Chapter 1

Peer To Peer - Review

1.1 Plaxton Distributed Search Technique

The Plaxton mesh [33] is a distributed data structure optimized to support an overlay network for locating named objects and routing messages to these objects, forwarding messages is *routing* and the nodes are *routers*, the data structure is the *Plaxton mesh*, it guarantees a delivery time within a small factor of the optimal delivery time, from any point in the network, however one must note that Plaxton mesh is a static data structure, without node insertion or removal. Objects and nodes have names independent of their location and semantic properties, a fixed length sequence represented by a common base (e.g., 40 hex digits representing 160 bits). A roughly even distribution in both node and object name-space is achieved by using hashing algorithms (such as SHA-1 [30])

1.1.1 Routing

Each node has local routing or neighbor maps to incrementally route overlay messages to the destination ID digit by digit (***8 → **98 → *598 → 4598), a similar approach to longest prefix routing is used in CIDR IP address allocation [46]. A node N has a neighbors map with multiple levels, level j represents a matching suffix up to the jth digit in the ID, the number of entries in each level is the size of the ID base, where the ith entry n level j is the ID and location of the closest node which ends in "i"$+suffix(N, j-1)$, for example, the 9th entry in the 4th level of node 325AE is the closest node (in network

distance) which ends in 95AE. With consistent neighbor maps this method guarantees that any existing unique node in the system will be found within at most $log_b N$ logical hops in a N name-space using IDs of base b. Since a node assumes that the preceding digits all match, at each level only a small constant entries (b) are kept (total routing map size is $b * log_b N$. In short, the Plaxton mesh of neighbor maps can be viewed as a set of embedded trees in the network, one rooted in every node (or data item), where the destination is the root.

1.1.2 Location

The location mechanism allows a client to locate and send messages to a named object residing on a server in the Plaxton mesh. A server S publishes that it has an object O by routing a message to the "root node" of the object O, the root node is uniquely defined for every object O. The publishing process consists of sending a message to the root node of object O, at each node along the way, a mapping is stored $\langle$ *Object-ID(O), Server-ID(S)* $\rangle$, mapping are simply pointers to server S (the location of object O), when multiple copies are available, the mapping is for the closest copy.

During a location query, client send messages to objects, initially a message is routed to O's root, at each step, if the message encounter a node with a mapping entry to O, it is immediately redirected to the server containing the object, otherwise it is rooted one step closer to the root of object O, at the root it is guaranteed to find an entry for the object location. The root node concept provides a guaranteed or *surrogate* node where location mapping can be found. Plaxton used a globally consistent deterministic algorithm for choosing a root node for an object, however it requires global knowledge to choose an existing node from a large, sparse name-space and it becomes a single point of failure regarding location queries.

1.1.3 Benefits and limitation

Plaxton provides a *simple fault handling*, since routing requires only suffix matching, message can be routed around a failed link by choosing node with a similar suffix, it is *scalable* since only location routing are kept at each node and is completely decentralized,

resolving each digit allows *exploiting locality* as the number of candidates is reduced by a factor of b (ID base) and the path taken to the root node by the publisher S storing O and the client C will likely converge quickly, so queries for local objects are likely to reach a router with a pointer quickly, Plaxton has proven that the total network distance traveled by messages during both publishing and locating is proportional to the underlying network distance, therefor provide *proportional route distance*.

However, *global knowledge* is requires to compute an object root node, the root node itself is a single point of failure (*route node vulnerability*) and the *lack of ability to adapt* limits the use of the Plaxton mesh.

1.2 Tapestry

Tapestry routing architecture [26] is a self-organizing, scalable wide-area infrastructure that efficiently routes requests to content, providing location independent routing of messages directly to the closest copy of an object without centralized services.

Tapestry provides a basic primitive, the ability to address messages with a location independent name and to request that these messages be routed directly to the closest copy of an object or service.

Tapestry is based on the Plaxton distributed search technique [33], augmented with additional mechanism to provide availability, scalability and adaptive behavior in presence of failures.

The basic scheme is similar to the Plaxton mesh, every node is capable of routing messages as described in the section 1.1, each neighbor map is organized in routing levels and each entry points to a set of nodes closest in network distance to a node which matches the suffix, a node also keeps back-pointers to each node referring to it. While Plaxton keeps a mapping (pointer) to the closest copy, Tapestry keeps pointers to all copies, instead of routing to the closest copy, it allows the application to define the selection operator.

Tapestry maintains soft-state in order to provide fault-tolerance. Caches are updated by a periodic refresh message or deleted by a lack of a refresh message. Keeping two backup neighbors (secondary neighbors) allows switching when the primary neighbor

fails with an overhead of a TCP timeout period, in order to avoid costly insertions, as a neighbor becomes unreachable, it is marked as invalid, and messages are routed via an alternative neighbor. Since most link failures are discovered quickly, during a *second change* period, messages are still routed to the primary neighbor and after a timeout to an alternative. This provides "probes" to the failed node, a successful message indicates a recovery, and the neighbor is marked valid again, a probabilistic function controls the traffic volume to the "invalid" node, if within this time frame the node is not repaired, it is removed.

Adding fault tolerance to location services is provided by assigning multiple "roots" to each object, adding a "salt" values (e.g. 1, 2, 3) to each object ID allows multiple roots. At each mapping entry a soft-state lease is kept and location is announced by a periodic mechanism, making object advertisement and removal transparent. Selecting a "root" node for each object is done using *surrogate routing*, tentatively choosing the root as the object ID, given the sparse name-spaces it is unlikely that such a node exists, none the less, Tapestry operates as if the node exists, sending a message to the root node, a route to a non-existent node will encounter empty neighbor entries at various positions along the way. A deterministic route selection will select an alternative route, routing terminates when the only non-empty entry is the current node, that node is designated as the surrogate root for the object. This method takes additional hops to reach the root compared to Plaxton mesh, however the difference is small.

Integrating nodes into Tapestry is performed using an incremental algorithm, while the algorithm does not guarantee an ideal topology; it is a reasonable approximation that can converge into the ideal topology. The process intuition is as follows: First, we populate the new node's neighbor maps at each level by routing to the new node, and copying and optimizing neighbor maps along each hop from the router, then it notifies it's neighbors so they can updates their maps, the process is started by a bootstrap mechanism.

The dynamic insertion of nodes is non-trivial and takes a non-negligible amount of time; this is the reason for quick re-integrations of repaired nodes. However, deleting nodes is trivial.

1.3 Chord

The *Chord protocol* [40] supports just one operation, mapping a key to a node, depending on the application, that node may store an object associated with the key or provide a service. In a steady state, every node that participate in Chord hold information about $O(logN)$ other nodes and can resolve lookups via $O(logN)$ messages. As nodes join and leave the system, Chord can with high probability maintain lookups with no more than $O(log^2N)$ messages.

1.3.1 The Protocol

Chord use consistent hashing, which provides load balancing (all nodes receive roughly the same number of keys) and requires a small ($O(1/n)$) keys to move when nodes join or leave the network, Chord nodes know only about a small number of other nodes (distributed routing) and resolve hash functions by communication with a few other nodes.

Nodes identifiers are ordered into an identifier circle modulo 2^m, key k is assigned to the first node whose identifier is equal or follows k, this node is called the *successor node* of key k, denoted by $successor(k)$. Consistent hashing is designed to let nodes enter or leave the network with minimal disruption. To maintain consistent mapping, when node n enters the network, certain keys assigned to n successor will move to n, while n leaves the network, all keys assigned to n are reassigned to its successor.

1.3.2 Simple Key Location

The simple key location provides a slow lookup algorithm, lookups can be implemented on a Chord ring utilizing the node successor. Queries for a certain identifier can be passed around the circle via the successor pointers until the successor is equal (or higher) than the identifier. This method uses $O(n)$ messages to perform a lookup.

1.3.3 Scalable Key Location

In order to accelerate lookups, Chord maintains additional routing information. This information is not essential for correctness as long as each node knows its successor,

however, it does provide a speedup.

Each node maintain at most $m = log(N)$ entries, called *finger table*, the i^{th} entry in the table contain the identity of the first node, s, that succeeds n by at least 2^{i-1} on the identifier circle. The finger entry holds the Chord identifier and the IP address of the node. This scheme has two important characteristics. First, each node stores information only about a small number of nodes and knows more about nodes closely following it than nodes farther away. Second, the node finger table does not contain enough information to directly determine the successor of an arbitrary key k. But this scheme allows to reduce the lookup to $O(logN)$.

1.3.4 Dynamic operations

To ensure correct lookups execution, Chord must ensure that each node successor is up to date; it does so using a basic "stabilization" protocol. When a node n starts (and wishes to join the network), first it locates it successor by running $n.join(n')$, where n' is a known node. Every node runs *stab*() periodically. When n runs *stab*() it asks the successor to report its predecessor p, it serves two purposes, allowing n to update its successor if node p has just joined and allowing the successor of n to update its predecessor if needed.

Eventually the nodes will adjust both their successor and predecessor and their finger table to reflect changes.

Failures and nodes leaving the system, may break the ring, Chord maintains a list of the first r successors allowing recovery in face of failures, moreover, when node leaves the system voluntary, it transfers its keys and notify its predecessor and successor of the change, minimizing any disruption time.

1.4 CAN

A Scalable Content-Addressable (CAN) [35] provides a distributed hash-table, each CAN node stores a chunk, called a *zone*. The design maps a node into a virtual d-dimensional Cartesian coordinate space on a d-torus. A messages travels between its source and destination coordinations in the Cartesian space. The Cartesian space is partitioned into zones, basically each node manages a zone and route messages to its neighboring zones

(which overlaps d-1 dimensions). To join the network a node must know an already active CAN node. This scheme provide $O(d(N^{1/d}))$ routing steps until a message arrives at its destination.

In order to decrease the path length, CAN introduce multi-dimensional coordinate spaces, a higher d value decrease the path, and realities, which add independent coordinate spaces, each node is assigned a different zone in a different reality (content is also replicated on each reality). CAN add routing metrics based on the network RTT (round trip time), a next node is chosen based on the ratio of progress to RTT. Overloading zones, allowing multiple nodes to be placed within a zone allows CAN to utilize RTT when selecting a neighbor within a zone and to shorten paths. Adding multiple hash functions adds to the availability of values (but also increase the required storage).

1.5 Pastry

Pastry [18] provide an object location and routing scheme, it route messages to nodes, like Tapestry it is a prefix based routing, but adds proximity and network locality issues (using a proximity metric). At each step, the message travels to a numerically closer node, it provide an expected $log(N)$ steps until a message reaches its destination. Pastry provides a location and routing infrastructure for other application, PAST [15, 16] provide persistent storage and SCRIBE [17] provide a scalable publish/subscribe.

Pastry maintains a *routing table*, a *neighborhood set* and a *leaf set*. The routing table is similar to Tapestry routing, it contains $log(N)$ levels, each level i contains entry that shares the same prefix length i. Pastry keeps at each entry a close (network wise) node which share that prefix. The neighborhood set holds pairs of node id and IP address of nodes which are closest to the local node, it is not normally used for routing purposes but for keeping locality, specifically when a new node joins the network and need to build its routing tables, this information is used to select nodes which are close to the new node. The "leaf set" contains nodes which are numerically close to the local node (both higher and lower), it is used as the first option if the destination node id is within the leaf range (numerically close to the current node), otherwise the prefix based routing scheme is used.

1.6 Open Issues in Peer-To-Peer Systems

In [22] Daswani, Garcia-Molina and Yang review open issues regarding current P2P solutions, specifically searching and security. Searching P2P networks defines the network topology, content placement (including meta information) and P2P network routing. Searching also needs to adhere to the system requirements regarding *expressiveness* of the query language, document retrieval *comprehensiveness* and to the nodes *autonomy* regarding neighbors and content placement.

The system is reviewed by *efficient*, measured in resources (such as bandwidth, storage and CPU). *QoS*, measured in number of results, latency, etc. The system is also reviewed by its *robustness* in face of failures and Byzantine behavior. P2P systems are characterized by their search capabilities, the simple form is *key lookup*, *keyword* searches allows partial or comprehensive searches. Keywords may be *ranked*, or the search may support *aggregation* or even *SQL* type queries. A system QoS can be measured in more than one method, response time, number of results (in partial search) and relevance (in ranked keywords) must be measured in face of its tradeoff, the cost of the query or placement (resources).

P2P System face great security challenge due to their open and autonomous nature, availability is endangered by DoS (Denial of Service) attacks for short term storage and Byzantine for long term. File authenticity is a great challenge when any node can join the network and provide results (incorrect ones), and anonymity is in great demand to keep the privacy of publisher, retrievers and storage locations. Access control is also a key factor when intellectual property is an issue.

Current systems tries to address a subset of these issues but no system yet provides a comprehensive solution.

Bibliography

[1] *"Multicast"*, *http://www.cisco.com/warp/public/732/Tech/multicast/*.

[2] *"NFS version 4"*, October 2002.

[3] Peter J. Braam, *"The coda distributed file system"*, *Linux Journal* no. 50, 1998.

[4] Peter J. Braam, Michael J. Callahan, and Philip I. Schwan, *"The InterMezzo filesystem"*, *In O'Reilly Perl Conference 3.0*, 1999.

[5] Peter J. Braam, *"The lustre storage architecture"*, *http://www.lustre.org/docs/lustre.pdf*, October 2002.

[6] Dimitry Brodsky, Alex Brodsky, Jody Pomkoski, Shihao Gong, Michael J. Feeley, and Norman C. Hutchinson, *"Using file-grain connectivity to implement a peer-to-peer file system"*, *21st IEEE Symposium on Reliable Distributed Systems – Workshop on Peer-to-peer Reliable Distributed Systems*, 2002, pp. 318–323.

[7] Jorge A. Cobb, *"Dynamic multicast trees"*, *IEEE International Conference on Networks*, 1999, pp. 29–36.

[8] Karl Czajkowskiy, Steven Fitzgeraldz, Ian Fosterx and Carl Kesselman, *"Grid information services for distributed resource sharing"*, *10th IEEE International Symposium on High Performance Distributed Computing*, 2001, pp. 181–184

[9] Michael Dahlin, Jian Yin, Lorenzo Alvisi and Calvin Lin, *"Using leases to support server-driven consistency in large-scale systems"*, *International Conference on Distributed Computing Systems*, 1998, pp. 285–294.

[10] Michael Dahlin, Jian Yin, Lorenzo Alvisi and Calvin Lin, *"Hierarchical cache consistency in a WAN"*, *USENIX Symposium on Internet Technologies and Systems*, 1999, pp. 13–24.

[11] Shlomi Dolev, *Self-stabilization*, MIT press, March 2000.

[12] Shlomi Dolev and Ronen I. Kat, *"Self stabilizing distributed file system"*, *21st IEEE Symposium on Reliable Distributed Systems – Workshop on Self-Repairing and Self-Configurable Distributed Systems*, October 2002, pp. 384–379.

[13] John R. Douceur and Roger P. Wattenhofer, *"Optimizing file availability in a secure serverless distributed file system"*, *In proceedings of 20th IEEE SRDS*, 2001 (New Orleans, Louisiana), October 2001, pp. 4–13.

[14] John R. Douceur and Roger P. Wattenhofer, *"Competitive hill-climbing strategies for replica placement in a distributed file system"*, *Proceedings of 15th DISC*, 2001, pp. 48–62.

[15] Peter Druschel and Antony I. T. Rowstron, *"PAST: A large-scale, persistent peer-to-peer storage utility"*, *In HotOS VIII* (Schloss Elmau, Germany), May 2001, pp. 75–80.

[16] Peter Druschel and Antony I. T. Rowstron, *"Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility"*, *Symposium on Operating Systems Principles*, 2001, pp. 188–201.

[17] Peter Druschel, Antony I. T. Rowstron, Anne-Marie Kermarrec and Miguel Castro, *"SCRIBE: The design of a large-scale event notification infrastructure"*, *Networked Group Communication*, 2001, pp. 30–43.

[18] Peter Druschel and Antony I. T. Rowstron, *"Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems"*, *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001, pp. 329–350.

[19] Zvi Dubitzky, Israel Gold, Ealan Henis, Julian Satran and Dafna Sheinwald, *"DSF - data sharing facility"*, *Tech. report, IBM Research Division, Haifa Research Laboratory, Israel*, 2002.

[20] Ian Foster and Carl Kesselman, *"The Globus project: a status report"*, *Future Generation Computer Systems* **15**, 1999, pp. 607–621.

[21] Ian Foster, Carl Kesselman, and Steven Tuecke, *"The anatomy of the Grid: Enabling scalable virtual organization"*, *The International Journal of High Performance Computing Applications* **15**, 2001, pp. 200–222.

[22] Hector Garcia-Molina, Neil Daswani and Beverly Yang, *"Open problems in data-sharing peer-to-peer systems"*, *9th International Conference Siena, Italy, January 2003. Proceedings, ICDT: the International Conference on Database Theory*, January 2003, pp. 1–16.

[23] John H. Howard, *"An overview of the Andrew File System"*, *Proceedings of the USENIX Winter Technical Conference* (Dallas, TX), February 1988, pp. 23–26.

[24] Paul Karlton, Alan Frier and Phil Kocher, *"The SSL 3.0 protocol"*, Netscape Communications Corporation, November 1996.

[25] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells and Ben Zhao, *"OceanStore: An Architecture for Global-scale Persistent Storage"*, *Proceedings of ACM ASPLOS*, 2000, pp. 190–201.

[26] John Kubiatowicz, Ben Y. Zhao and Anthony D. Joseph, *"Tapestry: An infrastructure for fault-tolerant wide-area location and routing"*, *Tech. Report UCB/CSD-01-1141*, UC Berkeley, April 2001.

[27] Nancy Lynch, Michael Merritt, William Weihl and Alan Fekete, *Atomic Transactions*, Morgan Kaufmann Publishers, 1994.

[28] Dahlia Malkhi, Moni Naor, and David Ratajczak, *"Viceroy: A scalable and dynamic emulation of the butterfly"*, *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[29] Daniel A. Menasce, *Sliding Through Operating Systems*, *http://cs.gmu.edu/∼menasce/osbook*, 1995.

[30] National institute of standards and technology, *"Secure Hash Standard"*, 1995.

[31] Network Appliance Inc, *"NFS version 4 protocol"*, RFC, December 2000.

[32] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel and David Hintz, *"NFS version 3 design and implementation"*, *Proceedings of summer USENIX Conference*, June 1994, pp. 137–152.

[33] Charles Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa, *"Accessing nearby copies of replicated objects in a distributed environment"*, *ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 311–320.

[34] Kavitha Ranganathan and Ian T. Foster, *"Identifying dynamic replication strategies for a high-performance data grid"*, *In GRID*, 2001, pp. 75–86.

[35] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, *"A scalable content addressable network"*, *Proceedings of ACM SIGCOMM 2001*, 2001.

[36] Matei Ripeanu, *"Peer-to-peer architecture case study: Gnutella network"*, *Tech. Report 2001-26*, Univeisity of Chicago, 2001.

[37] Ohad Rodeh and Avi Teperman, *"ZFS - A scalable distributed file system using object disks". 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, April 2003, pp. 207–218.

[38] Sun Microsystems Inc, *"NFS - network file system protocol specification"*, RFC, March 1989.

[39] Sun Microsystems Inc, *"NFS version 3 protocol specification"*, RFC, June 1995.

[40] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, *Chord: A scalable Peer-To-Peer lookup service for internet applications*, *In Proc. of ACM SIGCOMM*, 2001, pp. 149–160.

[41] Andrew S. Tanenbaum, *Modren operating systems*, Prentice-Hall International, Inc, 1992.

[42] Thedore Ts'o, Rmy Card and Stephen Tweedie, *"Design and implementation of the second extended filesystem"*, *Dutch International Symposium on Linux*, December 1994.

[43] Michael J. Tucker, Christopher A. Stein and Margo I. Seltzer, *"Reliable and fault-tolerant peer-to-peer block storage"*, *Tech. Report TR-04-02*, Harvard Computer Science, 2002.

[44] Michael J. Tucker, Christopher A. Stein and Margo I. Seltzer, *"Building a reliable mutable file system on peer-to-peer storage"*, *International Workshop on Reliable Peer-to-peer Distributed Systems*, 2002, pp. 324–329.

[45] Sudharshan Vazhkudai, Steven Tuecke and Ian Foster, *"Replica selection in the Globus data grid"*, *In International Workshop on Data Models and Databases on Clusters and the Grid (DataGrid 2001)*, 2001, pp. 106–113.

[46] Jessica (Jie Yun) Yu, Vince Fuller, Tony Li and Kannan Varadhan, *"Classless interdomain routing (CIDR) : An address assignment and aggregation strategy"*, RFC 1519, IETF, Sepetember 1993.

[47] Tatu Ylonen, *SSH - Secure login connections over the internet*, *Proceedings of the 6th Security Symposium (USENIX Association: Berkeley, CA)*, 1996, pp. 37–42.