we omit the formal proof here and only briefly sketch the argument. Consider fig. 2. In the worst case, only 50% of the existing nodes are known (cf. lemma 3). In the best case, all existing nodes are known. Assuming equal distribution we thus know on average 75% of the existing nodes. Thus, in 25% of the cases the calculated node does not exist. From the construction of the tree in fig. 2 follows that these nodes lie exactly between two existing nodes.

The closest node is responsible for that unoccupied position in the overlay ring, i.e., the predecessor of the unoccupied position. In order to fill this gap, that node registers its transport address with the respective address, too. As a result a lookup will always yield the correct transport address. This resolution of overlay addresses to transport addresses in the underlying network will be detailed in following section.

5.5. Address Lookup

As said above, each node stores its transport address in OpenDHT upon joining an application overlay. Thus, when routing a key in an application overlay, the nodes need to look up the transport addresses in OpenDHT. The lookup complexity depends on that DHT. For OpenDHT it is logarithmical. In practice, we can expect this lookup to be fast, because OpenDHT adopts proximity neighbor selection (PNS) [9]. From a study by Gummadi et al. [10] we know that the overhead is typically only about 30% as compared with the direct path in the underlying network.

Note that OpenDHT is just one means to implement the mapping of application to transport addresses. This choice is beneficial because OpenDHT is well established and runs on comparably stable nodes. Nevertheless, any DHT would suffice for our proposed KBR service. Such a DHT could be implemented on top of a basic KBR service that some or all nodes provide when they join for their first application. This KBR service could be implemented as overlay or on the network layer.

	ReDiR	Isolated Overlays	DimChord	Our Proposal
App. Inst. Join	$O(\log N \cdot \log M^*)$	$O(\log^2 N)$	$O(\log M)$	$O(\log M^*)$
State per App. Inst.	$O(1)$	$O(\log N)$	$O(\log M)$	$O(1)$
Load Ratio	$O(1)$	$O(1)$	$O(\log M)$	$O(1)$
Routing (worst case)	$O(\log N \cdot \log M^*)$	$O(\log N)$	$O(\log M^* + \log M)$	$O(\log M^*)$

TABLE I. Complexity Comparison

6. Comparison

In this section we compare our proposal to ReDiR, Diminished Chord, and the naive 'one isolated ring per application' approach. Let M be the overall number of nodes, M^* be the number of nodes in OpenDHT, and N the average number of instances per application.

Table I gives the complexity classes for several operations and states. Their derivation can be found in [4] [5] and sec. 5. The complexity for the isolated KBR rings follows directly from the complexities in e. g. Chord.

Note that by distinguishing M and M^* we also incorporate the effect of proximity neighbor and route selection (PNS/PRS) in DHTs. Even though an operation in OpenDHT takes $O(\log M^*)$ overlay hops, these hops have greatly differing actual cost and latency. As a result, the complexity of OpenDHT operations scales sub-logarithmically in practice.

As we see in table I, our proposal has several advantages. Most notably, our proposal requires only one lookup in OpenDHT to route a message for a given key and application, whereas ReDiR may require up to $O(\log N)$ lookups in OpenDHT. This is due to the fact that the predictable overlay address assignment enables us to calculate the responsible node's overlay address directly. We then only need to resolve its transport address via one OpenDHT lookup.

In ReDiR, the level predictor guesses the right rendezvous point. If it fails or if the requested key happens to fall into the subsequent partition, more lookups are required – in the worst case up to $O(\log N)$ lookups.

7. Conclusion and Outlook

In this paper, we have presented a novel solution to build application specific KBR overlays. Our solution is light-weight because it requires only $O(1)$ state per application instance. Similarly to ReDiR, this state is stored in OpenDHT so that applications can benefit from the stability and proximity awareness of that DHT. Unlike ReDiR, we do not need to look up the destination address at a rendezvous point, but can calculate it directly. This is achieved by a predictable overlay address assignment scheme. As a result, we achieve a worst case performance of only one OpenDHT lookup. We expect that this will enable latency critical applications to benefit from KBR overlays, too.

Currently, we are implementing this approach to study its performance both in simulations and on planet lab. We are also exploring the effect of concurrently joining nodes and the possibility to combine our solution with network layer KBR systems such as VRR.

References

[1] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," in *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, 2003.

[2] T. Fuhrmann, "A Self-Organizing Routing Scheme for Random Networks," in *Proceedings of the 4th IFIP-TC6 Networking Conference*, Waterloo, Canada, May 2–6 2005, pp. 1366–1370.

[3] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron, "Virtual Ring Routing: Network Routing Inspired by DHTs," in *Proc. ACM SIGCOMM '06*, Pisa, Italy, Sept. 2006.

[4] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "OpenDHT: a public DHT service and its uses," in *Proc. of the ACM SIGCOMM '05 Conference*, 2005, pp. 73–84.

[5] D. R. Karger and M. Ruhl, "Diminished Chord: A Protocol for Heterogeneous Subgroup Formation in Peer-to-Peer Networks," in *IPTPS*, 2004, pp. 288–297.

[6] M. S. Artigas, P. G. López, J. P. Ahulló, and A. F. Gómez-Skarmeta, "Cyclone: A Novel Design Schema for Hierarchical DHTs," in *Peer-to-Peer Computing*, 2005, pp. 49–56.

[7] P. Ganesan, K. Gummadi, and H. Garcia-Molina, "Canon in G Major: Designing DHTs with Hierarchical Structure," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS) 2004*, 2004.

[8] Y.-J. Joung and J.-C. Wang, "Chord2: A two-layer Chord for reducing maintenance overhead via heterogeneity," *Comput. Networks*, vol. 51, no. 3, pp. 712–731, 2007.

[9] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker, "Fixing the Embarrassing Slowness of OpenDHT on PlanetLab," in *Proceedings of USENIX WORLDS 2005*, December 2005.

[10] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The Impact of DHT Routing Geometry on Resilience and Proximity," in *Proceedings of the SIGCOMM 2003 conference*. ACM Press, 2003, pp. 381–394.