Efficient Heartbeats and Repair of Softstate in Decentralized Object Location and Routing Systems

Hakim Weatherspoon and John D. Kubiatowicz
Computer Science Division
University of California, Berkeley
{hweather, kubitron}@cs.berkeley.edu

Abstract

Redundancy alone is not sufficient to provide long-term guarantees in distributed systems. Instead, it must be coupled with mechanisms for automatic maintenance. In this paper, we show how Decentralized Object Location and Routing networks (DOLRs) with locality provide a framework for efficient heartbeats and continuous system repair.

1. Introduction

Recent peer-to-peer systems have adopted decentralized object location and routing (DOLR) infrastructures to assist in organizing and manipulating their data. Prominent examples of DOLRs include CAN[6], Chord[8], Pastry[2], Tapestry[3, 11], and other Plaxton, Rajaraman, Richa[5] structures. DOLRs provide sufficient probabilistic routing guarantees to find an object if it exists; but not enough, if any, reliability guarantees. This dilemma is often solved with replication[1, 2] or other forms of redundancy[4, 9]. Unfortunately, redundancy is a short-term mechanism since hardware eventually fails, software has bugs, people make mistakes, and regions of the infrastructure may be destroyed by natural disasters or malicious attacks. Thus, it is essential that systems provide long-term maintenance through automatic *fault detection* and *repair* of faults.

Fault detection and repair are significant challenges in global-scale DOLR-based systems since information is embodied in the aggregate resources of a constantly changing set of unreliable nodes. The sheer size of such systems dictates that each participant will possess enough storage to hold only a small piece of the system. This requires distributed maintenance techniques. Fortunately, DOLRs that provide *locality*[^1] can aid in automatic maintenance by sup-

[^1]: By locality, we mean the ability to utilize local resources over global ones whenever possible.

porting efficient heartbeats to detect faults and trigger repair of the DOLR and resident objects.

In this paper, we survey automatic fault detection and repair techniques for infrastructure state and data. We begin by specifying the properties that we need from the underlying DOLR in Section 2. In Section 3, we briefly describe how location-independent routing works in a DOLR in the context of Tapestry. Next, we survey some DOLR implementations in Section 4. Finally, we enumerate the properties of a DOLR that enable self-repair in Section 5.

2. Requirements

Replicas are distributed widely to enhance durability. This complicates the process of locating them for repair. A closely related problem is the need to direct *queries* to appropriate nodes. Since the *key* for routing to a *value* is named by opaque bit-strings – a *globally-unique identifier* or *GUID*, we need an infrastructure that can perform *location-independent routing* of messages directly to objects using only GUIDs. In addition, the routing layer should exhibit *deterministic location*, objects should be located if they exist anywhere in the network.

Location-independent routing in DOLR's use *softstate* to allow the system to be dynamic. DOLR's maintain softstate with *heartbeats* and/or republish messages (*i.e.* re-add an entry into the distributed directory). Server heartbeats help maintain routing state. Object heartbeats help maintain the distributed directory. Heartbeats assist in detecting failures. When a failure occurs, the system needs to account for the loss and refresh loss redundancy if a *threshold* is reached.

Problem If not careful, the cost for maintaining the routing structure and objects stored in it can easily render the system useless. The *maintenance resource over utilization* problem has three aspects that are reflected in the current literature of DOLR's: server heartbeats that cross the wide area, object heartbeats on a per-object-basis, and use of re-

dundant links for heartbeats and maintenance information. Per-object heartbeats can be considered dangerous if the system scale becomes large enough.

Heartbeat Locality   If the the overlay network (DOLR) is not aware of the underlying network topology, server and object heartbeats will cross the wide area increasing the system's bisection bandwidth utilization. To permit locality optimizations it is important that the routing process exhibit *routing locality*, use as few network hops as possible and that these hops should be as short as possible. That is, routes should have low *stretch*[2], not just a small number of application-level hops.

Per-Object Heartbeats   Each node in a DOLR has the potential to store many objects; that is, more objects than neighbor links. Some of the DOLR's make no provisions to make sure that objects still exist; while, other DOLR's detect object failure with object heartbeats.

Redundant Links   The problem with object heartbeats is that they traverse redundant links. That is, the number of stored objects is much greater than the number of neighbor links. Most of the heartbeats can be aggregated together.

3. Location-Independent Routing

We now describe Tapestry[3, 11], a routing and location system. Tapestry is an IP overlay network that uses a distributed, fault-tolerant architecture to track the location of every object in the network. Tapestry has two components: a *routing mesh* and a *distributed directory service*.

Routing Mesh: Figure 1 shows a portion of Tapestry. Each storage server and client is a Tapestry node with a unique 40-digit hexadecimal address drawn from a random distribution. Tapestry nodes are connected via *neighbor* links of varying levels; these are shown as solid arrows. The level-1 links (L1) from a given node connect to the 16 closest, defined by network latency, nodes with different values in the lowest digit of the address. Level-2 links (L2) connect to the 16 closest nodes that match in the lowest digit and have different second digits, *etc*.

Neighbor links provide a route from every node to every other node. For example, Figure 1 shows a path (thick solid arrows) from node 5230 to node 8954. The routing process resolves the destination one digit at a time: $8***\implies 89**\implies 895*\implies 8954$, where $*$'s represent wildcards. This scheme is based on the hashed-prefix

---
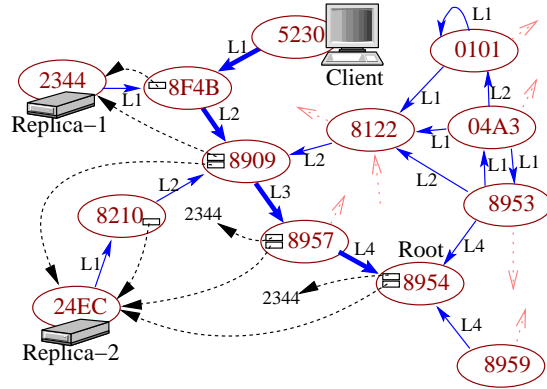
Figure 1. Tapestry Infrastructure: *Nodes are connected to other nodes via neighbor links (solid arrows). Any node can route to any other by resolving one digit at a time: e.g.* $0101 \rightarrow 8122 \rightarrow 8908 \rightarrow 8957 \rightarrow 8954$. *Each GUID is associated with one particular "Root" node (*8954 *in this example). A server* publishes *the location of a replica by sending a message toward the root, leaving back-pointers at each hop (dotted arrows). Clients locate replicas by sending a message toward a root until they encounter enough pointers. Client 5230 can locate two replicas after only two hops:* $5230 \rightarrow 8F4B \rightarrow 8909$.

---

routing structure originally presented by Plaxton, Rajaraman, and Richa [5].

Distributed Directory Service: To perform location-independent routing, Tapestry employs a mechanism that deterministically maps each GUID to a small (5) set of unique *root* nodes. A storage server then *publishes* the fact that it is storing a replica or other object by routing a message toward each of the root nodes (as described above), depositing *location pointers* to the object's location at each hop from the server to the root. Figure 1 shows two replicas with the same GUID stored at different nodes (nodes 2344 and $24EC$). This figure also shows one of the root nodes (8954). The location pointers are shown as dotted arrows that point back to storage servers. Note that each of the root nodes keeps track of the location of every replica that maps to its address, enabling distributed repair (Section 5).

To locate an object, a client sends a message toward one of the object's roots. As soon as the message encounters a pointer with the desired GUID, it routes directly to the object. In the figure, client 5230 can locate two replicas after only two hops: $5230 \rightarrow 8F4B \rightarrow 8909$. In the worst case, this involves routing all the way to the root. However, if the desired object is close to the client, then the path from the client to the root will intersect the path from a storage server to the root with high probability. In fact, it is shown

---

[2] Stretch is the ratio between the distance traveled by a query to an object and the minimal distance from the query origin to the object.

Scheme	Maintain Node	Maintain Object	Detect Node	Detect Object	Repair Object
PRR [5]	-	-	-	-	-
Chord [8]	server hb	-	timeout, msg	-	-
CAN [6]	server hb	-	timeout, msg	-	-
Pastry [7]	server hb	-	timeout, msg	-	-
Tapestry [3, 11]	server hb	obj hb republish	timeout, msg	timeout, msg	-
Tapestry w/ This paper	server hb expn bkf	server hb expn bkf republish notification	timeout msg	timeout msg	threshold

Table 1. *Repair in DOLR systems: Most systems implement server heartbeats (hb) to maintain routing state and detect node failure. Similarly, objects are maintained with object heartbeats or explicit republish messages. Object Failure is detected when a timeout occurs or a republish has not been received. Finally, Lost objects are repaired when a redundancy threshold is reached.*

in [5] that the average distance traveled in locating an object is proportional to the distance from that object[3].

4. System Comparisons

Table 1 compares maintenance techniques for several DOLR networks. All DOLRs can locate objects. However, they differ with respect to locality, *i.e.* not all systems minimize stretch. PRR [5], Tapestry[3, 11], and Pastry [7] provide locality in the connections between nodes in the DOLR, assisting efficient node-level heartbeats. CAN [6] and Chord [8] do not have such locality by default, but may evolve local connections over time. PRR and Tapestry also provide minimal stretch in routing to objects, providing for efficient object heartbeats. Pastry, CAN, and Chord have heuristics to reduce object location cost, so they may perform well in practice; however, efficient object-level heartbeats may be difficult to construct.

All the systems (except PRR, which assumes a static network) use some form of server heartbeats to maintain routing softstate. The server heartbeats allow the system to detect failed nodes and possibly route around them. However, the systems differ in maintaining object state. Only Tapestry and Wells describe object heartbeat techniques and only Wells explains how to repair lost redundancy.

The problem with straightforward object heartbeats in Tapestry[11] is that with enough objects in the system, this can be quite expensive[10]; that is, a significant (over 20%) amount of a servers bandwidth resources were used for object heartbeats. Wells[10] extended Tapestry heartbeats and proposed a server heartbeat exponential backoff, critical ob-

ject notification, and a trust metric to make maintenance, detection, and repair feasible.

Looking at this issue more carefully, however, we note that heartbeats for objects serve two totally different purposes: first, they allow us to detect that a *server* has failed and second, they permit us to repair inconsistencies in the DOLR data structures pointing at objects. The important insight is that slight inconsistencies in the DOLR can be handled through redundancy, while server failure should be noticed more precisely. We extend the work of Wells by aggregating information that will travel over the same links (multicast) to efficiently notice server failure and trigger object reconstruction. Inconsistencies in the DOLR state will still be repaired through periodically republishing object locations, but this process can be done more lazily and locally.

5. Maintenance and Repair

In a dynamic environment systems must adapt to changes in the infrastructure and repair damaged replicas or loss redundancy. A basic assumption of many DOLRs is that there are faulty and malicious nodes that attempt to corrupt data and deny service; however, we assume that there is a large number of "good" servers that properly adhere to the DOLR protocols. We also assume that nodes make provisions to keep local storage and state as stable as possible.

5.1. Infrastructure Repair

Repair needs the DOLR to adjust to changing network configurations. An example DOLR node insertion and deletion algorithm can be found in [3, 11]. Most DOLRs provide mechanisms for both planned and unplanned server removal. When possible, the departing server proactively informs its neighbors of its imminent departure so that neighbors may remove the node from their neighbor maps. Additionally, the node may move replicas to nearby nodes.

To address the unexpected departure of nodes from the network, DOLRs rely on server heartbeats. These server heartbeats are sent along neighbor pointers (as described in Section 3); the fanout of these heartbeats is limited, since the number of neighbor pointers is fixed. However, since the average network distance of these pointers increases geometrically with level number, we send heartbeats along level-1 links more frequently than level-2 links, *etc.*. By monitoring these heartbeats, the routing infrastructure can detect a server's departure and trigger the modification of the routing mesh and redistribution of pointers[4].

---

[3]Experiments show a small constant of proportionality; See [11].

[4]An ongoing area of research is determining legitimate server failure from denial-of-service attacks.
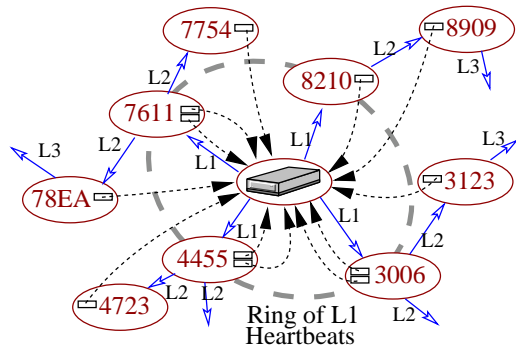
Figure 2. Data-Driven Server Heartbeats: *A slice of tapestry around a server (middle node). Pointers from DOLR back to server define natural multicast tree from server to nodes containing pointers to server. This tree is used for data-driven heartbeats. Locality is achieved by traversing top levels of tree more frequently than bottom levels, and by traversing only portions of the tree for objects below a certain redundancy threshold.*

5.2. Distributed Repair

Distributed repair mechanisms exploit DOLRs distributed information and locality properties[5]. Figure 2 illustrates how the pointers in a DOLR aid in detecting failure by directing server heartbeats. This figure illustrates the natural multicast tree from each server to the set of nodes containing pointers to that server. When a server crashes, this set of nodes must eventually be informed (to clean up pointers and possibly trigger repair). Repair utilizes this multicast tree for *data-driven server heartbeats*. To avoid flooding the network with an excessive number of heartbeats, we provide the same sort of exponential backoff as mentioned with the server heartbeats. Heartbeats go to the first level of the tree most frequently (shown in the figure as the "Ring of L1 Heartbeats"), second level less frequently, *etc*. With this scheme, nodes near a replicas recognize the majority of replica failures and tend to recognize them quickly, while nodes farther away protect against regional outages.

Repair employs two mechanisms to (1) keep the DOLR pointers as up-to-date as possible and (2) trigger repair as soon as the number of surviving replicas from a given block falls below a *threshold*. As shown in Figure 1, Tapestry contains information about the state of an objects replicas. In particular, the *root* nodes associated with an objects replicas know the number and location of all surviving replicas. Although this information is somewhat imprecise,it does provide a framework around which to trigger repair. When a server ceases to send its heartbeats for a sufficiently

long time, the nodes along this multicast tree recognize the failure and propagate pointer changes toward replica roots, which trigger repair when necessary.

The loss of a region of servers may require time to notice. Consequently, we can enhance the observability of certain replicas that belong to objects with "dangerously low" levels of redundancy: we inform all of the pointers on the path from server to root that we want notification as soon as the server ceases to function. The important observation here is that DOLR pointers provide a distributed framework for adjusting the rate of repair and observation as necessary.

6. Conclusion

We have discussed various distributed maintenance, detection, and repair techniques that increase system reliability. Reliable distributed systems are dependent upon location-independent properties of DOLRs. The system reliability is predicated on the use of efficient heartbeats.

References

[1] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, October 2001.

[2] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, 2001.

[3] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed data location in a dynamic network. In *Proc. of ACM SPAA*, 2002.

[4] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*. ACM, 2000.

[5] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of ACM SPAA*, June 1997.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM*. ACM, August 2001.

[7] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, November 2001.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM*. ACM, August 2001.

[9] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, 2002.

[10] C. Wells. The oceanstore archive: Goals, structures, and self-repair. Master's thesis, Univeristy of California, Berkeley, May 2001.

[11] B. Zhao, A. Joseph, and J. Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, 2001.

---

[5]Tapestry's neighbor links encode network locality.