Achord: A Variant of the Chord Lookup Service for Use in Censorship Resistant Peer−to−Peer Publishing Systems

Steven Hazel and Brandon Wiley
sah@thalassocracy.org, brandon@blanu.net

Abstract

Any peer−to−peer publishing system must provide a mechanism for efficiently locating published documents. For censorship resistant systems, it is particularly important that the lookup mechanism be difficult to disable or abuse. Chord [1] is a promising distributed lookup mechanism, because analysis has provided certain useful guarantees about the speed and correctness of Chord's operation. We examine the suitability of Chord for building censorship resistant peer−to−peer publishing systems , and suggest Achord, a variant of the Chord mechanism which takes into account the additional requirements imposed by the goal of censorship resistance.

1. Introduction

The provable performance and correctness of Chord make it an attractive option for distributed lookup. It is guaranteed that data will be located if it exists in the network, and that the number of messages required to perform a given lookup scales more−or−less logarithmically with the size of the network. In a fully optimized N−node network the number of messages involved in a Chord lookup will be limited to $O(\log N)$. As nodes join and leave the system, the network can be returned to full optimization with a number of messages that will with high probability be no greater than $O(\log^2 N)$. These guarantees are considerably stronger than those provided by the lookup mechanisms employed in Freenet [2] and FreeHaven [3], where data that exists in the network is not guaranteed to be found, and the number of messages involved in a successful lookup is unknown. It would therefore be desirable to implement Chord such that it could be used in a censorship resistant peer−to−peer publishing system.

Providing censorship resistance requires careful design of every component in the architecture of a peer−to−peer system. Where other applications might allow for a broad choice of suitable components, censorship resistance requires that each component function in a way that is compatible with the strategy by which the system aims to provide censorship resistance. In order for a lookup mechanism to be suitable for use in a censorship resistant peer−to−peer publishing system, it must locate data in a scalable manner while allowing the system to maintain the following properties:

Property 1 *It must be possible to insert data into the system without revealing the identity of the inserter, so that attacking those who insert information will not be a viable means of censoring that information.*

Property 2 *It must be possible to retrieve data from the system without revealing the identity of the recipient, so that attacking those who request information will not be a viable means of censoring that information.*

Property 3 *It must be difficult to introduce a new node to the network such that it will become responsible for a particular document, so that voluntarily assuming responsibility for a document and then deleting it will not be a practically viable means of censoring arbitrary documents.*

Property 4 *It must be difficult to identify the node which is responsible for storing a given document, so that identifying attacking individual container nodes will not be a practically viable means of censoring arbitrary documents.*

The last of these properties is particularly at odds with Chord's basic operation. While most lookup services map keys to values, Chord instead maps from keys to the nodes which are responsible for storing the associated values. The implementations of Chord suggested thus far have specified that the identity of that node be returned to the requester. This functionality has also been used extensively in the network stabilization procedure which occurs when a node joins or leaves a Chord network. Thus, because looking up the node responsible for a particular document is the basic operation Chord provides, Chord does not allow for the final property listed above to be maintained. This is only a problem with a few incidental specifics of Chord's design, however, and not a fundamental problem with the Chord lookup scheme.

This paper presents *Achord*, a service that provides Chord−like lookup of values, rather than nodes, in a way which does allow for the above properties to be maintained, without compromising any of the guarantees that can be made about the correctness and performance of Chord.

While there is nothing about Chord (or Achord) which would prevent some external mechanism from providing Properties 1 and 2, Achord provides some degree of requester and inserter anonymity directly, as part of its normal mode of operation.

2. Design

Chord nodes are assigned an m−bit identity (based, for example, on the SHA−1 hash of the node's IP), and keys are m−bit quantities (obtained, e.g., from the SHA−1 hash of a text string). A node is responsible for a key if its identity the nearest successor of that key. The fundamental strategy for providing Property 3 in Achord, as in Freenet, is to prevent nodes from choosing their own identities. Achord requires that a node's ID be chosen in a consistent way based on a link−layer−assigned identity which can be verified by the link layer during communication with that node. For Internet applications, the node's IP address is a perfect candidate, and the rest of this paper assumes that a node's ID will be based on the SHA−1 hash of its IP. (Chord also has a concept of "virtual nodes", which Achord does not alter, and which we do not discuss in depth here. If virtual nodes are used in an Achord implementation, we assume that they are numbered, and that their IDs are based on the SHA−1 hash of the concatenation of the node's IP and the virtual node number.)

In an N−node Chord network, each node maintains a *finger table* of only $O(\log N)$ other nodes. In the course of the operation of Chord, however, each node has access to considerably more information about the structure of the network. In pursuit of Property 4, the design of Achord attempts to limit each node's knowledge of the network as closely as possible to the data in its finger table.

The fundamental operation in Chord is *find_successor*. The node which has an ID that is the nearest successor of a key is responsible for storage of the value associated with that key, so *find_successor* is used to perform key lookups in Chord. When a new node joins the Chord network, it uses *find_successor* to determine its place in the network and initialize its finger table, and as the network operates, each node works to optimize the network routing, using *find_successor* to get information about surrounding. There are two ways in which a node might compute *find_successor* for a given key k [4]:

1. The *iterative* method, in which the node requesting the lookup contacts nodes in the network directly, making queries about their routing tables, until it has identified the node which is the successor of k.

2. The *recursive* method, in which the node requesting the computation contacts only the node in its own finger table which most closely precedes k, and that node performs the same operation for its own finger table, and so on, passing the query through the network until the successor node is contacted.

Because *find_successor* allows a node to determine which keys another node is responsible for, Achord limits its use strictly. The results of *find_successor* are only returned under very specific circumstances. Most significantly, Achord maps keys to values, rather than to nodes −− a key lookup never returns the result of *find_successor* to the requester.

Achord key lookups are performed using an operation, *connect_to_successor*, which works almost identically to the recursive *find_successor*. With *connect_to_successor*, however, the results of *find_successor* are never returned: when the successor node is contacted during a request, the value, rather than the node ID, is "tunneled" back along the recursive search path to the originator of the request. Similarly, to insert a value into the network, a node performs the *connect_to_successor* operation to establish a tunneled connection to the node which would be responsible for the given key, and sends the value along that connection path. As in Crowds [5], tunneling provides document requesters and inserters some measure of anonymity (and Achord provides Properties 1 and 2), because a node which receives a given request has no way to determine whether the immediate requester is proxying the request from a node even further away from the key. Similarly, the identity of the node responsible for storing a given key is protected. In this respect, Achord is similar to the lookup mechanism provided by Freenet, which also protects storage node identities by providing tunneled connections.

Beyond merely protecting node identities during requests and inserts, Achord must also maintain the state of the network while limiting the amount of information each node is able to gain about the network, and preventing each node from making directed inquiries about which nodes are responsible for which keys. Therefore, we have made considerable modifications to the stabilization procedure by which Chord optimizes a network as nodes simultaneously join and leave.

2.1 Stabilization Revised

In order for a Chord network to maintain fast and reliable lookups, certain properties of the network must be maintained as nodes join and leave the network. Nodes must have their successor and predecessor fields set properly and when possible their finger tables should contain the closest matching values for each slot.

In Chord, when a new node *n* joins the network, it must know the IP of a node which can introduce it to the network. It calls *find_successor* on that node to find its own successor. It then starts with its successor's *.predecessor* field and walks back in order to find its own predecessor. It periodically checks its predecessor and successors to make sure they are current and also updates its finger tables so that they remain current and optimal.

In order for Achord to restrict each node's knowledge of the network, access to other nodes' successor and predecessor fields and finger tables must be restricted to only valid, non–hostile uses. Fortunately, this information will only be accessed by a well–behaved node in certain verifiable contexts.

The first context in which a node may obtain a new IP address from another node is when calling *find_successor* to find its own successor upon first joining the network. However, in this case, a node with ID *n* will always call *find_successor*(*n*). This call will always be benign because every node is allowed, and in fact expected, to know its own successor. Since the ID of a node is based on its IP, it is verifiable by the node on which *find_successor*(*n*) is being called that *n* is calling *find_successor* on its own ID. It is important for the restrictions which will be imposed on accessing a node's *predecessor* field that one of the side effects of *find_successor* be that *n'.find_successor*(*n*) will set *n'.predecessor* to *n* if *n* is a better match than the current value for the *predecessor* field on *n'*. In Chord these two steps are considered to be orthogonal. However, it is important in Achord that after a node *n* joins the network (via successive calls to *find_successor*) there is some node *n'* such that *n'.predecessor* is *n*.

Rule 1 *Only the node with ID* n *is allowed to call* find_successor(n).

Chord presents two implementations of *find_successor*, iterative and recursive. However, because of this limitation being placed on when it is proper to call *find_successor*, only the iterative version is possible. A node participating in the calculation of the recursive version would only be able to retrieve the successor for itself, not for the requesting node.

Rule 2 *Only the iterative version of* find_successor *can be used.*

It is significant to note that since Chord provides that *find_successor* can be computed in no more than $O(\log N)$ messages, only at most $O(\log N)$ nodes need learn of the existence (and ID) of *n* in order for it to join the network.

Theorem 1 *Only* O*(log* N*) nodes need to find out about the existence of a node in order for it to join the network.*

The only context in which a node will access another node's *predecessor* field is during its periodic stabilization routine. In this case it will always been accessing *n'.predecessor* where *n'* is *n.successor*. Since *n'* does not know all of the IPs in the network, it is not possible for *n'* to determine if it is the proper successor for *n*. However, there is still a verifiable invariant in this case. In order for *n* to think that it's successor is *n'*, *n'* must have considered *n* to be its successor at some point in the past. This is true because when a new node joins the network it sets *n.successor=n'.find_successor*(*n*). This call both sets *n.successor=n'* and *n'.predecessor=n*. The only *predecessor* field that *n* has the right to access is the one on *n'* until *n.successor* changes. Therefore, *n'* can determine if the call to its *predecessor* field is valid by maintaining a list of old values for *predecessor* and seeing if *n* is in that list. Additionally, access of the *predecessor* field will always return *n* until *n* has a new successor. When an access to *n'.predecessor* yields something other than *n*, say *n''*, *n'* can remove *n* from its list of nodes allowed to access the *predecessor* field, as all valid future accesses from *n* of a *predecessor* field should be on *n''* instead. Using this restriction, the *predecessor* field can only be used to find one IP at a time and the rate at which the *predecessor* field changes is set by the rate at which nodes are joining and leaving the network, not by the attacker.

Rule 3 *A node* n *is only allowed to access the* predecessor *field of another node* n' *if* n *was previously a value of* n'.predecessor, *and* n *has not accessed* n'.predecessor *since* n'.predecessor*'s value changed from* n.

The final context in which a node validly needs to discover the IPs of other nodes is when updating its finger tables. In Chord this is accomplished by taking a random node from the current finger table and then walking the ring from that node until either a better node is found for that position, or else it is determined that there is no better node in the vicinity. This method is entirely replaced in Achord by another method of obtaining better finger table entries. In Achord, updating the finger table is accomplished by choosing a random node, *n'*, from the current finger table and calling *n'.find_best_match*(*i*), where *i* is the index into the finger table for that node. There is no way for *n'* to know if it was actually the i^{th} node in *n*'s finger table. However, it is not necessary to do so in order to limit the abuse of the finger table update method. The node *n'* knows the IP address of *n* and can thus determine the ID of *n* as well as what the ideal IDs are for each slot in *n*'s finger table. The *find_best_match* function is used iteratively just like *find_successor*. The node *n'* will look at its finger table in the i^{th} position and return whichever is the better match for *n*'s i^{th} position, either the node in that slot or *n'* itself.

Rule 4 *Finger tables are updated using the* find_best_match *function, which only returns a new IP if the new IP is a closer match to one of the slots in* n*'s finger table slot than* n'.

The benefit of this method of updating the finger table is that it limits the number of IPs which can be harvested by any one node. Since new IPs are only returned if they are closer to the node's ideal set of IPs, repeated querying of all known nodes in a static network will quickly result in a static set of IPs being returned. Due to the nature of Chord routing it is only possible to harvest $O(\log N)$ IPs for each entry in the finger table. So, in a static network, at most $O(k \log N)$ IPs can be harvested by any one node, where *k* is the size of the finger table. If the network is dynamic then all of the nodes will find out about new nodes over time, as is necessary for the network to properly function. The rate at which new IPs can be harvested in a dynamic network depends on the rate at which nodes are joining and leaving the network and is not under the control of the attacker.

Theorem 2 *In a static network, a single node can harvest at most* O*(*k *log* N*) IPs.*

3. Future Work

There are a number of potential attacks on Achord that are worth exploring.

It would be possible to attack an Achord network by using a large number of IPs to get more chances to have a node which is responsible for a given key. It's not entirely clear how many IPs an attacker would have to command in a network of a particular size in order to become responsible for a given set of keys.

Achord's attempt to disguise the identities of inserters and requesters might not be as effective as Freenet's. In Achord, a node receiving a request will have some idea of the distance between the key and the requester's node ID, and perhaps this might be used to estimate the probability that the requester in fact originated the request.

Despite the precautions taken in section 2.1, nodes have some limited ability to learn about other nodes in the network as it stabilizes. It is unclear how much a node can learn about a network in a given certain amounts of time and network traffic. It might be fruitful to explore various schemes for preventing nodes from learning about the network too quickly, such as tying stabilization information to data requests, or having nodes push stabilization data out to other nodes at regular intervals, rather than returning it upon request.

4. Conclusion

Achord provides a lookup mechanism that is equivalent to Chord in performance and correctness, but which is suitable for use in censorship resistant peer−to−peer publishing systems. Its basic tactics for providing anonymity and limiting each node's knowledge of the network are similar to those of Freenet. In most areas, stronger claims can be made about Achord's performance characteristics than can be made about Freenet's. Further analysis is needed to determine how secure Achord is against attack.

Acknowledgements

We would like to thank Roger Dingledine for bringing Chord to our attention, and for his valuable comments and analysis.

5. References

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer−to−peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM 2001*, San Diego, California, August 2001. An early version appeared as LCS TR−819 available at `http://pdos.lcs.mit.edu/chord/#pubs`.

[2] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, California, June 2000. `http://www.freenetproject.org/`.

[3] Roger Dingledine, David Molnar, and Michael J. Freedman. The Free Haven Project: Distributed Anonymous Storage Service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000. `http://www.freehaven.net/papers.html`.

[4] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building Peer−to−Peer Systems with Chord, a Distributed Lookup Service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001. `http://pdos.lcs.mit.edu/chord/#pubs`.

[5] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for Web Transactions. In *Communications of the ACM 42(2)*, June 1999. `http://www.research.att.com/projects/crowds/`.