

Are Virtualized Overlay Networks Too Much of a Good Thing?

Pete Keleher, Bobby Bhattacharjee, and Bujor Silaghi

Department of Computer Science
University of Maryland
College Park, MD 20742, USA
{keleher,bobby,bujor}@cs.umd.edu

Abstract. The majority of recent high-profile work in peer-to-peer networks has approached the problem of location by abstracting over object lookup services. Namespace virtualization in the overlay layer provides load balance and provable bounds on latency at low costs.

We contend that namespace virtualization comes at a significant cost for applications that naturally describe their data sets in a hierarchical manner. Opportunities for enhancing browsing, prefetching and efficient attribute-based searches are lost. A hierarchy exposes relationships between items near to each other in the topology; virtualization of the namespace discards this information even if present at client, higher-level protocols.

We advocate encoding application hierarchies directly into the structure of the overlay network, and revisit this argument through a newly proposed distributed directory service.

1 Introduction

Peer-to-peer (P2P) networks have recently become one of the hottest topics in OS research [1–8]. Starting with the explosion of popularity of Napster, researchers have become interested because of the unparalleled chance to do relevant research (people might actually use it!), and the naive approach of many of the first P2P protocols deployed.

The majority of most recent high-profile work has described middleware that performs a single task: distributed object lookup. This seemingly simple function can be used as the basic building block of more complex systems that perform a variety of sophisticated functions (file systems, event notification system, etc.).

P2P networks differ from more conventional infrastructures in that the load (whether expressed as CPU cycles, data storage costs or packet forwarding bandwidth) is distributed across participating peers. This load should ideally be balanced, as the load is in some sense the “payment” for participating in the network. Overloading some peers while letting others off without performing any work towards the common good is clearly unfair.

The approach many recent systems [6, 4, 7, 5] have taken towards ensuring load balance is to virtualize data item keys by creating the keys from one-way

hashes (SHA-1, etc.) of the item labels. The peer node ID's are similarly encoded, and data items are mapped to "closest" nodes by comparing keys and hashed node ID's. We refer to this as *virtualization* of the namespace. By contrast, a "non-virtualized" system is one where data items are served by the same nodes that export them.

A virtualized approach helps load balance because data items from one very popular site will be served by different nodes; they will be distributed randomly among participating peers. Similarly, routing load is distributed because paths to items exported by the same site are usually quite different.

Just as importantly, virtualization of the namespace provides a clean, elegant abstraction of routing, with provable bounds on routing latency.

The contention of this position paper is that this virtualization comes at a significant cost, as described below:

1. *Virtualization destroys locality* - By virtualizing keys, data items from a single site are not usually co-located, meaning that opportunities for enhancing browsing, prefetching, and efficient searching are lost.
2. *Virtualization discards useful application-specific information* - The data used by many applications (file systems, auctions, resource discovery) is naturally described using hierarchies. A hierarchy exposes relationships between items near to each other in the hierarchy; virtualization of the namespace discards this information.

The rest of this paper elaborates on these points and outlines an alternative approach.

To be clear, the environment assumed in this paper is that of a set of cooperating, widely-separated peers, running as user-level processes on ordinary PC's or workstations. Peers "export" data, and keys are "mapped" onto overlay servers. The set of peer nodes that export data is also the set of overlay servers. A "node" is a process participating in the system.

2 Locality is a *Good Thing*

The first form of locality with which we are concerned is spatial locality. Users who access d_i are more likely to also access d_{i+1} than some arbitrary d_j . Consider web browsing: a given page might require many nearby items to be accessed, and the next page accessed by a user is likely to be on the same site as well.

Virtualization of this process loses several opportunities for performance improvement. First, the route to the exporting site only has to be discovered once in a non-virtualized system. Subsequent accesses to a second data item in logical proximity can follow the same route and shortcuts would be employed. In a virtualized system, there will likely be nothing in common between the two routes. Second, an exporting site (or the access initiator) might choose to prefetch nearby data in a non-virtualized system. Prefetching, when it works, enables the system to hide the latency of remote accesses. While prefetching can be made to work with a virtualized namespace, it is much more difficult. For example,

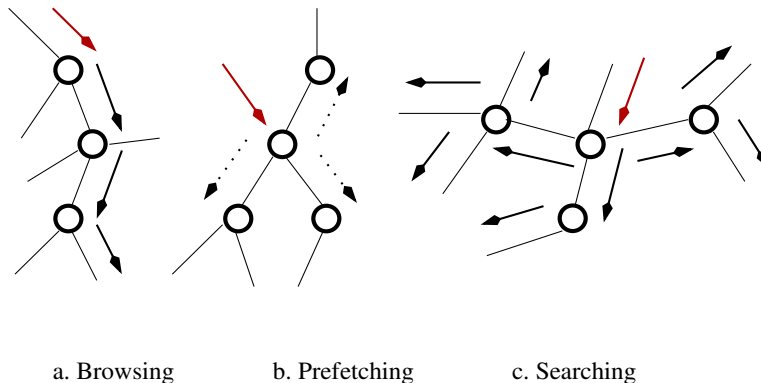


Fig. 1. Browsing, prefetching and searching are examples of operations where object relationships can be exploited by integrating them with the routing procedure in the overlay network layer. This allows subsequent steps, indicated by arrows, to be performed in $O(1)$ as opposed to $O(\log N)$, etc. for virtualized namespace schemes.

CFS [9], a cooperative file system built on top of Chord [6], can prefetch file blocks. A peer receiving a request for block i of a file can prefetch the second by locally reconstructing the virtualized name for block $i + 1$ and sending a prefetch message to the site serving it. However, each such prefetch requires an object lookup which translates to additional messages in the network. Worse, prefetching blocks is relatively easy because the name of the nearby object (block $i + 1$) is easy to predict. However, the names of nearby items on the exporting site are not easy to predict, and prefetching could probably only be accomplished via an application overlay that indexes exporting sites.

Current approaches also fail to exploit temporal locality as much as they might. None of Chord, CAN [7], Pastry [5], or Tapestry [4] currently use caching. Repeated accesses by one site to the same data item require repeated traversals through the overlay network. However, caching could easily be added to these systems, and is used in some applications built on top of these systems (e.g., Past [10], CFS). Further, some systems (most notably Pastry) attempt to exploit locality in charting a route through the overlay network's nodes. The result is that the total IP hop count may be only a small constant higher than a native IP message.

Note that there is an implicit assumption of spatial locality in all of the virtualization work. Virtualization distributes load well only assuming that the only form of locality present is spatial, not temporal. Stated another way, virtualization can not improve load balance in the presence of single-item hot-spots; it can only distribute multiple (possibly related) data items across the network.

By contrast, current systems use replication both to provide high availability and to distribute load when single items become hot spots.

3 Searching

Most distributed object stores require some search capability in order to be used effectively. We distinguish two types of searching: indexing and keyword/attribute searching. “Indexing” refers to indexing entire documents, e.g. Google’s index of the web. Indexing of documents served by a distributed overlay system requires local indexes to be created, combined, and then served, presumably through the same overlay system, although this could also be done at a higher level. The difficulty of indexing is not affected by whether the namespace is virtualized; it is a hard problem for any distributed system.

Section 2 considered spatial locality in terms of browsing and prefetching. A similar argument can be made regarding attribute searching (see Figure 1).

Assume that we wish to search for “red cars”, where ‘red’ and ‘car’ are encoded as attributes of certain documents. Virtualized namespaces do not encode attributes in the overlay structure, so this search could only be accomplished by: (i) visiting every node and thus flooding the network or (ii) resorting to some higher-level protocol that would capture additional object relationships. Such relationships (topological constraints like hierarchies for instance) are orthogonal to the overlay layer where the lookup routing is performed. Therefore virtualized namespace approaches cannot benefit from such extra information in optimizing the routing procedure.

We discuss how embedding attributes in the overlay structure of a non-virtualized system allows efficient attribute searching in Section 5.

4 Adding Information Back In

There are two approaches to adding application-specific information and support for locality back into a virtualized system: use of higher-level application layers, and eliminating virtualization entirely. We discuss the former here, and one approach to the latter in the next section.

Support for locality in the query stream can be added back at higher levels. As an example, both Past and CFS cache data along paths to highly popular items. The advantage of doing so at the file system layer rather than the routing layer is that both location information, and the file’s data itself, are cached together. If address caching were performed at the routing level, file caching would be less effective.

File system prefetching can also be accomplished at this level. For example, one could prefetch the rest of a directory after one file is accessed by (1) deriving the file’s directory name from the target file’s name, (2) routing a message to the directory object, (3) reading the directory object to get the set of other files in the directory, and then (4) sending prefetches to each of those files.

However, not only is this inefficient, but it only makes use of information about accesses to a single file. Consider how a system with a virtualized namespace would support a policy that only prefetches entire directories if two or more files of the directory are accessed within a short time.

5 Eschewing Virtualization

Consider the types of applications that are being built on top of the overlay networks discussed so far: file systems, event notification systems, distributed auctions, and cooperative web proxies. All of these applications organize their data hierarchically, and any locality in these applications is local in the framework of this hierarchy. Browsing the hierarchy is, therefore, the only way of extracting and exploiting locality. Yet, this information is discarded by virtualized namespaces.

Another approach is to encode this hierarchy directly into the overlay layer. While this paper is not about TerraDir, we discuss it as an example approach that addresses some of the shortcomings discussed above. A TerraDir [8] is a non-virtualized overlay in the form of a rooted hierarchy that explicitly codifies the application hierarchy. By default, routing is performed via tree traversal, taking $O(\log N)$ hops in the worst case¹. Availability, load balance, and latency are all addressed further by caching and replication. The degree to which a node is replicated is dependent on the node’s level in the tree. This approach helps load balance and only adds a constant amount of overhead per node, regardless of the size of the system.

TerraDir provides comparable performance to the other systems (probably somewhat better latency because of the caching, probably a bit worse load distribution), but leaves application locality and hierarchical information intact.

Locality is retained because a given data item is mapped to the node that exports it, rather than to another randomized host. Not only does this save a level of indirection, but co-located items are mapped to the same locations, meaning locality can be recognized and exploited without network communication. Note that replication is per-node. All items exported by a node are replicated together, so any replica can perform prefetching.

Caching addresses both spatial and temporal locality. Repeated accesses to the same remote object are serviced by a local cache if caching of data is turned on. Otherwise, the cache provides the network address of the exporting node, limiting routing to a single hop through the overlay network.

Accesses to items “near” each other in the application hierarchy are handled efficiently because they are also near each other, or co-located, in the overlay network. Hence, the number of hops in the overlay network are again minimized.

Data item keywords are explicitly coded into the overlay hierarchy, so searching for keywords is handled efficiently. For example, consider searching for red cars in the hierarchy shown in Figure 2. The query would be of the form “/vehicles/cars/red/*”, and would be routed to the smallest subtree on the left side of the figure. The wildcard will then cause the query to be split and to flood that subtree. However, the rest of the hierarchy, aside from the path from the query initiator to the “cars/red” subtree, is untouched. By contrast, searching for “red cars” can only be accomplished efficiently via some higher-level service in a virtualized system. To be fair, note that the query is handled efficiently only because

¹ Assuming a relatively well-balanced tree.

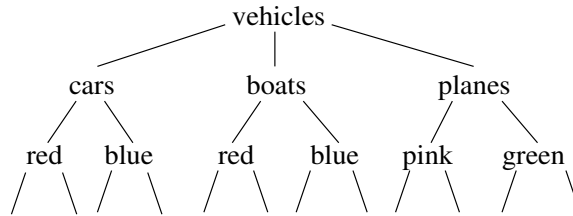


Fig. 2. An example TerraDir. Searching for “red cars” is more efficient than searching for “anything red”.

the query structure matches the hierarchy’s structure. Searching for “anything red” would cause all leaves to be visited. This problem is addressed by allowing “views” to be dynamically materialized. A client that expects to make multiple queries with a different structure (e.g. “all red things”, then “all blue things”, etc.) inserts a *view query* into the system. The view queries specifies an ordering on the set of attributes², which is used to build a new overlay hierarchy. Building the new hierarchy requires the entire tree to be visited once; subsequent queries will be handled efficiently.

The TerraDir approach has at least two other important advantages. First, maintenance overhead is significantly less. The virtualized approaches generally require $O(\log N)$ operations to allow a node to leave or join the overlay, whereas these operations require only a constant number of operations under TerraDir.

Finally, TerraDir nodes “maps” the key of a node back to that same node, meaning that the data item and its mapping are not distributed across administrative boundaries.

6 Summary

Distributed lookup services using virtualized namespaces can be important building blocks for building sophisticated P2P applications. Namespace virtualization provides load balance and tight bounds on latency at low cost.

In doing so, however, it discards potentially useful information (application hierarchies) and object relationships (proximity within the hierarchy). This is not always a problem: certain types of functionality are more efficiently provided at higher layers (this is merely the end-to-end argument [11]). However, many applications can benefit from increased functionality in the lookup layer.

We advocate encoding application hierarchies, where needed, directly into the structure of the overlay network. This approach allows systems to exploit locality between objects and to provide searching without centralized indexing or flooding.

² View queries can also name *tag functions*, which can be seen as dynamic attributes synthesized from the static attributes.

References

1. Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing nearby copies of replicated objects in a distributed environment. In: Proc. of the ACM Symposium on Parallel Algorithms and Architectures, Newport, RI (1997) 311–320
2. Petersen, K., Spreitzer, M., Terry, D.B., Theimer, M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Symposium on Operating Systems Principles. (1997) 288–301
3. Kubiawicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., 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. (2000)
4. Zhao, B., Kubiawicz, K., Joseph, A.: Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California at Berkeley Technical Report (2001)
5. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms. (2001)
6. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the ACM SIGCOMM '01 Conference, San Diego, California (2001)
7. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content addressable network. In: Proceedings of the ACM SIGCOMM 2001 Technical Conference. (2001)
8. Bhattacharjee, B., Keleher, P., Silaghi, B.: The design of TerraDir. Technical Report CS-TR-4299, University of Maryland, College Park, MD (2001)
9. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles, Chateau Lake Louise, Banff, Canada (2001)
10. Rowstron, A., Druschel, P.: Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles. (2001)
11. Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. *Computer Systems* **2** (1984) 277–288