

Robust Object Replication in a DHT Ring

Jinoh Kim and Eric Chan-Tin

May 4, 2007

Abstract

There have been a lot of studies on distributed storage systems and peer-to-peer file systems based on distributed hash tables (DHTs), in which nodes have global unique identifiers generated by a hash function. Since such distributed systems are built on top of a heterogeneous and unreliable infrastructure, replication techniques are often used to improve object availability. However, typical consecutive replication technique provided by DHTs can encounter replication failures caused by storage imbalance due to some reasons such as storage heterogeneity, various file size distribution, or insert rush to some ID regions. In this project, we develop heuristics to provide robust object replication by exploring more nodes than the traditional replication technique. According to the simulation result conducted on PlanetSim, the suggested heuristics significantly diminish replication failure rate with a little overhead. In addition, we present how to maintain replicas with the suggested heuristics in churn situations which is pretty common in wide-area distributed systems.

1 Introduction

The distributed hash table (DHT) is a base architecture of structured overlays, in which nodes have global unique identifiers and keys and nodes share the same identifier space by using predefined hash functions. By sharing the ID space, the DHT ring facilitates key-based routing in overlay networks, in which messages are forwarded according to key prefix and finally delivered to the corresponding node within at most $O(\log N)$ hops. Recent well-known implementations of the key-based routing are Chord [7], Pastry [6], CAN [5], and Tapestry [8].

Recently a variety of distributed storage systems such as PAST [1], OceanStore [3], and Glacier [2] have been introduced, which are based on DHT ring architectures. Since many distributed systems are built on top

of a heterogeneous and unreliable infrastructure, replication techniques are often used to improve object availability. However, the typical replication technique in the DHT ring, in which the object is replicated to consecutive nodes, can encounter replication failures caused by storage imbalance. For example, some nodes in the replica set might fail during the object insert operation due to insufficient storage to accommodate the object. Thus the system might not be able to meet the proper replication level that the application requires. Furthermore, in some strict systems, such partial failure can be recognized as the failure of an insert operation. Storage imbalance are caused by many reasons in wide-area distributed systems such as heterogeneity of contributed storage, a broad range of file size, insert rush to some specific regions for a while, and so forth.

To address this problem, we developed some heuristics which can lower replication failures by exploring more nodes than traditional *consecutive* approach. In other words, the heuristics make it possible to increase the insertion success rate. Although the heuristics impose some overhead to object insertions, we show the heuristics significantly diminish replication failure rate which might be a weighty metric in some systems.

It is challenging to maintain proper replication level in churn situations, in which nodes frequently join and leave the ring. When a node departs from the system, an object which was stored in that node does not meet the replication level anymore. At join time, it is also necessary to consider the change of neighbor because the newly joined node might be part of the consecutive nodes used as replicas. For this reason, we also developed a technique to handle churn, which is common in distributed systems.

The rest of the paper is organized as follows. Section 2 discusses replication problem. Section 3 presents the heuristics lowering replication failures. Section 4 explains how we handle the churn, and evaluation results are given in section 5. Finally we give a conclusion.

2 Replication Problem

To manipulate the object o in the DHT ring, the corresponding key is hashed to $key(o)$, and there exists a root node $root(o)$ responsible for the hashed key. In the DHT design, the next successor node is $successor(o)$. The object is stored in the root node in insert time, and is retrieved from the root node in lookup time.

For replication, the replication factor is defined as $k > 1$. The $repset(o)$ is defined as the replica nodes for the object o , and its size is the same as

the replication factor k . In the typical DHT design, $repset(o)$ is the node set of closest *successors* to the $key(o)$, and we call it *consecutive* replication. In consecutive replication, o is replicated into $repset(o)$, each of which will later respond to lookups for the object.

The consecutive replication simply assumes sufficient storage in $repset(o)$ nodes for object replication. In reality, this might not be the case; that is, some nodes of $repset(o)$ might not have sufficient storage. Hence, the consecutive replication can suffer from replication failure. Although this replication failure is inevitable because of finite storage, it is possible to lower the failure rate in object replication. We discuss how to lower this replication failure in the next section.

Churn threatens maintaining replication level. When a node in $repset(o)$ leaves the ring, the $root(o)$ tries to replicate its objects to the next successor of the departing node to maintain the defined replication level. On the other hand, when a node joins the ring, the node should have responsibility to the objects with keys relevant to the node. Thus object migration is necessary. In addition, the farthest node in the $repset(o)$ may be evicted from the $repset(o)$, if necessary.

3 Heuristics

The traditional consecutive replication is sometimes improper due to the possibility of storage imbalance. This section presents some heuristics: *random-fit*, *first-fit*, *best-fit*, and *worst-fit*, which help to lower replication failure rate by exploring more nodes than the consecutive approach. Exploring more nodes than the replication factor k raises chances to find nodes which can accommodate the object. To explore more nodes, we define followings:

- f : the number of candidate nodes ($f \geq k$)
- $candset(o)$: a set of candidate nodes to accommodate the object - the size of the set is equal to f

The factor f is a configurable variable to expand the candidate nodes $candset(o)$. Of $candset(o)$, proper nodes are selected as $repset(o)$. Therefore, if f is large, the chances for replication success will increase. However overhead will also increase. The suggested heuristics used are described below:

- *random-fit*, in which the object o is replicated into k randomly chosen nodes with sufficient storage among $candset(o)$
- *first-fit*, in which the object o is replicated into first k successor nodes with sufficient storage among $candset(o)$
- *best-fit*, in which the object o is replicated into k nodes which have smallest storage that is adequate among $candset(o)$
- *worst-fit*, in which the object o is replicated into k nodes which have largest storage among $candset(o)$

Intuitively worst-fit is useful to balance the available storage by assigning objects to larger storage nodes. On the other hand, best-fit is likely to accelerate storage imbalance because tiny storage nodes are drained first. However, by leaving large storage, fairly big files can have more chances to be accommodated. Besides, first-fit induces relatively low overhead because walking through the ring requires half the overhead of the other heuristics on average, while all the others require f "walking".

Figure 1 shows how each heuristic accommodates an object. Random-fit relies on randomness to select replica nodes. First-fit walks through the ring and accommodate the object if current node has sufficient storage for the object. In best-fit, it traverses all candidate nodes, and then selects k nodes with smallest storage that is adequate to store the object. Worst-fit selects nodes with largest storage.

4 Handling Churn

In the typical setting, each root node is responsible to maintain the required replication level. When a node leaves the ring, neighbor nodes check if they replicated objects into the departed node. It's fairly simple because of the consecutiveness of replication. If a new node joins, the neighbor nodes migrate their objects from the successor of the joined node to the new node to keep their replicas consecutive.

In our heuristics, it is a bit more complicated than the traditional one, since the heuristics explore more nodes as possible replicas. Furthermore, sometimes even the root node does not hold the object whose key is mapped to the root node. Hence, we maintain a replica table in each root node (and successor nodes) that keeps which replica nodes store its objects, whose keys should be covered by the root node. This metadata should be added

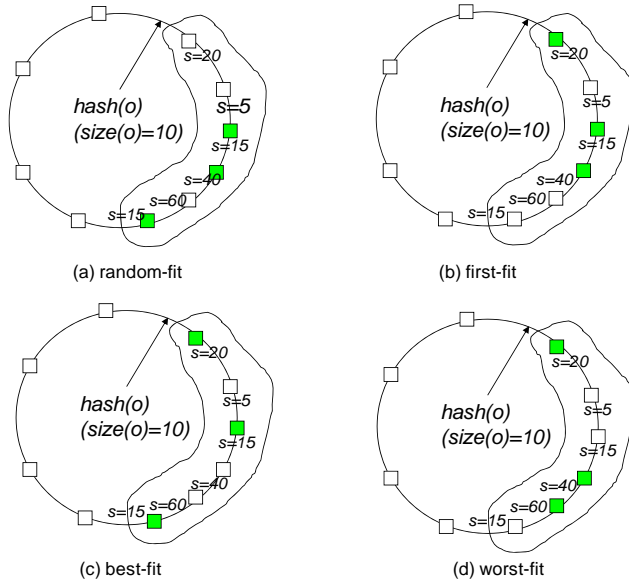


Figure 1: Example object accommodation with suggested heuristics

in replication time. Importantly, the replica table is *soft state*. That is, the table can be restored by scanning successor nodes if necessary.

The metadata stored in each node is not a big overhead, since the size of the metadata is small compared to the node's storage space and the object sizes.

4.1 Node Join

When a node joins, the joining node claims to the neighbors that it is the new root node for the identifier space (*predecessor ID of new node, new node ID*]. The successor of the newly joined node then transfers all entries in its replica table whose keys belong to the ID space that the new node is in charge of.

After receiving the metadata, the new root node tests if there are any replicas pointing to nodes that are further away than the new candidate set. The new root node will then replicate the objects according to the heuristic used.

The predecessors and successors of the new node will also do the same test described above.

4.2 Node Departure

When a node leaves (either normal leaving or due to failure) the ring, all predecessor and successor nodes check their metadata. If any entry points to the departed node, the node searches for another replica node for the object.

There might be a problem to transfer the metadata if more than two successive nodes leave the DHT ring at the same time. In that case, the new node scans successor nodes within the candidate size so as to recover the replica table.

Some overlay networks have specific protocols to deal with churns and these protocols can be used, for example Chord's stabilize protocol.

4.3 Object Lookup

When a node S needs to lookup an object o , and $key(o)$ maps to K , node S will find the node R that is in charge of that ID space where K is mapped to. Once the node R is found, node S can then query R for all the replicas for the object o .

Even with churns (nodes leaving, failing, and/or joining the DHT), at least one replica that holds the object will be found, unless the whole successor list of node R leaves the network. If the DHT ring is well-balanced, this is unlikely to happen.

5 Experimental Results

5.1 Setup

PlanetSim [4], available from Sourceforge.net, is used as the simulator. PlanetSim is implemented completely in Java and has support for various DHT implementations such as Chord and Pastry. However, there is no support for object storage or replication. Besides, PlanetSim boasts the feature of being able to run as an emulator (using normal TCP connections) as well, with no code changes required. The emulator part of PlanetSim was not tested but is regarded as a possible future work of this project.

The missing components critical to this project, such as object storage, object replication, the heuristics, and suitable statistics collection were added to PlanetSim.

5.2 Evaluation

The simulations were performed using the different heuristics mentioned in Section 3 - consecutive, random-fit, first-fit, worst-fit, and best-fit; varying the number of replicas and the number of possible candidates.

A DHT ring with 1,000 nodes was created. The node IDs were generated so that the ID space owned by each node would be well-balanced, that is, each node will be in charge of approximately the same amount of IDs.

The storage space of each node was randomly assigned, following a uniform distribution from 100 MB to 100 GB. The same nodes and storage space were used for all the simulations.

Each object size was exponentially distributed with a mean of 100 MB, because there are likely a lot of small objects (such as text and music files) and only a couple of huge objects (such as movie files). The key that each object would hash to was randomly generated. The workload generated was thus random.

Two statistics were measured as follows:

- Insert Success rate - the percentage of objects that were successfully inserted in the DHT ring.
- Message Overhead - the total number of messages sent, normalized to the consecutive heuristic.

The consecutive heuristic is used as the baseline. It is expected that worst-fit would provide a higher insert success rate since the nodes with the most storage space are chosen as replicas. Best-fit will probably not give as high an insert success rate than the other heuristics since the nodes are almost out of storage space after each insertion. It is expected that first-fit will also provide a higher insert success rate than the baseline. Random-fit should provide an insert success rate comparable to first-fit and worst-fit.

However, it is also expected that the heuristics used (compared to the baseline) will incur a higher overhead in terms of messages sent. For example, in worst-fit, random-fit, and best-fit, all the candidate nodes have to be queried for their available storage space before the replicas are chosen.

5.2.1 Random Workload with 10,000 object insertions

10,000 objects were inserted in the DHT ring to random IDs. The workload used was random.

The insertion success rate for each heuristic with varying number of replicas and candidates is shown in Figure 2. All the developed heuristics give

Random Workload with 10,000 object insertions

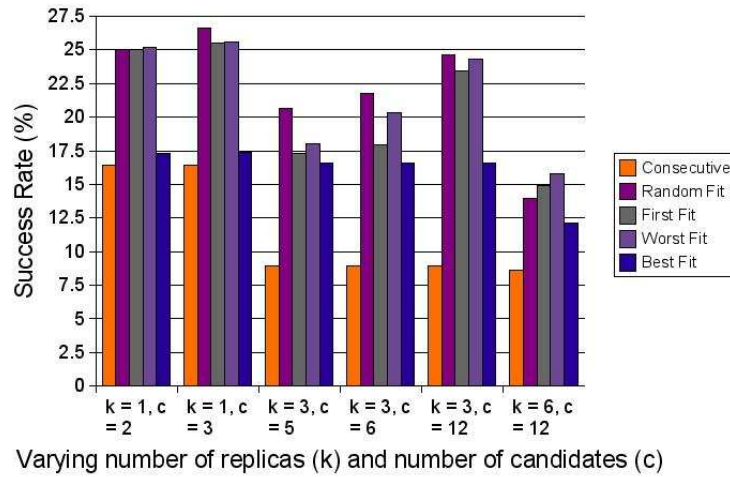


Figure 2: Insert success rate for 10 000 object insertions with a random workload

a much higher insert success rate than the baseline, regardless of the number of replicas and number of candidates used. Surprisingly, the random-fit heuristic performs much better than all the other heuristics.

However, the higher success rate does not for free. Figure 3 shows the total number of messages sent, normalized to the baseline (consecutive heuristic).

As expected, for random-fit, worst-fit, and best-fit heuristics, all the candidate nodes have to be queried, thus the number of messages sent is much higher than the baseline and is dependent on the number of candidates used. The first-fit heuristic shows slightly lower overhead because not all the candidates nodes have to be queried - only those that are needed, but the gaps are not significant unlike what we expect.

5.2.2 Different Workload with different number of object insertions

1,000 object insertions with the similar random workload is also simulated and the results obtained are comparable to the 10,000 object insertions with a random workload.

Moreover, a "popular" workload is generated, where many objects will

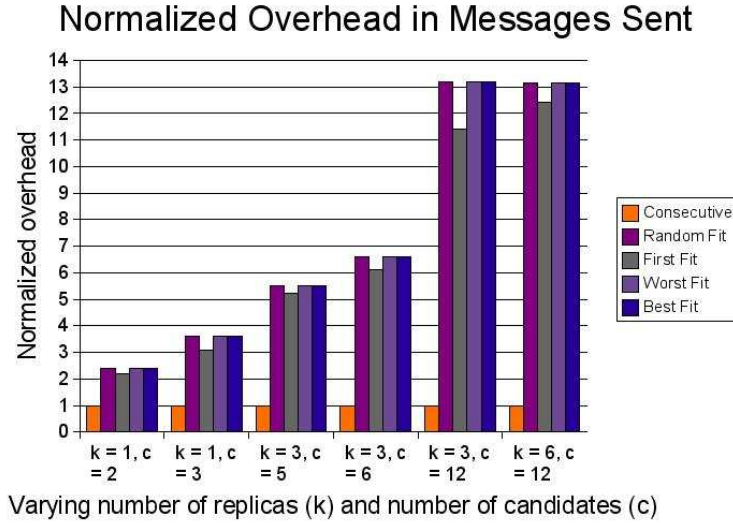


Figure 3: Total number of messages sent, normalized to the consecutive heuristic, for 10 000 object insertions with a random workload

map/hash to the same ID, that is, have the same key. 10,000 object insertions are used and again, the results are comparable to the 10,000 object insertions with a random workload.

The tables for all the simulations performed can be found in the Appendix.

5.3 Churn and Lookup

Lookup operations are not simulated because the performance would be similar to the usual lookup operations used in other DHT implementations, such as Chord. Once the root node is found, all the replicas can be obtained. It is just one extra message sent and there is no overhead for all the heuristics compared to the baseline since for any replica heuristic used, the root node will hold the replica list.

Churn is also not simulated, because as described in Section 4, the Chord stabilize protocol can be used and there is thus, not much overhead.

5.4 Discussion

The higher insert success rate does come with a price - a higher number of messages sent. However each message is small: a query will consist of the sender's ID, IP address, port, the receiver's ID, IP address, port, and a bit indicating this is a query message. The reply message will consist of the same fields plus a field containing the amount of storage space available. With fast Internet, a message can be sent within milliseconds.

The most unexpected result is that the random-fit produces the best results. In fact, our experiments have conducted with limited workloads in the limited setting in terms of storage and file distributions, so there may be some unexplored areas which will be our future study.

The first-fit shows comparable results to the worst-fit and random-fit. Given that the latter two heuristics must traverse all candidate nodes before deciding replica nodes, the first-fit strategy seems more practical as well as strongly competitive in real settings.

6 Conclusion

Storage imbalance might lead to replication failures even though storage in the system is still under utilized. Since replication is often used to increase availability, failing to replicate an object can be critical in some distributed systems. We pay attention to this replication failure in this project. To mitigate the failure, particularly in under-utilized storage state, we developed heuristics to provide robust object replication. The suggested heuristics greatly lower replication failure rate by exploring more nodes than the traditional replication technique. Of heuristics, both random-fit and worst-fit produces the best results, but first-fit also shows comparable results with low overhead, thus implying that it is more practical in real settings. Furthermore, we presented how we handle churn by employing a soft-state replica table (metadata) in each root node.

References

- [1] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, pages 75–80, Schloss Elmau, Germany, May 2001.
- [2] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings*

of the 2ndt *USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

- [3] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. *j-SIGPLAN*, 35(11):190–201, Nov. 2000.
- [4] PlanetSim, <http://planet.urv.es/planetsim/>.
- [5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01*, pages 161–172, 2001.
- [6] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329+, 2001.
- [7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM'01*, pages 149–160, New York, NY, USA, 2001. ACM Press.
- [8] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiawicz. Tapestry: A resilient global-scale overlay for service deployment. In *IEEE Journal on Selected Areas in Communications*, 2003.

Appendix

This appendix contains tables of all the simulations performed.

| | Consecutive | Random Fit | First Fit | Worst Fit | Best Fit |
|-----------------|-------------|------------|-----------|-----------|----------|
| $k = 1, c = 2$ | 16.45 | 25.02 | 25.02 | 25.16 | 17.34 |
| $k = 1, c = 3$ | 16.45 | 26.59 | 25.48 | 25.59 | 17.36 |
| $k = 2, c = 3$ | 16.4 | 20.47 | 16.95 | 17.11 | 16.56 |
| $k = 2, c = 4$ | 16.4 | 23 | 17.34 | 22.06 | 17.34 |
| $k = 3, c = 4$ | 8.92 | 20.67 | 16.55 | 17.1 | 16.55 |
| $k = 3, c = 5$ | 8.92 | 20.63 | 17.33 | 18.02 | 16.58 |
| $k = 3, c = 6$ | 8.92 | 21.8 | 17.98 | 20.3 | 16.61 |
| $k = 3, c = 7$ | 8.92 | 21.06 | 18.12 | 20.4 | 16.61 |
| $k = 3, c = 8$ | 8.92 | 22.92 | 18.95 | 22.63 | 16.62 |
| $k = 3, c = 9$ | 8.92 | 23.01 | 19.04 | 22.62 | 16.62 |
| $k = 3, c = 12$ | 8.92 | 24.63 | 23.42 | 24.32 | 16.61 |
| $k = 4, c = 6$ | 8.91 | 17.9 | 16.57 | 16.88 | 14.91 |
| $k = 4, c = 8$ | 8.91 | 19.51 | 16.83 | 19.12 | 15.49 |
| $k = 6, c = 8$ | 8.66 | 11.28 | 10.26 | 10.92 | 10.26 |
| $k = 6, c = 10$ | 8.66 | 11.5 | 11.37 | 12.19 | 10.3 |
| $k = 6, c = 12$ | 8.66 | 13.94 | 14.93 | 15.75 | 12.12 |

Table 1: Insert Success Percentage for 10,000 object insertions and random workload

| | Consecutive | Random Fit | First Fit | Worst Fit | Best Fit |
|---------------|-------------|------------|-----------|-----------|----------|
| k = 1, c = 2 | 86.4 | 100 | 99.9 | 100 | 93.8 |
| k = 1, c = 3 | 86.4 | 100 | 100 | 100 | 93.8 |
| k = 2, c = 3 | 86.4 | 100 | 90.1 | 90.7 | 86.4 |
| k = 2, c = 4 | 86.4 | 100 | 93.8 | 100 | 93.8 |
| k = 3, c = 4 | 13.2 | 99.9 | 86.4 | 90.7 | 86.4 |
| k = 3, c = 5 | 13.2 | 100 | 93.8 | 99.4 | 86.4 |
| k = 3, c = 6 | 13.2 | 100 | 99.8 | 100 | 86.4 |
| k = 3, c = 7 | 13.2 | 100 | 100 | 100 | 86.4 |
| k = 3, c = 8 | 13.2 | 100 | 100 | 100 | 86.4 |
| k = 3, c = 9 | 13.2 | 100 | 100 | 100 | 86.4 |
| k = 3, c = 12 | 13.2 | 100 | 100 | 100 | 86.4 |
| k = 4, c = 6 | 13.2 | 99.9 | 86.4 | 88.9 | 69.6 |
| k = 4, c = 8 | 13.2 | 100 | 88.7 | 100 | 75.3 |
| k = 6, c = 8 | 13.2 | 34.3 | 23.5 | 29.6 | 23.5 |
| k = 6, c = 10 | 13.2 | 13.2 | 31 | 41.1 | 23.5 |
| k = 6, c = 12 | 13.2 | 62.7 | 69.6 | 77.5 | 41.8 |

Table 2: Insert Success Percentage for 1,000 object insertions and random workload

| | Consecutive | Random Fit | First Fit | Worst Fit | Best Fit |
|---------------|-------------|------------|-----------|-----------|----------|
| k = 1, c = 2 | 15.47 | 24.21 | 23.98 | 24.02 | 16.27 |
| k = 1, c = 3 | 15.47 | 25.04 | 24.41 | 24.63 | 16.27 |
| k = 2, c = 3 | 15.43 | 19.02 | 15.97 | 15.98 | 15.53 |
| k = 2, c = 4 | 15.43 | 21.33 | 16.3 | 21.01 | 16.25 |
| k = 3, c = 4 | 8.07 | 19.43 | 15.52 | 15.97 | 15.52 |
| k = 3, c = 5 | 8.07 | 19.77 | 16.24 | 16.86 | 15.53 |
| k = 3, c = 6 | 8.07 | 20.52 | 16.88 | 19.22 | 15.54 |
| k = 3, c = 7 | 8.07 | 20.66 | 17.08 | 19.27 | 15.54 |
| k = 3, c = 8 | 8.07 | 22.04 | 17.87 | 21.39 | 15.56 |
| k = 3, c = 9 | 8.07 | 22.13 | 17.93 | 21.43 | 15.57 |
| k = 3, c = 12 | 8.07 | 23.27 | 22.49 | 23.28 | 15.56 |
| k = 4, c = 6 | 8.06 | 17.84 | 15.53 | 15.77 | 13.76 |
| k = 4, c = 8 | 8.06 | 18.64 | 15.8 | 18.1 | 14.31 |
| k = 6, c = 8 | 7.88 | 10.32 | 9.37 | 10.04 | 9.33 |
| k = 6, c = 10 | 7.88 | 10.27 | 10.47 | 11.23 | 9.36 |
| k = 6, c = 12 | 7.88 | 13.56 | 13.78 | 14.66 | 11.19 |

Table 3: Insert Success Percentage for 10,000 object insertions and popular workload

| | Consecutive | Random Fit | First Fit | Worst Fit | Best Fit |
|---------------|-------------|------------|-----------|-----------|----------|
| k = 1, c = 2 | 1 | 2.39 | 2.2 | 2.39 | 2.39 |
| k = 1, c = 3 | 1 | 3.59 | 3.09 | 3.59 | 3.59 |
| k = 2, c = 3 | 1 | 3.59 | 3.39 | 3.59 | 3.59 |
| k = 2, c = 4 | 1 | 4.78 | 4.39 | 4.78 | 4.78 |
| k = 3, c = 4 | 1 | 4.39 | 4.29 | 4.39 | 4.39 |
| k = 3, c = 5 | 1 | 5.49 | 5.21 | 5.49 | 5.49 |
| k = 3, c = 6 | 1 | 6.59 | 6.12 | 6.59 | 6.59 |
| k = 3, c = 7 | 1 | 7.69 | 7.02 | 7.69 | 7.69 |
| k = 3, c = 8 | 1 | 8.78 | 7.92 | 8.78 | 8.78 |
| k = 3, c = 9 | 1 | 9.88 | 8.81 | 9.88 | 9.88 |
| k = 3, c = 12 | 1 | 13.18 | 11.42 | 13.18 | 13.18 |
| k = 4, c = 6 | 1 | 6.59 | 6.37 | 6.59 | 6.59 |
| k = 4, c = 8 | 1 | 8.78 | 8.2 | 8.78 | 8.78 |
| k = 6, c = 8 | 1 | 8.76 | 8.56 | 8.76 | 8.76 |
| k = 6, c = 10 | 1 | 10.95 | 10.53 | 10.95 | 10.95 |
| k = 6, c = 12 | 1 | 13.14 | 12.43 | 13.14 | 13.14 |

Table 4: Message overhead, normalized to the baseline, for 10,000 object insertions and random workload

| | Consecutive | Random Fit | First Fit | Worst Fit | Best Fit |
|---------------|-------------|------------|-----------|-----------|----------|
| k = 1, c = 2 | 1 | 14.71 | 8.35 | 14.71 | 14.71 |
| k = 1, c = 3 | 1 | 22.06 | 8.36 | 22.06 | 22.06 |
| k = 2, c = 3 | 1 | 22.06 | 15.71 | 22.06 | 22.06 |
| k = 2, c = 4 | 1 | 29.41 | 16.43 | 29.41 | 29.41 |
| k = 3, c = 4 | 1 | 4.61 | 4.46 | 4.61 | 4.61 |
| k = 3, c = 5 | 1 | 5.76 | 4.61 | 5.76 | 5.76 |
| k = 3, c = 6 | 1 | 6.91 | 4.68 | 6.91 | 6.91 |
| k = 3, c = 7 | 1 | 8.06 | 4.69 | 8.06 | 8.06 |
| k = 3, c = 8 | 1 | 9.22 | 4.69 | 9.22 | 9.22 |
| k = 3, c = 9 | 1 | 10.37 | 4.69 | 10.37 | 10.37 |
| k = 3, c = 12 | 1 | 13.82 | 4.69 | 13.82 | 13.82 |
| k = 4, c = 6 | 1 | 6.91 | 6.44 | 6.91 | 6.91 |
| k = 4, c = 8 | 1 | 9.22 | 6.74 | 9.22 | 9.22 |
| k = 6, c = 8 | 1 | 9.22 | 8.9 | 9.22 | 9.22 |
| k = 6, c = 10 | 1 | 11.52 | 10.65 | 11.52 | 11.52 |
| k = 6, c = 12 | 1 | 13.82 | 11.81 | 13.82 | 13.82 |

Table 5: Message overhead, normalized to the baseline, for 1,000 object insertions and random workload

| | Consecutive | Random Fit | First Fit | Worst Fit | Best Fit |
|---------------|-------------|------------|-----------|-----------|----------|
| k = 1, c = 2 | 1 | 2.37 | 2.18 | 2.37 | 2.37 |
| k = 1, c = 3 | 1 | 3.55 | 3.08 | 3.55 | 3.55 |
| k = 2, c = 3 | 1 | 3.55 | 3.37 | 3.55 | 3.55 |
| k = 2, c = 4 | 1 | 4.73 | 4.36 | 4.73 | 4.73 |
| k = 3, c = 4 | 1 | 4.35 | 4.26 | 4.35 | 4.35 |
| k = 3, c = 5 | 1 | 5.44 | 5.18 | 5.44 | 5.44 |
| k = 3, c = 6 | 1 | 6.53 | 6.09 | 6.53 | 6.53 |
| k = 3, c = 7 | 1 | 7.62 | 7 | 7.62 | 7.62 |
| k = 3, c = 8 | 1 | 8.7 | 7.9 | 8.7 | 8.7 |
| k = 3, c = 9 | 1 | 9.79 | 8.8 | 9.79 | 9.79 |
| k = 3, c = 12 | 1 | 13.05 | 11.42 | 13.05 | 13.05 |
| k = 4, c = 6 | 1 | 6.53 | 6.33 | 6.53 | 6.53 |
| k = 4, c = 8 | 1 | 8.7 | 8.17 | 8.7 | 8.7 |
| k = 6, c = 8 | 1 | 8.69 | 8.51 | 8.69 | 8.69 |
| k = 6, c = 10 | 1 | 10.86 | 10.48 | 10.86 | 10.86 |
| k = 6, c = 12 | 1 | 13.03 | 12.39 | 13.03 | 13.03 |

Table 6: Message overhead, normalized to the baseline, for 10,000 object insertions and popular workload