Coherent and Network-Aware Tracking of Objects

Chi Zhang Junwen Lai Sumeet Sobti Nitin Garg Fengzhou Zheng
Arvind Krishnamurthy Randolph Wang

Abstract

The ability of tracking the locations of distributed objects and maintaining cache coherence when they change is crucial for many applications. In this paper, we propose an object tracking system that supports strong coherence semantics and can exploit its awareness of network topology. The network-awareness is especially beneficial in a wide area network where the system's ability of confining routing messages to a smallest possible locality is important. Experiments with a deployment of the system on a real-world Internet overlay show substantial benefits of the system compared to existing approaches.

1 Introduction

In this paper, we study how to track locations of objects distributed in a network, how to route requests to copies of these objects, and how to maintain cache coherence for reading and writing. We would like to support at least two types of application semantics: for a read request, we need to locate *a* copy of the data; and for a write request, we may need to locate *all* obsolete copies (if any) to invalidate them.

A typical application for such a system is a shared file system, with which users can cooperatively read/write data across a wide area network. To achieve good performance, it is desirable to cache data at arbitrary locations close to data users. Users may sometimes also desire strong (serializable) cache coherence semantics.

1.1 Existing Solutions

One existing class of solutions, which we call the *manager-based* approach, is exemplified by the way one locates data objects in cluster systems such as distributed file systems [2, 9, 20], distributed memory systems [6, 10], and cluster applications such as scalable email services [16]. In these systems, a "manager" is responsible for tracking the current locations of the replicas of an object. Read or write requests are first sent to the manager, which in turn forwards the read requests or sends invalidation messages to the hosts which have replicas. To avoid bottlenecks at a single manager, the management of the whole data set is typically distributed among multiple managers. Each object is mapped to one manager which keeps full record of the copies. Consistency of update operations is guaranteed by the single serialization point, the manager.

These systems are initially conceived in cluster environments, where the topology of the network is less of an issue. When the network grows larger and its topology becomes more complex, however, the simple manager-based approach may become problematic. Attempts at co-locating a manager with one of the data users that it serves can be complicated by the fact that the best location of the manager may be unclear in a complex network. Routing all read and write requests via managers, whose locations can be sub-optimal, may not only degrade performance, but also raise the cost of deploying such distributed systems across the wide area, as more wide area bandwidth is consumed by routing requests involving the managers.

Another existing class of solutions is based on distributed hash tables (DHTs) [14, 15, 19, 21]. The chief advantages of this approach are its ability to scale to many nodes and the small amount of routing state on any node. There are also some limitations. Indeed, proponents of the DHT-based approach recognize that there are classes of applications for which this approach is not appropriate. These include applications that require strict consistency among many writers or fine-grained control over the physical location of data [4].

The hash algorithms dictate the placement of data and, therefore, the higher-level systems lose the flexibility of making their own placement decisions with pin-point accuracy. Data replication in DHT-based system can alleviate this difficulty but the replicas only offer limited placement flexibilities and replication can be costly. Compared to a manager-based approach, the number of network hops involved in a DHT lookup can be relatively high, especially in a system of modest size, where the DHT scalability of managing host joins and departures is not a crucial issue.

While a read-only use of the system requires locating just *one* copy, a read-write use of the system with strong coherence semantics would require locating *all* replicas. If the system allows only a fixed number of replicas residing at locations determined by the hashes, locating these copies for invalidation upon writes is easy but read performance may suffer due to lack of caching. On the other hand, if the system allows caching at arbitrary locations, the locations of the replicas are no longer determined by hashes and are therefore difficult to determine. When the scale of the target system is not massive, these limitations of the DHT-based approach may be too high a cost to pay.

It is also possible to combine the manager-based and

DHT-based approaches, so the location of the manager is decided by the DHT algorithm. The DHT-based approach provides a scalable way of routing toward managers, and the manager-based approach allows arbitrary data locations to be tracked. The lack of network-awareness in the manager locating mechanism, however, persists; and the amount of location state on a manager can be proportional to the product of the number of objects and the number of hosts, a figure that negates a key advantage of the DHT-based systems, namely, its small routing state.

Finally, it is worth noting that we do not see the system that we propose in the rest of the paper as a competitor to DHTs. Instead, it is a network-aware extension of the manager-based approach that addresses a class of applications or environments for which DHTs are less suitable. As we discuss in Section 3.1, there are also ways of combining DHTs and the proposed system that address some weaknesses of each approach.

1.2 CANTO Properties

We call our object tracking and routing system CANTO (Coherent And Network-aware Tracking of Objects). It has the following properties. (1) The system is coherent. It supports strong serializable consistency semantics. (2) The system is network-aware. For a read request, the system attempts to locate a copy that is "nearby." When routing to all copies of the object, such as for invalidation for writes, the messages are confined to a network area spanning only those copies. (3) CANTO allows higher-level systems to make their own arbitrary data placement decisions. (4) The precise routing state for a piece of data is not widely dispersed to every node in the system. Even when the object location changes drastically, this controlled narrow distribution avoids the need of expensive updates to the routing state on every node. (5) CANTO is self-synchronizing, in that it preserves the integrity and consistency of its data structures during concurrent read and write operations without resorting to an external locking mechanism or fixed single serialization points. (6) CANTO is self-recovering in that it reconstructs its in-memory routing state from persistent storage and/or neighboring nodes after a crash.

CANTO utilizes a tree-like routing topology[1]. The use of a tree routing topology is by no means new—existing systems such as Scribe [5] and Globe [3] utilize similar topologies as well. We explain the detailed secondary differences between CANTO and these systems in later sections. The primary difference, however, is the first CANTO property listed above, namely, that CANTO provides strong cache coherence semantics when facing concurrent read/write operations, semantics that publish/subscribe systems or generic location tracking systems are not designed to provide. A key technical innovation of CANTO is the operational details of a protocol that provides for this cache coherence on a tree-like routing topology, not the tree-like topology itself. Furthermore, CANTO provides persistence and recovery of routing state in case of node failure, while some existing systems only guarantee recovery of topology connectivity.

A property that is missing from the above list is compact routing state on CANTO nodes. CANTO requires per node routing state that can be proportional to the product of the number of objects and the number of neighboring nodes[2]. In addition to several state compression techniques that we discuss in a later section, CANTO stores most of this routing state on disks and uses a memory buffer for caching and write-behind. In effect, CANTO trades disk storage of routing state for reduced usage of wide-area networks and better performance. We believe that this approach is consistent with both hardware technology trends and software architecture trends, exemplified by the popularity of systems such as web caches and content distribution networks. (The amount of routing state should be small compared to the storage devoted to data replication, and unlike CDN-like applications, CANTO is designed to support coherent read-write applications.)

2 Algorithm

2.1 Network Routing Analogy

Before we describe the CANTO algorithm, let us consider Internet host routing as an analogy. We may present an object ID (an IP address in this case) to any node in the system. Even though this node may not "know" the exact location of the target, it "knows" where the next-hop destination is in order to make progress. An advantage of this approach is that it can exploit topological locality well in certain situations: for example, when an object moves within a small locale, routing table entries of far-away nodes are unlikely to be disturbed. A disadvantage is that some amount of routing information needs to be widely disseminated to essentially every node in the system so any node is capable of routing requests for any object. If the location of an object changes drastically, routing table entries of a large number of nodes may be affected.

Unlike routing for conventional Internet hosts, which do not often change their locations drastically and cannot be "replicated," routing for generic objects distributed over the network need to contend with these possibilities. A naive adoption of an existing routing algorithm, such as the link state protocol, may result in high cost incurred by excessive routing state propagation, slow convergence toward consistent distributed routing state, the complexity of managing inconsistency during routing state propagation, and the com-

[1] We call it a "tree-like" topology because, as we explain in Section 2.3, it is not exactly a tree.

[2] It is worth noting that Scribe and Pastry [15], the DHT system upon which Scribe is built, have different amount of routing state: while Pastry has minimum DHT routing state, Scribe's additional state can be proportional to the product of the number of objects and the number of overlay "neighbors," which is no less than that of CANTO.

plexity of coping with routing loops. Despite these disadvantages, the topological awareness advantage of the hop-by-hop Internet routing is one that we would like to incorporate into CANTO.

2.2 Terms and Data Structures

We define several terms and data structures used by the CANTO routing protocol. The precise roles of these terms should become clearer in subsequent sections. For each object tracked by CANTO, there is an *anchor* node. The object-to-anchor mapping is similar to a "manager map" in several cluster systems [2, 6, 9, 20] in that it is a relatively static and compact map that is known to all nodes. In the basic CANTO algorithm, routing messages (including both requests and replies) travel along edges of a spanning tree rooted at the anchor. We later discuss the use of more general topologies. With respect to each object, all nodes fall in one of two categories: *active* and *inactive* nodes. Formally, if a copy of the data is stored on node X, then all nodes between the anchor and X (including the two end points) are active. All nodes that are not active are inactive. On each node, logically, there is one bit associated with each edge and each object. We call this bit an *edge bit* and the edge bit is one only if there is a copy of the data in the direction of that edge pointing away from this node. We call such an edge a *1-bit edge*. An inactive node has no 1-bit edges. An invariant of the system is that there is always a path of 1-bit edges leading from the anchor to a replica and it is possible to reach all replicas from any active node by following the directed 1-bit edges. We refer to these edge bits collectively as metadata or routing state.

2.3 Basic Operations

There are essentially two cases of routing. (1) Starting from an active node, one should be able to reach one or all replicas of the data by traversing 1-bit edges. (2) Starting from an inactive node, one first routes a request toward the anchor. Once this message encounters an active node en route, which can be an active node lying on the path between the originating node and the anchor or the anchor itself, which is itself active, further routing of the request proceeds as in case (1). We now illustrate the basic CANTO routing algorithm using a series of example operations (shown in Figure 1).

Figure (a) shows the initial state of the spanning tree. The only active node is the anchor. In Figure (b), node X creates the data. Since it is an inactive node, it sends a write request toward the anchor to inform the rest of the system the current location of the data and to invalidate any potential obsolete copies. The first active node encountered by this message is the anchor and since there is no obsolete copy to invalidate, the message terminates there. As the reply message flows back along the same path, edge bits are set to maintain the invariant. Figure (c) shows the resulting tree state. All nodes between X and the anchor are now

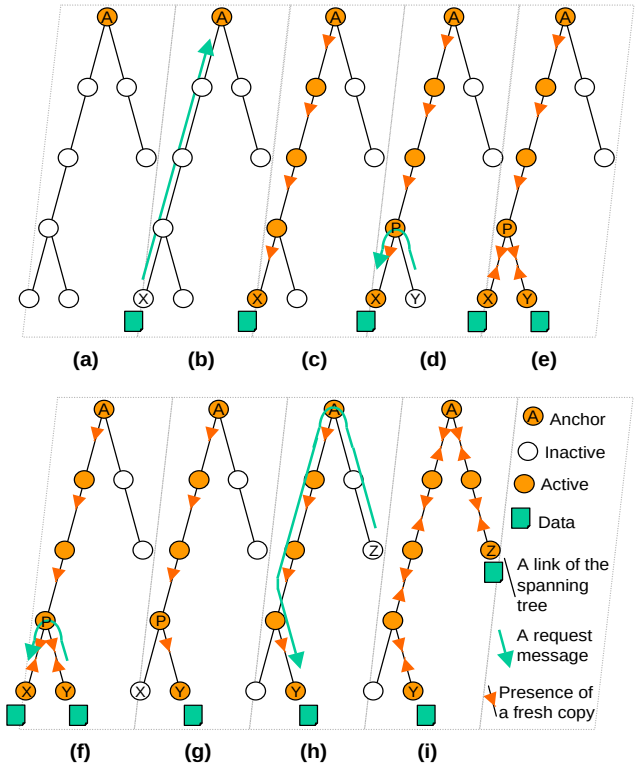


Figure 1: Example basic CANTO operations.

active. All edge bits of edges lying on the directional path from the anchor to X are set to one.

In Figure (d), node Y issues a read request. Y is an inactive node so the request is routed toward the anchor. As soon as the message reaches an active node P en route to the anchor, the message starts to follow 1-bit edges instead and is diverted toward a replica at node X, which sends the requested data back to Y. Note that the data reply need not follow the spanning tree edges but the metadata reply, which is responsible for restoring the invariant, does. Figure (e) shows the resulting tree state, assuming node Y has chosen to cache locally a copy of the data (which it is not required to). The edge bit of the single edge of node X is newly set, pointing toward Y (via P), and the edge bit of the single edge of node Y is also newly set, pointing toward X (also via P). Node Y has become active. Note that the CANTO invariant demands more than simply maintaining a path of 1-bit edges from the anchor to the replicas: the two replicas in this case also need to point toward each other. The utility of these extra 1-bit edges would become clear in the invalidation operations to be described next. Also note that although we have chosen to only cache data at leaf nodes in these examples, CANTO does not intrinsically differentiate leaf nodes from interior nodes and data can be cached at any node and tracked by CANTO.

In Figure (f), node Y issues a write request. Y is an active node so the request is multicast to all other replicas to invalidate them by following the 1-bit edges. A reason for

CANTO to employ a tree topology to route its request and reply messages is to avoid potential routing loops. Note that not all active nodes are involved and the anchor need not always participate in the invalidation process: the 1-bit edges from the anchor to the common ancestor of X and Y are not disturbed in this case. This is why CANTO does more than simply maintaining paths of 1-bit edges from an anchor to all the replicas. The fact that it is possible to reach *all* replicas from *any* active node allows the invalidation messages to be confined to a small neighborhood when possible. This is one of the important differences between CANTO and some existing routing schemes, such as the one employed by OceanStore [8], whose invalidation operations always need to reach a "root" node first. (We discuss other differences in the related work section.) Figure (g) shows the resulting tree state. Node X has become inactive and the edge bits of node X, Y, and their common ancestor, P, are updated to reflect this change. (The algorithm "knows" which edge bits to clear and which to set by examining the direction from which the invalidation messages arrive. For example, the edge bit on $X \rightarrow P$ is cleared as this edge points toward the anchor, while the edge bit on $P \rightarrow Y$ is set as this edge points away from the anchor.)

In Figure (h), node Z issues a read request. Since Z is inactive, it routes the request toward the anchor, which happens to be the first active node that the message encounters in this case. From there, the message follows the 1-bit edges to reach node Y, the only replica in the system, which supplies the data to Z. Figure (i) shows the resulting tree state. All nodes between (and including) Z and the anchor are active as a result of a copy being cached at Z. All the edge bits lying on the tree path between Y and Z are set as the metadata reply retraces the steps of the request message.

Active nodes maintain dynamic, precise, verbose, and hop-by-hop information of where the replicas are, while inactive nodes rely on relatively static, generic, compact, and direct pointers to anchors. One implication of this differentiation is that the maintenance of the precise routing information is limited to the nodes that have participated in operations on the data. This is in contrast to routing protocols such as the link state protocol where some amount of up-to-date routing state must be disseminated to all nodes. In a way, CANTO is similar to Mobile IP [12] as it relies on a relatively static "home" node as the last resort forwarding agent, a role that is similar to that of a CANTO anchor. (Unlike a Mobile IP home node, a CANTO anchor is not always involved in routing operations, as requests routed toward the anchor by an inactive node may be diverted by an active node en route.)

We should also note that although we have loosely referred to CANTO routing graphs as "trees," they are not trees in a strict sense: although the *undirected* skeletons of the routing graphs in Figures 1 are trees, the *directed* graphs are more complex—each undirected tree edge may have zero, one, or two directed 1-bit edges associated with it.

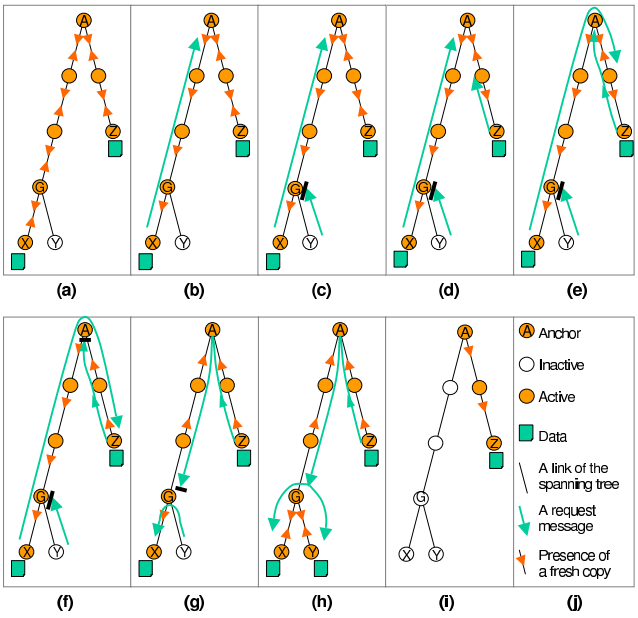


Figure 2: Example of conflict resolution.

This is in contrast to existing systems, such as some shared virtual memory systems [10], Scribe [5], and GLS [3], where the routing topologies are strictly trees. One purpose of these explicit 1-bit edges is to ensure that messages resulting from updates (such as publishes, writes, and deletes) are confined to a network neighborhood that does not necessarily have to always include a tree root.

2.4 Conflicting Operations

In previous descriptions, we assume there are no conflicts in metadata operations. In reality, different users can ask to read/write the same object at the same time. From the system's point of view, when the concurrent requests try to change route state along their paths, they may interfere with each other and ruin the integrity of the routing state. From the users' point of view, the read and write operations need to be ordered to ensure consistency. Essentially, these are the two sides of the serialization problem.

We do not use fixed serialization points like managers or external locks. Instead, we allow the serialization happen at dynamic points close to requesters. When a request reaches a node, the node checks for conflicts with other pending requests. (Conflicts happen between read and write, or write and write requests for the same object.) If there is a conflict, the new request will be blocked until the pending request finishes. In case of a "head-on-head" collision, i.e. two nodes send conflicting requests to each other at the same time, we break the tie with the topology order. A request from an "upstream" node (closer to the anchor) gets higher priority. The "downstream" request has to wait.

Figure 2 gives an example of the conflict resolution. X sends a write request toward the anchor. Y tries to read the same object shortly after. Y's request is blocked at node

G by the request of X. In step (e), X's request runs into another write request from Z. Since they collide "head-on-head", we give request X the green light. After request X invalidates Z's copy, it turns back and releases Z's write and Y's read requests in turn. Y's request will follow the new bits to reach X's data. Finally in step (i), Z's request clears the copies in X and Y. Besides ensuring routing state integrity, the step by step local serializations above also give the requests a global serialization order of X, Y, Z.

Hop by hop serialization brings up the concern of potential deadlock. In the above example, if request X gets blocked by request Z, and Z somehow reaches Y via another invalidation branch and gets blocked, there will be a deadlock by circular wait. However, this cannot occur in a tree topology. First, the loop-free property ensures that request Z cannot reach Y without passing node G. Second, the topology order entitles request Z to take precedence over any other request that reaches node G at the same time that request Z does, since it will be an upstream request there. Generally, the topology order ensures that one request (the one closest to the anchor) has higher priority than any other one and thus can always make progress.

2.5 Enriching Routing Topologies

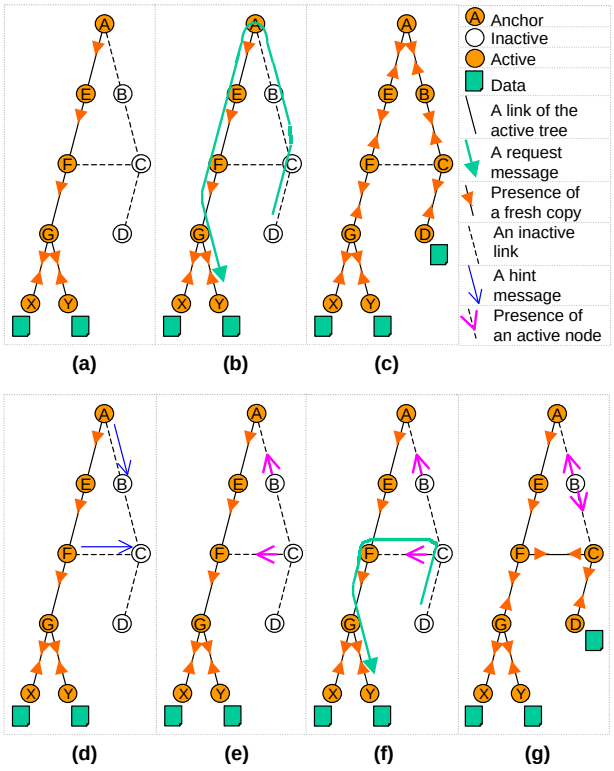


Figure 3: Comparison of routing scenarios with and without hint edges.

In the basic CANTO operations discussed above, we insist on routing protocol messages along edges of a spanning tree that is statically determined. The tree-topology restriction guarantees the absence of routing loops at the expense of excluding the use of certain links. One potential cause of concern is that the tree edges near the anchor may become bottlenecks. This concern, however, is unlikely to be a real problem, due to at least two reasons.

First, although there is only one tree per anchor, there should be many anchors and, therefore, many trees. Edges unused or lightly used in one tree could be more heavily used in another, so the overall utilization of the links and nodes will probably be well distributed. Second, the CANTO protocol is designed to allow both read and write requests to be satisfied by active nodes at lower levels of the tree without participation by the anchor.

Nevertheless, the use of a statically determined spanning tree for routing could be overly restrictive. In this section, we examine a simple example technique of enriching the topology with more dynamic information and route choices. Let us consider the example illustrated in Figure 3.

Figures 3(a)-(c) show a read request as we have seen in Section 2.3. While adhering to the statically determined tree edges correctly locates the object, it may not be the optimal way. Note the possible path taking the $C \rightarrow F$ shortcut.

The remaining panes of Figure 3 show an enhancement to the basic algorithm. In Figure (d), active nodes A and F send *hint* messages to their inactive neighbors B and C, informing them about the presence of active neighbor nodes. Figure (e) shows the result: nodes B and C store *hint bits* pointing to their active neighbors, and we call the inactive links pointing to the active neighbors *hint edges*.

Figure (f) shows the alternate path taken by a read request from node D: instead of following the statically determined tree edges, the request is diverted along the dynamically formed hint edge $C \rightarrow F$, toward an active node F. From an active node, the routing of any request proceeds as previously described in Section 2.3. This path is shorter than the one determined by the basic algorithm in Figure (b). Figure (g) shows the routing state after serving this request. Note the difference between (c) and (g): the subset of active nodes and edges connecting them are different. We call the subtree consisting of active nodes and edges between them the *active tree*. In Figure (g), the *active tree* is more concentrated around the data copies, alleviating the anchor burdens of routing future requests.

In this sequence of example operations, we have shown the setting of hint bits as neighboring nodes become active. The converse, namely the clearing of hint bits, should occur when active nodes become inactive due to invalidate operations. To amortize the cost of these propagations, we delay and send them in batches. As a result, the hint bits can be out-of-date with respect to the neighboring nodes that they point to. To cope with inaccurate hint bits, we fall back to the statically determined default spanning tree edge routes if a message reaches an inactive node after mistakenly following a hint edge.

The flexibility of forming active trees in the dynamic algorithm introduces additional complexity. Now the objects

sharing a single anchor no longer share an identical active tree. For each node, its parent node in the active tree becomes dynamic instead of being static as in the basic algorithm. Recall that in the process of serving a write request, in order to determine whether an edge bit needs to be set or cleared, a node needs to "know" whether the request is arriving on a link pointing toward the anchor or pointing away from it. This determination is easy under the basic algorithm because the parent node is always static and unique to the anchor. In the case of dynamically constructed active trees, however, we must keep additional state, namely the parent node of each node for each object. This additional state is yet another example of the trade-off between keeping extra state and reducing network usage.

In Section 2.1, we have discussed the cost and complexity of routing to distributed objects over a general mesh. The static routing trees discussed in Section 2.3 has less routing state propagation cost and less complexity. These two topologies for routing, namely an arbitrary mesh and a static tree, represent two extremes. The enhancement discussed in Section 2.5 is an example design point in between, as the combination of the active edges and hint edges form a routing mesh that offers more flexibility than that afforded by the static trees. This enhancement, however, is only a relatively simple attempt: the single-hop propagation of hint messages from active nodes to their inactive neighbors introduces relatively little cost; and routing loops are trivially avoided on the resulting mesh. Although this approach appears to provide substantial benefits in some situations, we view this enhancement as merely a starting point for ongoing research into better and more general routing topologies.

3 Discussions

3.1 The Making of CANTO Nodes, Anchors, and Routing Graph Edges

In describing the CANTO algorithm, we have deliberately been vague on issues such as how the anchors are assigned and how the CANTO routing graph is formed. There are several options for addressing these issues and these options are orthogonal to the workings of the basic CANTO protocol.

There are two options for making CANTO nodes: (1) physical routers embedded inside the network; or (2) overlay nodes at the edge of the network[3]. Technologies such as "extensible routers" [17] may allow CANTO protocols to be efficiently implemented in a certain class of physical routers, while emerging overlay networks [1, 13] provide platforms for the second alternative.

There are two options for assigning CANTO anchors: (1) relatively unstructured "manager maps," as those employed in several cluster systems [2, 9, 20, 6, 10, 16]; or (2) consistent hashing [7], of which DHTs [14, 15, 19, 21] are one class of examples.

[3]Note that "overlays" are more general than DHTs.

There are three options for making CANTO routing graph edges: (1) physical Internet routes (between nodes and their anchors); (2) "unstructured" overlay routes; or (3) "structured" overlay routes, such as the successive lookup hops determined by DHTs.

A cross-product of all the above options produces many combinations; of course, not all these are meaningful. We consider some of the most obvious combinations.

Combination A: overlay nodes $\times$ unstructured manager map-based anchor assignment $\times$ unstructured overlay routes. This is one of the most flexible combinations; and it has the potential of best performance and least scalability. One can choose to place anchors at strategic locations in the network to avoid bottlenecks, or move them closer to some clients as access locality is observed. One can include in the CANTO routing graph more overlay links between nodes that closely share objects, or deliberately avoid some slow links. Section 3.2 gives more scenarios where one might want to manipulate node degrees in the CANTO routing graph. (The investigation of an optimal CANTO routing graph, however, remains an open research question.) In addition to better network-awareness, an additional advantage of an anchor assignment and routing scheme that is not hash-based is that "related" data items can share identical or similar routes and routing state. This results in higher spatial locality in accesses to data and meta-data, more compact routing state, and better routing performance. (We discuss the management of routing state more fully in Section 3.2). Combination A, however, is only appropriate for a modest-sized system, where the change of node membership is not too frequent.

Combination B: overlay nodes $\times$ consistent hashing-based anchor assignment $\times$ structured DHT-based routes. Under this combination, the routing state is similar (but not identical) to that maintained by the Scribe publish/subscribe system [5]. (As explained in Section 1, a key difference between CANTO and Scribe is that CANTO provides strong cache coherence semantics.) Combination B has the exact opposite attributes of combination A: it is less flexible, less network-aware, less efficient, has less compressible routing state (due to randomization of hashing), but can scale to a larger number of nodes. Combination B is the reason that we view CANTO and DHTs as complimentary technologies: the former provides a cache coherence protocol while the latter provides scalable anchor assignment and routing.

Combination C: overlay nodes $\times$ consistent hashing-based anchor assignment $\times$ unstructured overlay routes. This combination is a middle-ground between combinations A and B: consistent hashing minimizes data movement when anchor set membership changes while unstructured overlay routes allows flexible construction of network-aware CANTO routing graphs.

Combination D: physical nodes $\times$ unstructured manager map-based anchor assignment $\times$ physical routes. In environments such as a campus-wide enterprise network,

or a bi-coastal company, where the concerned organization may have administrative control over a number of programmable routers, CANTO may naturally derive its network-awareness by partially mirroring the physical interconnect. Anchor assignment may reflect internal organizational structures: for example, large chunks of a file system name space may be "anchored" at nodes near "owner departments."

3.2 Managing the Amount of Routing State

On each node, CANTO logically stores a bit for each object and each network link, denoting whether a replica of the object is present in the direction of that link. The amount of this routing state is therefore proportional to the product of the number of the objects and the number of the incident links. By allowing arbitrary placement of data objects and tracking their precise locations, CANTO pays the cost of state storage space to gain the benefits of coherent semantics, reduced network usage, and better performance. This trade-off is a key goal of CANTO. Section 2.5 has given an example of managing this trade-off where we use additional per object state to build better routing topologies that should result in more savings on network usage. We examine several additional ways of managing this trade-off.

When a read request arrives at a node, encountering multiple 1-bit edges, each of which can lead to a separate replica, CANTO must make a choice of which 1-bit edge to forward the request on. So far in our discussions, we have assumed that we store one bit per object per edge. This is sufficient for locating a copy but CANTO does not have enough information for locating a "good" copy. A possible enhancement is to store more than one bit of routing state per object per link, allowing better "resolution" of differentiation of the "quality" of these replicas. This is analogous to the "distance" metric employed in routing protocols such as "distance vector."

There are several complications with this approach. First, to maintain these distance metrics, one needs to propagate extra messages beyond what is required by the algorithms described earlier. Of course, these distance metrics are merely hints, so the propagation can be delayed and batched, and the propagation frequency can be flexibly controlled. Second, as is the case with Internet routing, how one computes the distance metrics can be a complex question. Possible factors include the number of network hops (in terms of the number of intermediate CANTO nodes, for example), latency, and bandwidth. The properties of the object being read (such as its size) may allow CANTO to choose a better metric. Third, data replies can be sent directly from the host housing the chosen replica to the original requester, potentially bypassing the CANTO nodes responsible for forwarding the CANTO protocol request and reply messages, and complicating the distance computation on the CANTO nodes. The distance metrics computed purely based on the CANTO protocol messages (instead of the data replies), however, can still be a useful hint.

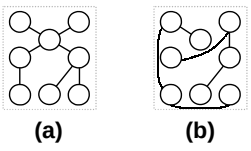


Figure 4: Partitioning.

While the above techniques introduce more state for better routing, one may sometimes desire the opposite: a reduction in the amount of routing state on a CANTO node at the expense of less accuracy or less flexibility of routing. This, for example, may allow a CANTO node to cache the routing state of a larger fraction of the objects in its memory, thus speeding up lookups on an individual node. The decision that one must weigh is whether this local speedup is justified by the overall distributed cost. We consider several routing state compression techniques and their cost.

The first technique is to restrict the routing graph topology: limiting the maximum number of possible incident edges to a CANTO node. This technique is most relevant to the case of a CANTO system made of overlay nodes, where one has a large degree of freedom choosing an overlay topology. Reducing the incident degree of nodes, however, increases the diameter of a routing graph (or the height of a routing tree) in terms of hops, thus potentially increasing routing overhead.

The second technique is to restrict the number of replicas. Suppose a node has an incident degree of D and we dictate that no more than $0 <= n <= D$ of these edges can be 1-bit edges, then the number of bits needed to track the replicas is $\log \sum_{i=0}^{n} \left(\begin{array}{c} D \\ i \end{array} \right)$. Limiting the number of replicas, of course, reduces the effectiveness and flexibility of caching.

The third technique is to move replicas to strategic locations inside the CANTO routing graph. Note that the CANTO algorithm allows replicas to be kept at any node. Consider the example routing tree in Figure 1(e). Instead of keeping two replicas at nodes X and Y, we may instead keep a single replica at their parent P. (We assume that we store inactive state using compact sparse representations.) This movement reduces the amount of state on all three nodes at the expense of a more distant replica for both X and Y when they reuse the data.

We call the fourth technique "partitioning." Figure 4 shows an example. Instead of having a single large routing tree that encompasses all the nodes in the system (Figure (a)), we partition the nodes into k subsets ($k = 2$ in Figure (b)), and nodes of a single routing tree are always drawn from a single partition. Each routing tree or partition is responsible for tracking a subset of all data objects. This approach has several potential advantages. First, we limit the maximum amount of routing state on any CANTO node to $1/k$th of the original maximum. Second, in the original ap-

proach of encompassing all nodes in a single tree, although each anchor is only associated with a subset of the data being tracked, an interior CANTO node may participate in the routing of many "hot" objects that are tracked by different anchors. Partitioning prevents the possible development of such bottlenecks. Third, partitioning may reduce the number of routing hops without compromising CANTO's ability of exploiting topological locality of requests, as long as nodes within a partition are properly spaced. Of course, there is a limit to how large k can be without compromising the CANTO goals. In the extreme case, when k equals to the number of the nodes in the system, each partition becomes a singleton set and the system degenerates to a manager-based solution.

4 Recovery and Reconfiguration

The goal of the recovery and reconfiguration mechanisms is to ensure consistency of the distributed CANTO routing state. We consider three recovery mechanisms. The first copes with a node that has crashed but reboots shortly thereafter. The second employs redundancy to allow backup nodes to take over the responsibilities of a node that fails for a more prolonged period of time. In case redundancy is absent or not sufficient, we employ a third mechanism of repairing the routing trees to exclude the use of failed links and/or nodes.

While CANTO strives to maintain the consistency of the routing state and uninterrupted routing operations, we note that it is *not* the responsibility of CANTO to guarantee the reliability of the *data* that is being tracked: instead, it is the responsibility of the systems that are built on top of CANTO to decide what levels of data reliability they desire and what mechanisms they should employ to achieve them.

4.1 Recovery from Unplanned Reboot

A CANTO node maintains its routing state on disks for two reasons: first, there might be more state than can fit in main memory; and second, persistent storage allows a CANTO node to "page in" its routing state after a node reboots. It may be too costly to synchronously write state changes to disk, however. For this reason, CANTO buffers state changes in memory and logs them to disk in batches periodically. Though the most recent changes can be lost upon an unplanned reboot, we can recover the state with the logs from its neighbors.

CANTO nodes maintain counters for messages they send to neighbors. They also keep counters about messages received from neighbors. Since all state changes are associated with message exchanges between neighbors, we can determine the current state given the set of message counters that we have sent to and received from the neighbors. We call this set the *counter vector* for the current state. When a node writes a log block, the current counter vector is also included.

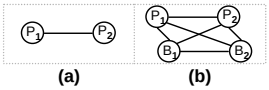


Figure 5: Redundancy.

The recovery process starts by reading logs from disk. After locating the tail of a valid log and reading the counter vector, the node contacts every neighbor with the counters. A neighbor compares the send/receive counters with what it has, and sends back any log blocks containing larger counter values. After acquiring logs from all of its neighbors, a node can replay all the logs to recover the state before the reboot. For replay purpose, the log entries are actually read/write operations instead of bit changes.

In cases of concurrent failures, the above log transfer process is still used. The guarantee, however, is consistent state in the system instead of exact recovery to the state before the crash. Two restarting nodes can exchange their counter vectors and log blocks. In the end, they will have the same view of the state, but some updates before the reboot may be lost if not logged into persistent storage in any node. When more nodes are rebooting, the pair-wise log exchanges continue until every node has received every log entry it did not have. The system will have consistent state across all nodes after convergence. Before this is done, the rebooting nodes delay the processing of new requests.

4.2 Redundancy

In the previous section, we discussed how to recover the routing state after a node reboots from a crash. CANTO also uses this mechanism to provide node backups in cases where the reboot process can take a long time. Figure 5 shows an example of *primary* nodes P_i and their *backups* B_i. During normal operation, P_i periodically sends its log to its backup B_i. If P_i crashes, node B_i starts the recovery process by reading through the log and sending the log counter vectors to the neighbors of P_i. After the neighbors send back the log entries missing from B_i's log, B_i can replay the logs and gain full knowledge of P_i's routing state. Then B_i can takes over P_i's routing responsibilities.

To enable node backup, a single link in Figure (a) is replaced by a complete graph linking a pair of primaries and backups in Figure (b). We do not necessarily need to double the number of physical nodes to provide the desired redundancy: backup nodes in one routing tree or in one part of the system can be primaries elsewhere. Some of the redundant links can be overlay links that share underlying physical links, although some amount of redundancy in the physical link topology is still needed to maintain connectivity upon physical link failures. Naturally, it is desirable to designate backups that are close to their corresponding primaries. In a simple example in Figure 5(b), P_1 and B_2 could share a single physical node, while P_2 and B_1 could share another physical node. A failure of node P_i would prompt B_i to initiate the recovery process and take over the role of P_i.

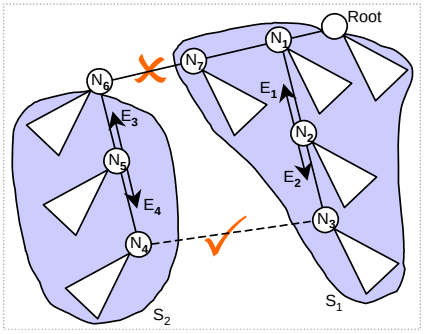


Figure 6: Repairing a routing tree. The failed link $N_{6,7}$ is replaced by $N_{3,4}$. Each triangle represents a forest of subtrees rooted at the node represented by the circle above the triangle.

Routing under this situation may become less efficient because this single physical node now serves all neighbors of the two nodes. However, there should be no interruption of routing.

The cost of maintaining backup nodes is the bandwidth and processing needed for transferring logs from a primary node to its backups. This overhead is not substantial since the log only records metadata operations, and thus is small. The log transfer can occur in the background. There is a trade-off between the overhead in normal operation and prompt response to failure.

If we use a DHT to locate anchors and use overlay hops determined by the DHT algorithm as CANTO routing tree edges (as described in Section 3.1), CANTO may need to cooperate with the redundancy mechanism already provided by the DHT algorithm. For example, in the routing algorithm of Chord [19], the immediate successors of P_i would be used for routing if P_i failed. CANTO should place the corresponding backups B_i at those locations. If node P_i fails, the Chord routing layer will redirect messages destined for P_i to the backup node, triggering the recovery process.

4.3 Repairing Routing Trees

When a physical link fails, it may not be necessary to immediately reconfigure the routing tree: one may be able to find an alternate route (that is either a physical or an overlay route) that reconnects the end points of the failed link, so the routing tree can remain functional. This routing tree, however, may no longer be optimal. If the link failure is long-lasting, one may wish to reconfigure the routing tree to exclude the failed link.

Figure 6 shows an example. In the short run, messages over the failed $N_{6,7}$ link can be routed over the alternate $N_{6,4,3,1,7}$ route. This rerouting happens transparently to CANTO and the original routing tree does not have to be reconfigured to remain functional. In the longer run, however, it is better to reconfigure the tree to replace the $N_{6,7}$ link with $N_{3,4}$. We continue to use this example to illustrate the tree repair algorithm.

When a link is severed, the nodes of the original routing tree are divided into two subsets: S_1 and S_2. We locate a new link that reconnects the two subsets. The example in Figure 6 is general: all cases of severance and reconnection can be illustrated with a figure similar to this one. It is easy to see that the routing state on all the nodes within all the subtree forests illustrated by the triangles do not need to change when the tree is reconfigured. This is because the only way to route into or out of these forests is through the common forest roots and this fact is not affected by the reconfiguration. In fact, the only nodes that need to have their state updated are the ones on the cycle that includes both the severed link and the new link, namely, $N_{1,2,3,4,5,6,7}$ in this figure.

To see how this state update is accomplished, let us consider N_2, a "typical" node on the cycle that is in S_1. We need to change the state corresponding to E_1, the directed edge pointing toward the severed link, and E_2, the directed edge pointing toward the new link. For every object in S_2, if it has a 1-bit edge on E_1, we need to clear this bit and set the corresponding bit for E_2, to reflect the fact that one must route through E_2 instead of E_1 to reach any object in S_2. Similarly, for N_5, a "typical" node on the cycle that is in S_2, for every object in S_1, if it has a 1-bit edge on E_3, we clear this bit and set the corresponding bit for E_4. We say we *flip* the bits for these objects.

The repair process starts at N_7. For each object that has a 1-bit edge on $N_{7,6}$, the broken link, we include the object in a set X. We flip the bits for X at N_7. Next, we exclude from X all the objects in it that can also be found in the subtree rooted at N_7 (illustrated by the triangle). More precisely, for each object in X, if it has a 1-bit edge on any of N_7's incident links excluding the two on the cycle, or if the object is cached on N_7 itself, we remove the object from X. We call these removed objects *unaffected objects at* N_7. Next, N_7 transmits the resulting set X to N_1. Upon receipt of the set, we flip the bits for X at N_1. We then further remove from X all unaffected objects at N_1 and send the resulting X to N_2. The process repeats until it reaches N_3, the node responsible for establishing the new link. At the same time, a similar process starts at N_6 and ends at N_4.

This recovery process requires the transmission of the routing state of the "affected" objects on the cycle of "affected" nodes. This process may occur in the background while foreground routing continues on the old tree using the alternate $N_{6,4,3,1,7}$ route to emulate the broken link.

5 Implementation and Applications

We have implemented and deployed CANTO as a real system. Most of the algorithms and features we discussed above have been implemented, including the static and dynamic tree routing algorithms, conflict resolution, and recovery after crashes. The core CANTO routing algorithms take up about 5,000 lines of C++ code. Other modules contain another 5,000 lines. CANTO nodes communicate with

each other via TCP connections. The internal structure of a CANTO server uses a single process event-driven model, with some helper threads for reading bits from disk when the bitmap cannot fit into main memory. Other parts of the system, like node backup and routing topology repair, are left for future work.

CANTO is designed as a middle layer for building distributed applications that require persistent tracking of objects. Possible applications include, but are not limited to, wide area read/write storage, multicasting services, and distributed lock managers. We have implemented a distributed virtual disk and a distributed file system on top of CANTO. Both run on unmodified Linux with our kernel modules.

Our distributed virtual disk uses CANTO to track individual blocks. Each virtual block is mapped to a CANTO object. Clients of the virtual disk use local disks for cache copies. A kernel module exports a disk device driver interface to the Linux kernel. It redirects any disk requests to a user-level daemon, which either feeds back the data if the request hits in the cache, or invokes the corresponding CANTO operation for a miss. The client daemon also responds to commands from the CANTO server to forward or invalidate a disk block.

Our distributed file system supports shared read-write semantics similar, but not equivalent, to that of a local file system. We map each file or directory to a CANTO object. Clients cache some files/directories in local storage. CANTO keeps track of the cached copies in the client machines and routes a client request to the node(s) with cached copies, which invalidate and/or forward the object to the requester.

File systems provide richer semantics than reads and writes. To support object creation and deletion, CANTO provides operations to allocate/deallocate object IDs. Supporting file attributes can be tricky. It is common to access file attributes without touching data and in many cases, like "ls -l", the attribute information need not be precise. So we allow users to cache file attributes even when the file is not in the local cache. To avoid the overhead of tracking attributes separately, we only ensure that attributes are accurate when the file is actually cached in the local machine. In cases where this is not sufficient, a user can update her view with a special read request that only fetches the attributes.

Many applications have weaker update semantics requirements than strong serializable consistency. Some of them can be supported in CANTO. For example, session semantics can be implemented if the user calls CANTO to invalidate other copies at the end of a write session.

6 Experimental Results

This section presents results of routing throughput and latency on a subset of PlanetLab [13]. We use 35 nodes distributed in North America (30), Europe (4) and Australia (1). Among the nodes in North America, 27 are from .edu domains and another 3 from .net, .org and .gov domains.

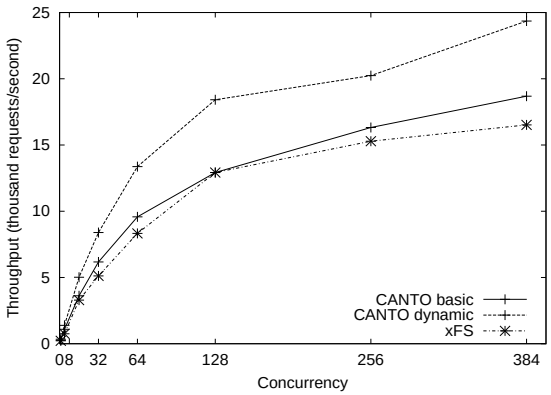


Figure 7: Throughput under random workload, 90% read.

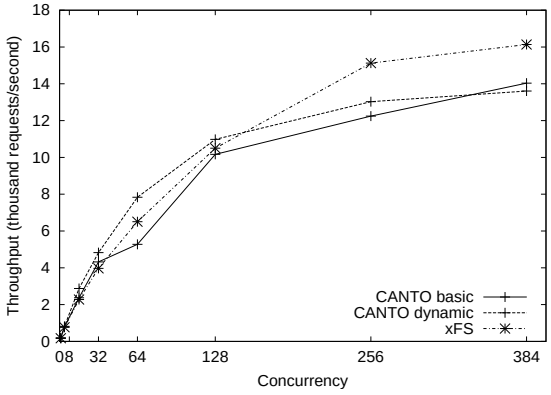


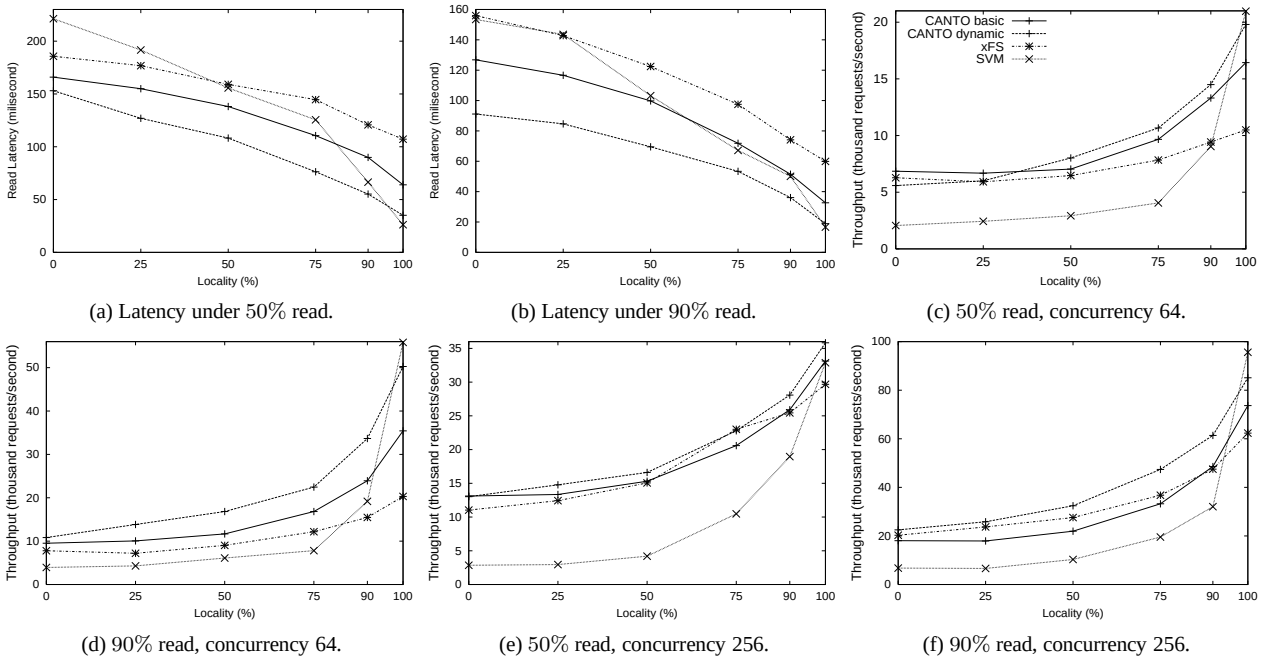
Figure 8: Throughput under random workload, 50% read.

We configure one CANTO server and one client on each node. In order to measure performance under different load levels, we limit the number of outstanding requests a client can have. This determines how many concurrent requests can be generated at this node. We call this number the request queue length or concurrency in the rest of this section. Unless otherwise stated, the tests are done with 0-byte data to show the metadata routing performance.

We compare CANTO against two distributed tracking algorithms: xFS style distributed manager [2] and SVM style dynamic manager [10], both of which use a fully connected topology. We place an anchor/manager at each node. The CANTO routing tree is built about 3 to 4 levels deep with an out-degree of 16 at the anchor nodes. We pick such a "fat" topology since the links are overlay connections without physical hierarchy. Two variants of CANTO are tested—the basic and dynamic algorithms as described in Section 2.

6.1 Throughput of Random Workload

In the first test, we show the performance under a random workload. Figure 7 compares the throughput of different algorithms when we increase the number of concurrent requests. The workload has a read-to-write ratio of 9:1. CANTO outperforms xFS by a significant margin because



(a) Latency under 50% read. (b) Latency under 90% read. (c) 50% read, concurrency 64.

(d) 90% read, concurrency 64. (e) 50% read, concurrency 256. (f) 90% read, concurrency 256.

Figure 9: Throughput and latency under varied locality.

CANTO can locate copies close to requesters without always going to a remote manager. However, under a write-heavy workload, there are few copies of each object in the system and the CANTO advantage will diminish. Figure 8 shows results from such a workload of equal number of reads and writes. xFS can be viewed as a degenerated case of CANTO where the routing tree is only 1 level tall. This is an example that one should consider access patterns when building CANTO routing trees.

6.2 Performance with Geographic Locality

It is not uncommon that users geographically closer to each other will access more of the same files than widely separated users. This test is designed to explore the performance of different object tracking schemes under geographic locality. The 35 nodes are divided into 8 groups according to their locations. Nodes in the same group share a working set of objects. We say accesses have locality p when a node accesses the local working set (as opposed to an object chosen randomly from the whole set) with probability p.

Figures 9(a)-(b) show the change in read latency when we increase the locality in the workload. These tests are done with light workload so that the latency reflects the time for individual read requests with minimal queuing delay. In both cases, the CANTO variants exhibit much lower latency, indicating shorter request paths.

The advantage of topology-awareness is also reflected in the other four throughput graphs. When the locality in workloads improves, throughput in CANTO increases faster than it does in xFS, because CANTO enjoys the additional benefit of reading from closer copies, while xFS only improves its local cache hit rate. CANTO also benefits from lower consumption of aggregate throughput. The dynamic manager algorithm labeled as SVM yields much lower throughput until locality reaches nearly 100%. This algorithm allows the ownership of an object to move around the nodes accessing the object. However, to locate the manager, a request may have to traverse several links determined by previous accesses. In cases of random accesses across a wide area, the link traversal can be very expensive. When geographic locality is 100%, the ownership of an object is confined to nodes in a much smaller area, closer to the LAN environment it was designed for.

6.3 Performance When Reading Routing State From Disk

In this experiment, we explore the impact of paging the routing bitmap from disk when the bitmap does not fit into main memory. Due to limitations of the testbed, we cannot run a test with enough objects to create a bitmap larger than the 512MB main memory. So we enforce an artificial memory hit rate to bitmap accesses. When the access misses, we emulate the paging process by randomly reading a byte from a 1GB disk file. Here we show the results under the workload with read-to-write ratio of 9:1.

Figure 10 shows the individual request latency under light load. In this case, the frequency of disk access shows little impact, because disk latency is much lower than network latency.

Figure 11 shows throughput results under heavy workload. When the bitmap hit rate is low, the performance is actually decided by the throughput of reading data from disks. Since xFS places all metadata on the manager node,

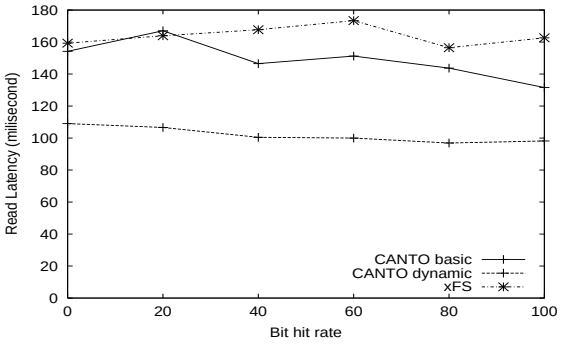


Figure 10: Latency vs. bit hit rate. 90% read and concurrency 1.

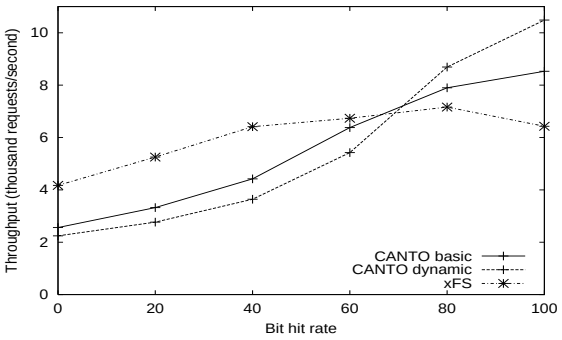


Figure 11: Throughput vs. bit hit rate. 90% read and concurrency 64.

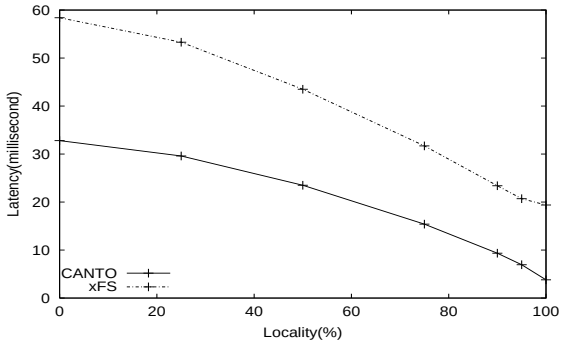


Figure 12: Latency on the Rocketfuel topology.

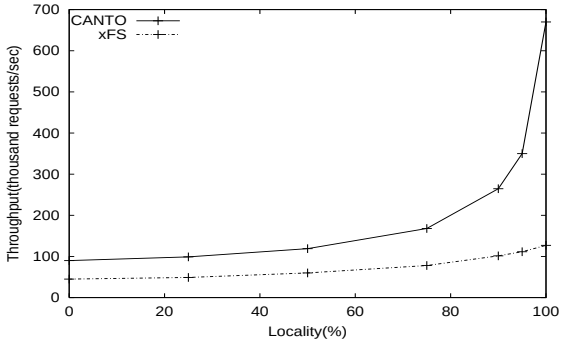


Figure 13: Throughput on the Rocketfuel topology.

it requires fewer bitmap accesses than CANTO does in the hop-by-hop routing, so more xFS requests can be served. When the bitmap hit rate increases, CANTO provides better throughput as the disk is no longer the bottleneck. If more disks are used to store routing state, the bottleneck in disk accesses can be alleviated even when the bitmap hit rate is low.

6.4 Simulation Results of a Larger Topology

We ran simulations to test the routing performance in a larger topology built using data collected by Rocketfuel [18], which maps the topology of ISP router connections. The topology consists of 315 routers and 1,944 links between them. We model the CPU processing time, disk latency and network communications in the simulator.

Figures 12 and 13 compare the performance of the basic CANTO algorithm and xFS in the Rocketfuel topology. We place a CANTO/xFS node in each router. CANTO runs on the physical topology, while xFS uses overlay connections for the all-to-all communication between its nodes. As in previous tests, the workloads have geographic locality from 0% to 100%. 95% of the accesses are read requests. CANTO shows a significant advantage over xFS in both latency and throughput.

6.5 Comparison with DHT-Based Routing

Section 3.1 gives several options for assigning CANTO anchors and deciding CANTO routing graphs. Although CANTO and DHT-based routing are orthogonal technologies in that CANTO can add cache coherence to DHT-based routing, one should not always rely on DHT-based routing: in situations where appropriate, CANTO should achieve better performance with more aggressive exploitation of network-awareness than that is possible with DHT-based routing. To isolate this effect, we compare the following publish/subscribe simulations based on the Rocketfuel topology. The first is based on a modified version of CANTO (that provides publish/subscribe semantics without the strict serialization in the original system) embedded in the Rocketfuel routers and using physical Rocketfuel routes. The CANTO routing trees, therefore, partially mirrors the physical interconnect. The second is Scribe running on Pastry. We use the optimal Pastry algorithm to build the Scribe routing tables. We place a client next to each Rocketfuel router. Each client subscribes to and unsubscribes from a total of 10,000 subjects, with a zipf distribution ($\alpha = 1.0$). The average number of subscriptions per client is 117. The curves labeled "CANTO" and "Scribe" in Figure 14 show the publishing latency of the two simulated systems under varied locality factors (as explained in Section 6.2). We see that although Pastry has its own means for accounting for locality, it can be limited. As the amount of locality in publish/subscribe operations increases, CANTO shows greater relative benefit because it confines messages to smaller areas spanning only the subscribers and publishers.

Section 2.3 explains that the use of explicit 1-bit edges

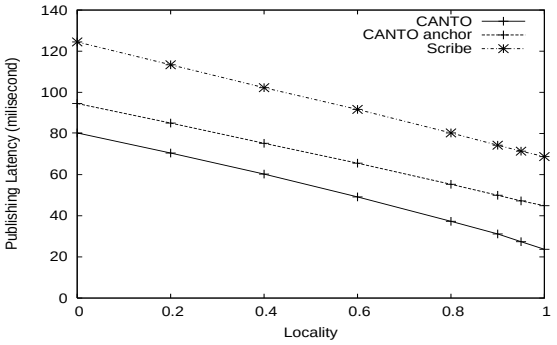


Figure 14: Publishing performance vs. subscription locality.

in CANTO (that may point both toward and away from anchors) allows update-related messages to be confined to network neighborhoods that do not have to include the anchors. As a corollary, publish messages can go from publishers to subscribers without necessarily involving any root (anchor) nodes. If we disable this feature and force publish messages to be always routed through the anchors first, the curve labeled "CANTO anchor" shows the resulting degradation.

6.6 Application Performance

We have implemented two applications of CANTO: a distributed virtual disk and a distributed file system, both of which provide cache coherence for read/write sharing (Section 5).

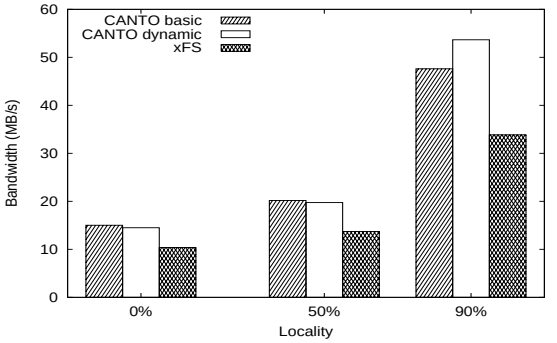


Figure 15: Bandwidth of distributed virtual disk.

Figure 15 presents the aggregate bandwidth achieved by the distributed virtual disk under a workload with read-to-write ratio of 9:1 and queue length 64. We vary the geographic locality as defined in Section 6.2. There are 100,000 data blocks of 4,096 bytes each. Since we do not have root access to PlanetLab machines, we employ a user level implementation instead of the kernel disk driver. The results show the benefit of topology-awareness provided by CANTO.

Table 1 gives the file system test results. The test has several steps, each addressing different types of operations. The numbers indicate the time used to complete each step. This test was run on an 8-node LAN cluster. The machines are 1GHz Pentiums with 1GB memory, connected with 100Mbps ethernet. We used a total of 160,000 files, each 8KB in size, in the tests.

Test step	Time (seconds)
Each node create 1000 directories	4.7
Each node create 20,000 files	102.4
Each node randomly read/write 20,000 files	86.9
Each node sequentially read all files	1640

Table 1: File system performance.

7 Related Work

We have already discussed the relationships among manager-based systems [2, 9, 20, 6, 10, 16], DHTs [14, 15, 19, 21], and CANTO in Section 1. Most of the manager-based cluster systems owe similarities to the data location mechanisms explored in the original shared virtual memory (SVM) work [10]. One of the alternatives examined in the SVM system is the "dynamic distributed manager algorithm" with "distributed copy sets." Under this algorithm, the location of the data is tracked by a tree of processors rooted at an owner processor. All tree edges are bidirectional. Any node in the tree can locate a copy of the data to satisfy read requests. Invalidations must start at the root and propagate downward to reach all nodes. If hints fail to locate a node in the tree, the system resorts to broadcasts to satisfy requests. The system runs in a cluster environment that has a simple topology and the tree is formed strictly based on the sequence of operations performed on the data. A goal of CANTO is to build a network-aware object tracking system for a more complex topology. The routing tree nodes in CANTO can be routers that do not necessarily initiate requests on their own and the shape of the routing tree is formed based on the network topology. In terms of details, the CANTO routing tree edges are not uniformly bidirectional: instead, the invariant in CANTO is that there is a path of directed 1-bit edges leading from any active node to all other active nodes (with the anchor being one of the active nodes). This invariant allows CANTO to more fully exploit topological locality so, for example, unlike existing systems, invalidation messages do not need to always reach an anchor or a manager, nor does CANTO need to resort to broadcast mechanisms.

We have already discussed the Globe [3] and Scribe [5] systems. Objects in Globe are mobile but, unlike those in CANTO, are unchangeable. Scribe implements DHT-based publish/subscribe. In contrast, CANTO provides cache coherence for read/write operations.

OceanStore [8], a wide-area storage system, proposes a two-level lookup process: a fast "local" lookup may fail and is followed up with a slower and more restrictive "global"

lookup. These two stages are based on different algorithms. CANTO employs a single lookup algorithm that seeks to both exploit locality and retain placement flexibility.

Invalidations in OceanStore, publishes in Scribe, and object movements in Globe are performed in a way analogous to that employed by the SVM system described above and they all must reach a root node. To the extent possible, CANTO strives to isolate invalidation messages within a topological locality without always resorting to a root. CANTO copes with persistence, recovery, and reconfiguration of routing state, issues not necessarily fully addressed in the above systems.

Ivy is a read-write file system built on top of a DHT-based storage system [11]. Instead of tracking the locations of the fresh data, the system searches the logs of all writers. The system is designed to work for a relatively small number of writers and the system has a relatively loose coherence semantics.

8 Conclusions

In this paper, we show the importance of network topology awareness in building object tracking and routing facilities in a wide area network. We have presented an object tracking system that possesses both strong coherence semantics and network topology awareness. Experiments with a deployment of the system on a real-world Internet overlay show substantial benefits of the system compared to existing approaches.

References

[1] ANDERSON, D. G., BALAKRISHNAN, H., KAAWHOEK, M. F., AND MORRIS, R. Resilient Overlay Networks. In *Proc. of the Eighteenth Symposium on Operating Systems Principles* (October 2001).

[2] ANDERSON, T., DAHLIN, M., NEEFE, J., PATTERSON, D., ROSELLI, D., AND WANG, R. Serverless Network File Systems. *ACM Transactions on Computer Systems 14*, 1 (Feb. 1996), 41–79.

[3] BAGGIO, A., BALLINTIJN, G., VAN STEEN, M., AND TANENBAUM, A. S. Efficient tracking of mobile objects in Globe. *The Computer Journal 44*, 5 (2001), 340–353.

[4] BALAKRISHNAN, H., DRUSCHEL, P., HELLERSTEIN, J., KAASHOEK, M. F., KARGER, D., KARP, R., KUBIATOWICZ, J., LISKOV, B., MAZIERES, D., MORRIS, R., SHENKER, S., AND STOICA, I. The IRIS ITR proposal. http://iris.lcs.mit.edu/-proposal.html, 2002.

[5] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., , AND ROWSTRON, A. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications) 20*, 8 (October 2002).

[6] FEELEY, M. J., MORGAN, W. E., PIGHIN, F. P., KARLIN, A. R., LEVY, H. M., AND THEKKATH, C. A. Implementing Global Memory Management in a Workstation Cluster. In *Proc. of the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 201–212.

[7] KARGER, D., LEHMAN, E., LEIGHTON, F. T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. 29th STOC*, pp. 654–663.

[8] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS2000)* (November 2000).

[9] LEE, E. K., AND THEKKATH, C. E. Petal: Distributed Virtual Disks. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996), pp. 84–92.

[10] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 7*, 4 (Nov. 1989), 321–359.

[11] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: A Read/Write Peer-to-Peer File System. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (December 2002).

[12] PERKINS, C. RFC 2002: IP mobility support, Oct. 1996.

[13] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. First Workshop on Hot Topics in Networks (HotNets-I)* (October 2002).

[14] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM 2001* (August 2001), pp. 161–172.

[15] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (November 2001), pp. 329–350.

[16] SAITO, Y., BERSHAD, B., AND LEVY, H. Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service. *ACM Transactions on Computer Systems 18*, 3 (August 2000), 298–332.

[17] SPALINK, T., KARLIN, S., PETERSON, L., AND GOTTLIEB, Y. Building a Robust Software-Based Router Using Network Processors. In *Proc. of the Eighteenth Symposium on Operating Systems Principles* (October 2001).

[18] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP topologies with rocketfuel. In *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM-02)* (New York, Aug. 19–23 2002), vol. 32, 4, ACM Press, pp. 133–146.

[19] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM 2001* (August 2001).

[20] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: A Scalable Distributed File System. In *Proceedings of the ACM Sixteenth Symposium on Operating Systems Principles* (Oct. 1997).

[21] ZHAO, B., KUBIATOWICZ, J., AND JOSEPH, A. An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD 01/1141, University of California at Berkeley, April 2001.