Practical Locality-Awareness for Large Scale Information Sharing*

Ittai Abraham[1], Ankur Badola[2], Danny Bickson[1], Dahlia Malkhi[3], Sharad Maloo[2], and Saar Ron[1]

[1] The Hebrew University of Jerusalem, Jerusalem, Israel
{ittaia, daniel51, ender}@cs.huji.ac.il
[2] IIT Bombay, India
{badola, maloo}@cse.iitb.ac.in
[3] Microsoft Research Silicon Valley and The Hebrew University of Jerusalem, Israel
dalia@microsoft.com

Abstract. Tulip is an overlay for routing, searching and publish-lookup information sharing. It offers a unique combination of the advantages of both structured and unstructured overlays, that does not co-exist in any previous solution. Tulip features locality awareness (stretch 2) and fault tolerance (nodes can route around failures). It supports under the same roof exact keyed-lookup, nearest copy location, and global information search. Tulip has been deployed and its locality and fault tolerance properties verified over a real wide-area network.

1 Introduction

Driven by the need to bridge the gap between practically deployable P2P systems, which should be easy and robust, and academic designs which have nice scalability properties, we present the Tulip overlay. The Tulip information sharing overlay obtains a combination of features not previously met simultaneously in any system. In a nutshell, these can be characterized as follows:

Locality-awareness: The algorithms for searching and retrieving information are designed to provably contain every load as locally as possible. Formally, this is expressed using the standard network-theoretical measure *stretch*, which bounds the ratio between routes taken in the algorithm and optimal routes. Formally, the Tulip overlay guarantees stretch-2 routing.

Flexibility and Simplicity: All protocols have firm, formal basis, but intentionally accommodate fuzzy deployment which applies optimizations that deviate from the theory, in order to cope with high churn and scalability.

Diverse tools: Tulip addresses under the same roof exact-match keyed lookup, nearest object location, and global data search.

Experimentation: In addition to formal proofs of locality and fault tolerance we analyze Tulip's performance with real measurements on a real planetary-scale

* Work supported in part by EC *Evergrow*.

M. Castro and R. van Renesse (Eds.): IPTPS 2005, LNCS 3640, pp. 173–181, 2005.

deployment. Tulip is deployed and tested over PlanetLab. Its locality awareness and fault tolerance properties are evaluated in a WAN setting. Furthermore, experience gained from practical deployment is fed back in Tulip to the formal design.

Tulip adopts the successful space-to-communication tradeoff introduced by Kelips [4], which allows nodes to maintain links to many, but not to all other nodes, and achieve highly efficient information dissemination paths. In Tulip, each node maintains roughly $2\sqrt{n}\log n$ links, where n is the number of nodes. Routes take 2 hops. Search or event data can be disseminated to $O(\sqrt{n})$ nodes, and retrieved from $O(\sqrt{n})$ nodes.

We believe this tradeoff is the right one for P2P overlays. In terms of space, even a large system of several millions of nodes requires storing only several thousands node addresses, which is not a significant burden. That said, this does not lead us to attempt at maintaining global information as in [3]. Indeed, we maintain sufficient slack to tolerate a large degree of stale and/or missing information. As a result of this design choice, Tulip exhibits extremely good fault tolerance (see Section 4). Furthermore, this slack also enables *static resilience*, which means that even as the system undergoes repair it can continue routing data efficiently. Some previous DHTs like Kademlia [5] and Babmboo [6] appear to cope well with churn with a lower node degree and more rigid structure. However, we believe that having $O(\sqrt{n})$ links with a semi-structured two hop network may give a very high level of resiliency.

Tulip enhances the Kelips approach in a number of important ways, detailed henceforth. The first feature in Tulip is **locality awareness**. Building self maintaining overlay networks for information sharing in a manner that exhibits locality-awareness is crucial for the viability of large internets.

Tulip guarantees that the costs of finding and retrieving information are proportional to the actual distances of the interacting parties. Building on the formal foundations laid by Abraham et al. in [1], Tulip provides provable stretch 2 round-trip routing between all sources and destinations [1]. Tulip extends the formal algorithm in [1] with methods that accommodate changes in the network. These include background communication mechanisms that bring links up to date with provably sub-linear costs.

The second feature of Tulip is its **flexibility and simplicity**. Structured p2p overlays often appear difficult to deploy in practical, Internet-size networks. In particular, they are sensitive to changes and require substantial repair under churn. They lack flexibility in that they require very accurate links in order to operate correctly. And faced with high dynamism, they may break quite easily.

By maintaining $O(\sqrt{n}\log n)$ links at each node and a simple two hop design, Tulip has sufficient redundancy to maintain a good level of service even when some links are broken, missing or misplaced. A multi-hop algorithm similar to Kelips [4] allows routing around failed or missing links with $O(1)$ communication

[1] The standard definition of stretch, as in [1], looks at source-destination routing but in a DHT it is natural to examine round-trip routing since the source requires a reply from the target.

costs. Furthermore, the repair procedures can be done in the background, while heuristics keep Tulip's service quality even while it is under repair.

The third feature of our system is its support of **diverse tools** for information sharing. This goal stems from our vision of a convergence of technologies empowering network leaf-nodes. These technologies include overlay networks supporting Grid and p2p file sharing, web caching, and large scale content delivery services. Though these are different services, the overlays that support them are converging toward a common set of protocols. The Tulip routing overlay can be utilized as an overlay for keyed lookup, for finding nearest copies of, replicated objects, for event notification and for global searching.

We have built a **real deployment** of the Tulip overlay and have conducted experimentation on wide area networks (WANs). All of our protocols are deployed and tested extensively over the PlanetLab WAN test-bed. In particular, Tulip's locality behavior, its stretch factor, distance measurements and fault tolerance are all ascertained over a real-life, planetary-wide network. To the best of our knowledge, our stretch performance data are the first to be measured over a real WAN, not via synthetic simulation. We also assess Tulip's behavior under intentional and unintentional churn.

2 Formal Foundations

The Tulip system builds on the locality-aware compact routing algorithm of Abraham et al. in [1]. It uses $O(\sqrt{n}\log n)$ space per node, where n is the number of nodes in the system. It provides a 2-hop routing strategy whose cost over optimal routing (the *stretch*) is at most 2. Continuous background gossip mechanism with a reasonable overhead is used to maintain and update the system and guarantee quick convergence after changes in the system.

Let $d(s,t)$ denote the communication cost between nodes s and t. It is natural to assume that $d()$ forms a metric space. However, to be precise, our lookup stretch result requires only that $d()$ is symmetric, or that it upholds the triangle inequality. In addition, the analysis of message complexity of the join algorithm and the protocol for finding nearest copies of data assume growth bounded densities, defined as follows. A growth-bound limits the number of nodes in a ball of radius $2r$ by a constant multiple of the number of nodes within radius r.

Vicinity balls. For every node $u \in V$, let the vicinity of u be the set of $\sqrt{n}\log n$ closest nodes to u according to $d()$, breaking ties by lexicographical order of node names.

Coloring. Our construction uses a partition of nodes into $\sqrt{n}$ color-sets, with the following two properties:

(i) Every color-set has at most $2\sqrt{n}$ nodes.
(ii) Every node has in its vicinity at least one node from every other color-set.

Each node belongs to one of the color groups determined by using a consistent hashing function to map node's identifier (IP address and port number) to one of the $\sqrt{n}$ values. This mapping is done by taking the first $\log\sqrt{n}$ bits of the

hash value. We denote by $c(u)$ node u's color. The use of cryptographic hash function such as SHA-1 ensures that the expected number of nodes in each group is around $\sqrt{n}$, and is under $\sqrt{n}\log n$ with high probability.

Routing information. Each node u maintains information classified under:

• *Vicinity list*: From each of the other color groups in the system, node u maintains information about the closest $\log n$ nodes of a particular color.
• *Color list*: A list containing information about all nodes belonging to the same color as u, i.e, to the color-set $c(u)$.

Each entry also carries an additional field of network distance. Each of the lists is sorted based on the relative distance value from the node.

Keyed Lookup. The lookup tool supports exact-match keyed lookup and routing for objects or nodes whose names are known precisely. It guarantees locating any target with lookup stretch of at most 2, and with up to 2 lookup hops.

An object is stored on the node whose identifier is the longest prefix of the object's hash value. Objects are also mapped to colors by taking the first $\log \sqrt{n}$ bits of their hash. Given a source node s that is looking for an object o with color $c(o)$ that is stored in node t:

• *First hop*: Node s routes to the node w in s's vicinity list that has the same color as the object $c(w) = c(o)$, and whose identifier is closest to the object's hash. If this node contains the object then the lookup has stretch 1.
• *Second hop*: Otherwise, using w's color list, s routes to node t (this is possible since $c(o) = c(w) = c(t)$). In this case we have $d(s,w) \leq d(s,t)$ and from symmetry the cost of the path $s \rightsquigarrow w \rightsquigarrow s \rightsquigarrow t \rightsquigarrow s$ is at most *twice* the cost of the path $s \rightsquigarrow t \rightsquigarrow s$ (see Figure 1).

Note that, the above scheme is *iterative*, and achieves stretch 2 without requiring triangle inequality. A *recursive* version would give stretch 2 but require triangle inequality, without requiring symmetry.

Finding nearest copy. This mechanism allows objects to be stored on any node the designer wants. Moreover, several copies of the same object may exist on different nodes. Assuming latencies form a metric space, we guarantee to retrieve the copy closest to the initiator of the searching node, with lookup stretch of at most 4, and with up to 2 hops.

Let node x store a replica of object o. A pointer of the form $\langle o \rightarrow x \rangle$ is stored in the following nodes:

• *Vicinity pointers:* All nodes u such that x is in u's vicinity list store a pointer $\langle o \rightarrow x \rangle$. Under the growth bound assumption, only $O(\sqrt{n})$ nodes will store such a pointer.
• *Color pointers:* All nodes u such that $c(u) = c(o)$ store a pointer $\langle o \rightarrow u(o) \rangle$ where $u(o)$ is the name of the node closest to u that stores a replica of o.

Lookup uses the pointers to shortcut directly to a replica. If the source does not have a direct pointer to the desired object it routes to the node w in its vicinity such that $c(w) = c(o)$. In such a case, node w will have a pointer to the closest replica from w.

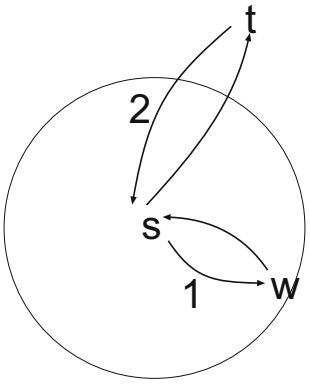

Fig. 1. Example of a 2 hop, stretch 2 round-trip path from source s to destination t and back

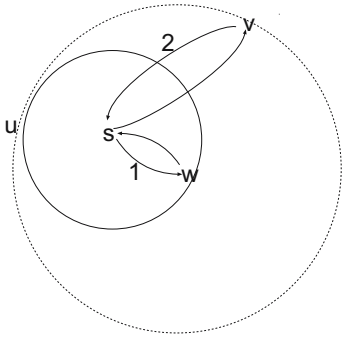

Fig. 2. Example of 2 hop, stretch 4, nearest copy search from s to v and back

• *Analysis.* Given source s searching for object o, let u be the closest node to s storing a replica of o, let w be the node in the vicinity of s such that $c(w) = c(o)$ let v be the closest node to w storing a replica of o. Then $d(w, v) \leq d(w, u)$ and by triangle inequality $d(s, v) \leq d(s, w) + d(w, v) \leq 3d(s, u)$, summing up and using symmetry $d(s, w)+d(w, s)+d(s, v)+d(v, s) \leq d(s, u)+d(s, u)+3d(s, u)+3d(s, u)$, hence the ratio between the cost of lookup and the cost of directly accessing the closest replica (stretch) is at most 4 (see Figure 2).

Global information search. This tool builds a *locality aware quorum system.* Information can be published to a global shared memory and later users can perform arbitrary search queries on all the published information. The search mechanism is locality aware, it requires communication only with nodes in the vicinity of the query.

Publishing an object o is done by storing information about o on all the nodes whose color is the closest to o's hash value. Each node may either store

the full o content, or summary data used for searching o, along with a pointer to the actual stored location. This creates roughly $\sqrt{n}$ replicas of the information.

Global searching is done in a locality aware manner. An initiator sends a query only to the nodes that are in its vicinity list. The computation of a query is maintained locally since each search involves only the $\sqrt{n}$ closest nodes. This is the first read/write quorum system whose read operations are locality aware.

3 Maintaining Locality Under Churn

Considerable effort is invested in Tulip's deployment in order to deal with the dynamics of scalable and wide spread networks. This includes protocols for node joining and deletion, and a background refresh mechanism that maintains locality under churn. Surprisingly, under reasonable assumptions, all of these mechanisms have sub-linear complexity. Our deployment also entails multi-hop query routing to cope with churn simultaneously with ongoing repair. The evaluation of this heuristical protocol is done experimentally.

Joining: A joining node requires one existing contact node in the system. The mechanism for obtaining a contact node can be a web site or a distributed directory such as DNS. Our approach for handling joins is for the joiner to first acquire a somewhat rough initial vicinity. Then, through normal background refresh mechanism (detailed below), the joiner gathers more accurate information about its vicinity and its color list.

More specifically, a joiner u first queries its contact point for the list of nodes in its vicinity. From this list, u selects a random node x. It then finds a node w from x's color list that is closest to u. Under reasonable growth bounded density assumptions w's vicinity has a sizable overlap with the u's vicinity. Node u adopts w's vicinity as its own initial vicinity, and informs its vicinity about its own arrival.

The communication complexity of the approximate closest-node finding and the establishment of an initial vicinity is $O(1)$ and $O(\sqrt{n}\log n)$ computational complexity.

Deletion: A departing or a failed node gradually automatically disappears from the routing tables of all other nodes, once they fail to communicate with it. Naturally, the departure of a node also means the loss of the data it holds. Clearly, any robust information system must replicate the critical information it stores. We leave out of the discussion in this short paper such issues.

Refresh mechanisms: Existing view and contact information is refreshed periodically within and across color groups. During each cycle, a node re-evaluates the distance of some nodes in its two lists (vicinity and color), and refreshes entries in them. Formally, these mechanisms maintain the following property: An individual node that misses information or has incorrect information (e.g., this is the case of a new joiner) learns information that significantly improves its vicinity and color list with $O(1)$ communication overhead and $O(\sqrt{n}\log n)$ computation overhead.

All our methods have sub linear communication complexity of $O(1)$ and $O(\sqrt{n}\log n)$ computational complexity. The three methods used in each cycle for refresh are as follows:

• *Vicinity list merging:* Node u chooses a random node x in its vicinity list and requests for x's vicinity list, while sending its own vicinity list to that random node (a combined push and pull flat model gossip). Both nodes merge the two vicinities, while keeping the list sorted according to distance and maintaining (if possible) at least one member from each existing color in the list. Intuitively, due to the expected overlap between the vicinities of close nodes, this step provides for quick propagation of knowledge about changes within the vicinity. This mechanism is quick and efficient in practice. However, formally it cannot guarantee by itself that nodes obtain all relevant vicinity information.
• *Same color merging:* Node u contacts a random node x from its color list and and requests for x's color list, while sending its own color list to that random node. Again, both nodes merge the two color lists.
• *Failed nodes detection:* When a failed node is detected (a node that had failed to respond to an outgoing communication), that node is immediately removed from all active nodes' lists. That node is then inserted into a failed nodes list, which also holds information about the failure detection time (a node's "death certificate"). This list is being propagated in two methods:

1. *Passive fault tolerance mechanism:* a node which is refreshing its color or vicinity list also sends its failed nodes list with its request and receives the other node's failed nodes list with the response. Both nodes then merge both lists, and remove all new found failed nodes from all their active nodes lists.
2. *Active fault tolerance mechanism:* a node that sends a routing info request to a node x also pushes its failed nodes list as a part of the request. Before processing the request, the receiving node merges its own failed nodes list with the received list, and removes the new found failed nodes from all active nodes lists. This somewhat prevents x from sending a next hop route data which includes a newly detected failed node.

A node is removed from the failed nodes list only after a period of time which is greater then the estimated gossip propagation time in the network.

Multi-Hop Query Routing: The scale and wide spreading of the systems we envision implies that the information held at nodes' routing lists at any snapshot in time may contain inaccessible nodes, failed links, and inaccurate distances. Although eventually the refresh mechanisms repair such errors, the system must continue routing data meanwhile. To this end, we adopt similar, heuristic strategies as in Kelips [4] to accommodate changes, while enhancing them with locality consideration, and in addition, evaluating them with real wide-area experimentation (in the next section).

Given a source node s that is looking for an object that is stored in node t, the two heuristics employed are as follows:

• If s cannot contact any node with color $c(t)$ from its vicinity list, then it contacts a random node x in the vicinity list and forwards the query for x to handle.
• If during a lookup, an interim node w with the target's color $c(t) = c(w)$ does not have t in its color list, then w responds with sending the details of a node v from its vicinity list, drawn randomly with preference to closer nodes.

4 Experimental Results

The Tulip client is implemented in C++ and the overlay is fully operational. Tulip is deployed in 220 nodes over the PlanetLab wide-area testbed [2] as of October 2004.

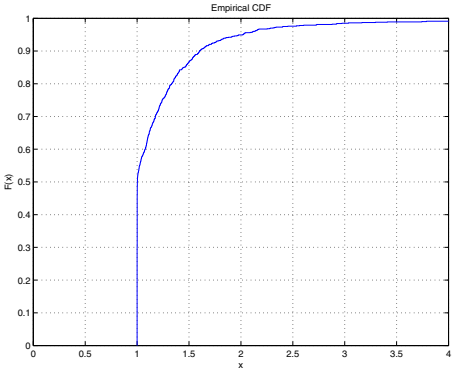

Fig. 3. Cumulative density of lookup stretch

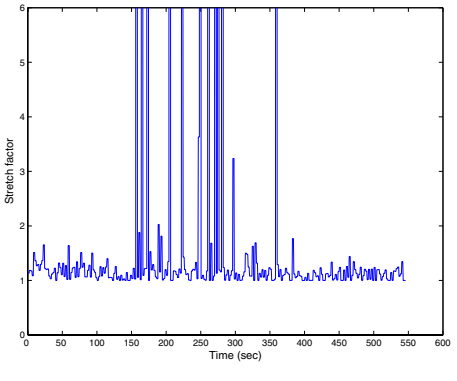

Fig. 4. Average stretch over time while randomly killing half the nodes at time 150

Figure 3 depicts the actual routing stretch experienced in our deployed system. The graph plots the cumulative density function of the stretch factor of one thousand experiments. In each experiment, one pair of nodes is picked at

random and the routing stretch factor between them is measured. The measured stretch factor is the ratio between lookup duration and the direct access duration: The lookup duration is the time a lookup takes to establish connections, reach the destination node storing the object via the Tulip overlay, and return to the source. The direct access duration is the time it takes to form a direct (source to destination) TCP connection and to get a reply back to the source.

The graph shows that about 60 percent of the routes has stretch 1, and therefore experience nearly optimal delay. Over 90 percent of the routes incur stretch lower than 2, and stretch 3 is achieved in nearly 98 percent of the routes. These results are comparable, and to some extent better, than the simulation stretch results provided for Pastry [7] and Bamboo [6].

The graph also demonstrates that due to dynamic nature of the network and due to fuzziness, stretch 4 is exceeded in about one percent of the cases. The graph is cut at stretch 4, and thus excludes a very small number of extremely costly routes; these do occur, unfortunately, in the real world deployment, due to failures and drastic changes in the network conditions.

Figure 4 depicts fault tolerance tests results on the PlanetLab testbed. We have used 200 Tulip nodes on different computers. Lookup requests were induced into the system at a rate of 2 per second. The graph depicts average stretch of every 4 lookup requests (reflecting two seconds each). At time 150 we randomly killed half the nodes in the system. The results show that after time 300 the systems has almost completely regained its locality properties.

References

1. I. Abraham, C. Gavoille, D. Malkhi, N. Nisan, and M. Thorup. Compact name-independent routing with minimum stretch. The Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 04).
2. Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.
3. A. Gupta, B. Liskov, and R. Rodrigues. One hop lookups for peer-to-peer overlays. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 7–12, Lihue, Hawaii, May 2003.
4. I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.
5. P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02*, March 2002.
6. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. Technical Report Technical Report UCB//CSD-03-1299, The University of California, Berkeley, December 2003.
7. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.