Accepted at **ISCC '09 -**
In proceedings of the fourteenth IEEE International Symposium on Computers and Communications (poster), July, 2009, Sousse, Tunisia. (to appear)

Babelchord: a Social Tower of DHT-based Overlay Networks*

Luigi Liquori Cédric Tedeschi Francesco Bongiovanni
INRIA Sophia Antipolis - Méditerranée, France
surname.name@sophia.inria.fr

Abstract

Chord is a distributed protocol performing efficient node lookup in a ring-based overlay network. Babelchord is a distributed protocol that performs resources virtualization and nodes self-aggregation via a social tower of Chord-based "floors". Nodes can register to one or many floors upon a floor's consensus. Babelchord provides a cost-effective alternative to hierarchical structured P2P systems and DHT merging by connecting smaller structured overlay networks in an unstructured way. Lookup routing performs as in Chord but a node belonging to more than one tower's floor, floods the request to all the floors it belongs to. The final peer (resp. floor), responsible for the successful lookup, is inserted into an hot peer (resp. hot floor) list that will be used to generate new floors' joins and creations. As such, inter-floors connections take place through "nodes at crossroads", a sort of neural synapses*. Results from simulations show that Babelchord scale up logarithmically with the number of Babelchord nodes and floors: moreover a little number of synapses is sufficient to achieve an exhaustive lookup.*

1 Introduction

A significant part of today's Internet traffic is generated by peer-to-peer (P2P) applications, used originally for file sharing, and more recently for real-time multimedia communications and live media streaming.

Distributed hash tables (DHTs) or "structured overlay networks" have gained momentum in the last few years as the breaking technology to implement scalable, robust and efficient Internet applications. DHTs provide a lookup service similar to a hash table: (key, value) pairs are stored in the DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from names to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows DHTs to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

Chord [8] is one of the simplest protocols addressing key lookup in a distributed hash table. Chord adapts efficiently as nodes join and leave the system. The Chord protocol scales up logarithmically with the number of nodes. Chord uses consistent hashing in order to map keys and nodes' addresses, hosting the distributed table, to the same logical address space. No reputation mechanism is required to accept, reject, or reward peers that are more reliable or more virtuous than others. Merging two Chord rings together is a costly operation because of the induced message complexity and the substantial time the distributed finger tables needs to stabilize. Both rings have to know their relative hash functions and have to decide which ring will absorb the other one, the latter point being critical because of the politics and security reliance's. In this paper, we propose to connect smaller Chord networks in an unstructured way via special nodes playing the role of *neural synapses*.

1.1 Features

Schematically, the main Babelchord's features are:

Routing over SW/HW-Barriers. Namely, the ability to route queries through different, unrelated, DHTs (possibly separated by firewalls) by "crossing floors". A peer "on the border" of a firewall can bridge two overlays (having two different hash functions) that were not meant to communicate with each other unless one wants to merge one floor into the other (operation with a complexity linear in the number of nodes).

Social-based. Every peer has data structures recording peers and floors which are more "attractive" than others. An "hot" node is a node which is stable (alive) and

*Supported by AEOLUS FP6-IST-15964-FET Proactive: Algorithmic Principles for Building Efficient Overlay Computers.

which is responsible for managing a large number of (keys-values) in all hosted DHTs. An "hot" floor is a floor responsible of a high number of successful lookups. Following a personal "good deal" strategy, a peer can decide to invite an hot node on a given floor it belongs to, or to join an hot floor, or even create from scratch a new floor (and then invite some hot nodes), or accept/decline an invitation to join an hot floor. This social-behavior makes the Babelchord network topology to change dynamically. As observed in other P2P protocols, like Bittorrent, peers with similar characteristics are more willing to group together on a private floor and thus will eventually improve their overall communications quality. Finally, the "good deal" strategy is geared up to be further enhanced with a reputation-system for nodes and floors.

Neural-inspired. Since every floor has a proper hash function, a Babelchord network can be thought as a sort of *meta overlay network* or *meta-DHT*, where inter floors connections take place via crossroad nodes, a sort of neural synapses, without sharing a global knowledge of the hash functions and without a time consuming floor merging. The more synapses you have the higher the possibility of having successful routings is.

1.2 Suitable applications

Because of the above original features, the following are examples of applications for which Babelchord can provide a good groundwork (in addition, of course, to all genuine Chord-based applications, like cooperative mirroring, time-shared storage, distributed indexes and large-scale combinatorial search).

Anti Internet censorship applications. Internet censorship is the control or the suppression of the publishing or accessing of information on the Internet. Many applications and networks have been recently developed in order to bypass the censorship: among the many we recall Psiphon, Tor, and many others. Babelchord can support such applications by taking advantage of intra-floor routing in order to bypass software barriers.

Fully Distributed social-networks applications. Social-networks are emerging as one of the Web 2.0 applications. Famous social networks, such as Facebook or LinkedIn are based on a client-server architecture; very often those sites are down for maintenance. Babelchord could represent a scalable and reliable alternative to decentralize key search and data storage.

1.3 Related work

Apart from Chord protocol, our proposal is inspired by the generic Arigatoni overlay network [1, 5] built over a number of "agents", organized in *colonies*, and ruled by a broker leader, elected democratically or imposed by system administrators. Every agent asks the broker to log in the colony by declaring the resources that can be offered. Once logged in, an agent can ask the broker for other resources. Colonies can recursively be considered as evolved agents who can log in an outermost colony governed by another super-leader. Every broker routes intra- and inter-service requests by filtering its resource routing table, and then forwarding the request first inside its colony, and second outside, via the proper super-leader. When the client agent receives notification of all (or part of) the requested resources, then the real resource exchange is performed directly by the server(s) agents, in a pure peer-to-peer fashion.

Hierarchical structured overlay networks are one of the latest trends in peer-to-peer systems. Authors of Brocade [9] introduced a two-level DHT. Their key idea is to let an administrative domain build its own DHT in which one or several leaders are elected according to various metrics such as CPU, bandwidth, etc. These leaders will then enter a *interdomain* DHT which connects local DHTs by high-speed links. Authors in [3] generalized the concept to any number of levels, and adapted to the IP-numbering [4].

Another kind of approach, introduced in [6] consists in using a set of stable nodes distributed on the physical network, called *landmarks*. These landmarks are used to dispatch nodes in virtual *bins* considering a given metric, such as the latency. Each node computes the latency between itself and each landmark, sorts them and thus finds its own bin. The intuition behind this is that nodes that are close to each other have similar landmark measurements.

To achieve exhaustiveness, hierarchical approaches require mergers. [7] focused their attention on merging several similar overlays together. However, as argued in [2], the mechanisms involved generate a significant message overhead, not to mention the time it takes for the future unique ring to converge towards a steady state.

2 Babelchord's social tower

Babelchord is a distributed protocol that performs resources virtualization and nodes self-aggregation via a social tower of Chord "floors". Nodes have "generic resources" and they can register to one or many floors upon a floor's consensus. The capacity to enter an overlay network on the basis of a peer negotiation

(strongly related to social networking phenomenais another Babelchord peculiarity. One peer must convince the peer playing the role of floor "entry point" that he represent a "good deal" for the floor community. Babelchord extends Chord in the following points:

Nodes and their resources. Every peer comes with a list of resources that can be requested/offered from/to other peers belonging to the Babelchord network; the true resource exchange will be performed in a pure point-to-point mode but resources must be declared at the time the peer enters the overlay. When a peer enters the network, Babelchord updates the DHT mapping every resource to the list of IP addresses offering that resource. The rationale is simple: the more (relevant) resources the node injects in the network (more specifically in the 'floor'), the higher the possibility to successfully enter the Babelchord network (floor) and get more connections. This operation is based on the tit-for-tat strategy[1], commonly used in economics, social sciences and in the Bittorrent protocol. It is clear that the more floors the node is registered to, the larger the fingers table (matrix) will have to be managed. However we can assume that the numbers of floors a node belongs to would be pretty low and moreover it is the node's choice to belong to more floors, thus it knows it has the capacity to deal with a this routing and storage overhead.

Floors and floors intersections. Nodes can belong to many rings, called "floors". Every floor has a proper hash function in order to perform consistent hashing of nodes and keys within it. Node structures, like `pred`, `succ` and `finger`, employed to perform routing on a single ring, must be upgraded so to take into account the multi-floor extension.

Multi-floor routing. When a node lookups for a resource on a given floor (using the hash function peculiar to the current floor), a BabelChord routing is launched. A unique tag identifier of the query, based on the node's system time and IP-address[2] for instance, is created. If the routing goes through a synapse node, then the lookup is launched recursively on all the floors the synapse node belongs to, otherwise the routing goes as in Chord. To do this, every node has the hash function relative to the floor we are currently routing on. This means that resources and nodes must be hashed at every floor change. The rationale of this propagation is simple: the more floors you propagate the lookup, the higher possibility to solve the lookup you have. It is important to notice that while the routing complexity of a single floor lookup is *exhaustive* and *logarithmic* in the number of nodes, the whole lookup in Babelchord *can be non exhaustive* with a routing complexity that can vary according to the *number of floors* (inter-floor routing) *times a logarithmic factor* (intra-floor routing).

Multi-floor "cut-over" strategy. Suppose a node belongs to more than one floor. Every time a node is contacted, the unique tag of the query is recorded by the node (each node maintains a tags' list which is purged periodically). When the node is contacted to resolve a query on a given floor it also launches recursively the same query on all floors it belongs to. Further routing of the same query will be aborted just by checking if the lookup previously passed there. Every lookup has also a TTL, in order to avoid circular routing. This feature, combined with the unique lookup tag, prevents the system from generating unnecessary queries and thus reducing the global Babelchord number of messages. A nice property of Babelchord's routing mechanisms is that with a fairly low amount of synapses, we can still achieve a pretty high query exhaustivity.

Peers and floors "mercato". Every peer contains a list of the hot peers that are responsible for successful lookups on a given floor and a list of hot floors that are responsible for the biggest number of hot peers. Run periodically on every node, Babelchord can either:

- suggest the current node to join a "hot" floor; thus increasing the current node's connectivity;
- suggest a "hot" node to join a floor the current node belongs to;
- suggest the current node to create a new floor from scratch: this can be done using a "hash function generator" that, for every unique seed (depending, *e.g.*, on the IP-address, the MAC-address, the request time), identifies the new floor to be created.

2.1 An example

We illustrate the Babelchord protocol by giving an example of a simple two floors overlay topology and multi-floor lookup routing. Figure 1 shows a topology made of two floors `F1` and `F2`. Nodes are depicted by capital letters, *i.e.* `A,B,..I`, and `J`. Assume that every node offers a single resource, *i.e.* `a,b,...,i`, and `j`. The two Babelchord floors intersect via two synapses nodes `A` and `B`. Hops are labeled by an integer `i` denoting a global time of arrival on a given node. Since intra-floor routing is performed as in Chord, fingers and hand tables

[1]The strategy for joining a floor is left to the developers since it strongly depends on the content exchanged in the floors.

[2]You can base the creation of the tags on what you want, as long as you can guarantee unicity.

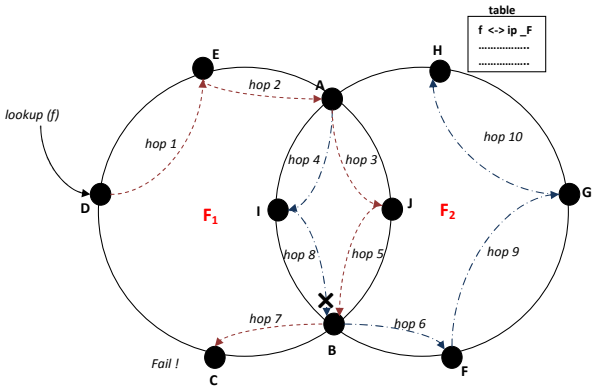

Figure 1. Multi-floor routing example

are hence omitted. For graphical purposes, routing in the first floor moves clockwise, while in the second floor moves anticlockwise. For clarity, we omit to hash nodes and resources via the two different hash functions.

A `lookup(f)` is received by node `D` on floor `F1`; the intra-floor routing (hops `1` and `2`) goes to `E` and `A` (the synapse) that, in turn, will trigger an inter-floor routing on floor `F2` (hop `4`). The routings proceed in parallel on both floors passing by nodes `I`, `J`, and `B` (hops `3`, `4`, `5` and `8`, respectively).

Since node `B` is the second synapse, when it receives the `lookup(f)` by hop `8` on floor `F2`, it will not forward it since it already saw the lookup triggered on `F1`. In the meantime, the two lookups will continue their route to nodes `C` (routing failure on floor `F1`), on nodes `F`, `G`, and finally terminate on node `H` hosting the value `ip_F` offering the resource `f`.

3 Babelchord's protocol

3.1 Overview

In this section, we present the details of the algorithms building a BabelChord network. First, we focus on the data structures used, *i.e.*, how Chord's data structures are extended to support multi-floor architectures and protocols. Then, in Section 3.3, we give the main part of the protocol, *i.e.*, the lookup within a BabelChord network. Finally, in Section 3.4, we give an idea of how peers can negotiate the creation of new rings and the insertion of some node. Recall that the word *floor* refers to one Chord ring. Both words designate the same object.

3.2 Data structures for every peer

The structures used are detailed in Figure 2. Let `f` represents both the identifier of a floor and, with a slight abuse of notation, the hash function `hash(f)(-)` used at this floor (we assume a distinct cryptographic hash function for each floor). The `tag` structure is a list of unique identifiers of Babelchord requests, serving the *cut-over* strategy. The `res` structure represents the set of resources offered by a single peer, while the `table` structure is the part of the hash table the peer manages, containing the associative array of resource keys and IP-addresses providing these resources. Every peer contributes actively to routing through its `table` and to resource exchange through the resources declared in `res`. The `succ`, `pred`, and `hands` structures contain predecessor, successor and finger information for every floor a node belongs to. Finally, `hotpeers` and `hotfloors` contain information collected through lookups about peers for potential collaborations and floors for potential participation.

3.3 The lookup protocol

The multi-floor lookup is illustrated in Figure 3. Lines 1.01 to 1.04 initiate a `lookup` on a resource `r`.

After creating a new unique tag for this request, the current node initiates the lookup by sending a `FINDSUCC` message to itself, and waits for the response, *i.e.*, a `FOUND` message specifying the IP-address of a node storing the sought key. On receipt of a `FINDSUCC` message (Line 1.10), the current node distincts two types of messages:

Join routing (Lines 1.12-1.16). When the first element of the message is a `join` tag, it means, as we will detail later, that the lookup serves a join purpose. The request is then routed as in a simple Chord `join`, and corresponds to the routing process of either a resource registration or a peer insertion. Note that when the requested `id` falls between the current node and its successor `succ`, the node targeted by the routing is `succ`. A `FOUND` message is returned to the initiator of the routing `ip` at Line 1.13.

Resource lookup routing (Lines 1.18-1.25). If the first element of the message is a numeric tag, it means that the message is part of a resource lookup request. The message can then be routed in several rings the current node belongs to. First, the tag of the request is checked (Lines 1.17-1.18). If the request was already processed, it is simply ignored, implementing the *cut-over* strategy. Otherwise, the request tag is saved in order to avoid routing the same request several times. Note that, the requests are routed according to every floor a node belongs to. When the routing reaches its destination on a given floor, a `FOUND` message containing the address of the node storing the

Node's Data structures

```
f(-)      def= hash(f)(-) with hash:int->Sha1      the hash function f (small notation abuse)
tag       def= (int)*                              the list of all unique tags identifying a babelchord packet
res       def= (r)*                                the list of all resources offered by the current node
table     def= (r,(ip)*)*                          the associative array of resource-peers offering that resource
succ      def= (f,ip)*                             the associative array of successors at floor f
pred      def= (f,ip)*                             the associative array of predecessors at floor f
fingers   def= [ip]                                array of ip addresses
hands     def= (f,fingers)*                        the associative array of fingers at floor f
hotpeers  def= (ip,(f)*)*                          the associative array of hot peers view by different floors
hotfloors def= (f,(ip)*)*                          the associative array of hot floors inhabited by different peers
```

Figure 2. Node's data structures

The Babelchord's protocol

```
1.01 on receipt of LOOKUP(r) from ip do                looking for nodes hosting resource r
1.02   t = new_tag(ip);                                new unique tag for this lookup
1.03   this.insert_tag(t);                             insert tag into the tag list
1.04   send FINDSUCC(t,f,r,ip) to this.ip;             send findsucc to itself
1.05   receive FOUND(f,ip2) from ip3                   wait for the Babelchord routing
1.06   if ping(ip2)                                    test the aliveness of ip2
1.07     this.update_hotpeers(ip2,f);                  update the hot peer list with ip2 at floor f
1.08     this.update_hotfloors(f,ip2);                 update the hot floor list with f signaled by ip2
1.09     return lookup_table(ip2,r);                   remote table lookup on ip2; return the list of ips offering the resource r

1.10 on receipt of FINDSUCC(t,f,r,ip) from ip2         find the successor of ip
1.11   if t = join                                     join a floor
1.12     if f(r) ∈ (f(this.ip),f(this.get_succ(f))]    as in Chord
1.13       send FOUND(f,this.get_succ(f)) to ip;       found the successor of ip
1.14     else
1.15       ip3 = this.closest_preceding_node(f,r);     internal chord routing
1.16       send FINDSUCC(t,f,r,ip) to ip3;             send to the next hop
1.17   else if not(this.in_tag(t))                     lookup not processed
1.18     this.push_tag(t);                             mark as ''already processed''
1.19     for all f ∈ this.dom_hands() do               for all floors of current node
1.20       if f(r) ∈ (f(this.ip),f(this.get_succ(f))]  test if arrived, as in Chord
1.21         send FOUND(f,this.get_succ(f)) to ip;     found a node hosting an entry for r
1.22         exit forall;                              stop the routing: ''game over''
1.23       else
1.24         ip4 = this.closest_preceding_node(f,r);   internal chord routing
1.25         send FINDSUCC(t,f,r,ip) to ip4;           send findsucc to the next hop
```

Auxiliary functions

```
1.26 closest_preceding_node(f,r)                       internal function as in Chord
1.27   for i = m downto 1 do                           for all fingers of floor f
1.28     if this.lookup_hands(f)[i] ∈ (f(this.ip),f(r))  testing the hand table as in Chord
1.29       return this.lookup_hands(f)[i];             return the finger of floor f
1.30   return this.ip;                                 return the current node ip
```

Figure 3. Pseudocode for multi-floor resource lookup

```
2.01 on receipt of JOIN(f) from ip                     current peer invited by ip to join f
2.02   if this.good_deal(f,ip)                         the invitation is a ''good deal'' (strategy left to implementers)
2.03     this.add_hands(f,⊥);                          add floor f to the hands associative array
2.04     this.add_succ(f,⊥);                           add a successor for floor f to the successor associative array
2.05     this.add_pred(f,⊥);                           add a successor for floor f to the successor associative array
2.06     send FINDSUCC(join,f,this.ip,this.ip) to ip;  find my successor
2.07     receive FOUND(f,ip2) from ip3;                receiving the response
2.08     this.reassign_succ(f,ip2);                    reassign ip3 as my successor at floor f
2.09     for all r ∈ res do                            for all the resources offered by the current node
2.10       send FINDSUCC(join,f,r,this.ip) to ip;      find the node hosting the table entry for r
2.11       receive FOUND(f,ipr) from ip4;              waiting for response
2.12       if ping(ipr)                                test the aliveness of ipr
2.13         update_table(ipr,r,this.ip);              the table stored on ipr is updated with the new bind for r with this.ip

2.14 on receipt of JOINREQ(f) from ip                  the current peer ask to ip to join the floor f
2.15   if this.good_deal(f,ip)                         accept ip at floor f is a ''good deal'' (strategy left to implementers)
2.16     send JOIN(f) to ip;                           accept ip at floor f
```

Figure 4. Pseudocode for join and join request

```
Runned periodically, in order to make some inter-floor business

Join a hot floor (increase local, i.e. node, connectivity)
3.01 join_new_floor()
3.02   select f ∈ (this.dom_hotfloors() \ this.dom_hands());        select one floor to join (strategy left free)
3.03   select ip ∈ this.select_node(f);                            select one node of f to send a join request
3.04     send JOINREQ(f) to ip;                                    send an invitation to ip to join floor f

Invite an hot node to a randomly chosen floor (increase semilocal, i.e. floor, connectivity)
3.05  invite_new_node()
3.06    select  f ∈ this.dom_hands();                              select one floor to invite a node (strategy left free to impl.)
3.07    select ip ∈ this.dom_hotpeers();                           select one hot node to invite (strategy left free to impl.)
3.08    if this.good_deal(f,ip)                                    the invitation is a ''good deal'' (strategy left to implementers)
3.09      send JOIN(f) to ip;                                      send an invitation to ip to join floor f

Create a new floor from scratch (increase global, i.e. BabelChord, connectivity)
3.10  create_new_floor()
3.11    f = new_floor(ip) ;                                        a new floor function is created
3.12    this.add_hands(f,⊥);                                       ⊥ is the new floor
3.13    this.add_pred(f,⊥);                                        ⊥ is the predecessor
3.14    this.add_succ(f,ip);                                       ip itself is the successor
```

Figure 5. Pseudocode for negotiating new joins

information on the requested resource, is sent back to the initiator of the lookup. On receipt of `FOUND` (Lines 1.05-1.09), the initiator checks the aliveness of the node `ip` returned and rewards it locally by updating the reputation of `ip` in its hot peer and hot floor lists. Finally, it remotely reads the values wanted (Line 1.09). Lines 1.26-1.30 detail one local routing step (finding the closest preceding node among my fingers) for the next step. This part is very similar to the Chord local routing step, but integrating the floor information.

3.4 New floor creation (tower building)

Figures 4 and 5 show the creation of new floors: `JOIN` and `JOINREQ`. Lines 2.01 to 2.13 detail the reception of a `JOIN(f)` message which is an invitation to join floor `f` from a node `ip`, already member of `f`. On receipt, the node decides whether it is a *good deal* or not to join `f` (Line 2.02). If this is the case, the current node initiates its join to `f` by sending a `FINDSUCC` message to `ip` and waits for its information required to belong to `f`, namely, its `successor`. On receipt, the current node registers its resources `res` into the floor (Lines 2.09-2.13). Lines 2.14 to 2.16 detail the receipt of the `JOINREQ(f)` message, used to request an invitation. On receipt, the node evaluates the advantages and drawbacks of accepting a new node at floor `f` and sends an invitation in the case of a positive evaluation.

Through Figure 5, we show the pseudocode for proposing, negotiating, and accepting new connections. The following functions are periodically triggered: `join_new_floor` (Lines 3.01 to 3.04) selects one floor (among the hot floors list) and requests an invitation to one node of this floor. `invite_new_node` (Lines 3.05 to 3.09) selects a node to invite at a given floor (this node must reach the `good_deal` requirements). `create_new_floor` (Lines 3.10 to 3.14) initiate the creation of a new floor for future invitations.

Note that any strategy for choosing nodes and floors can be implemented. Strategies may include evaluating nodes and floors according to performance, resources provided, presence rate, etc.

4 Simulation results

To better capture its relevance, we have conducted some simulations of the BabelChord approach. The simulator, written in Python, works in two phases. First, a Babelchord topology is created, with the following properties: (i) a fixed network size (the number of nodes) N, (ii) a fixed number of floors denoted F, (iii) a fixed global *connectivity*, *i.e.*, the number of floors each node belongs to, denoted by C. As a consequence: (i) The nodes are uniformly dispatched among the floors, *i.e.*, each node belongs to C floors uniformly chosen among the set of floors. (ii) Each resource provided by nodes is present at C floors. (iii) The average lookup length within one given floor is $\log((N \times C)/F)/2$.

In a second time, the simulator computes the number of hops required to reach one of the node storing one of the key of a particular resource. Results are given for different values of N, F, and C. Figure 6 gives the results for C=2 and F=10, 50, 100. Note that, in this case, the size of the routing table is in $O(\log((N \times C)/F))$<$O(\log(N))$. The curves clearly demonstrates the logarithmic behavior of such an architecture, even if the average number of hops remains slightly above the Chord reference ($\log(N)/2$). Note also that, the curves suggest that when the ratio C/F decreases, the lookup length increases. This statement is rather intuitive: at each multi-floor routing step, the relative

number of floors reached by a request depends on this ratio. Figure 6 also presents the same experiments with C=5. As expected, the lookup length is slightly reduced compared to the results with C=2. Finally, Figure 7 shows the number of synapses vs. the lookup success rate. Only 5% of synapses made of 2 (resp. 3, 5, 10) floors connections in the whole node population is enough to achieve more than 50% (resp. 60%, 80%, 95%) of exhaustive lookups in the Babelchord network.

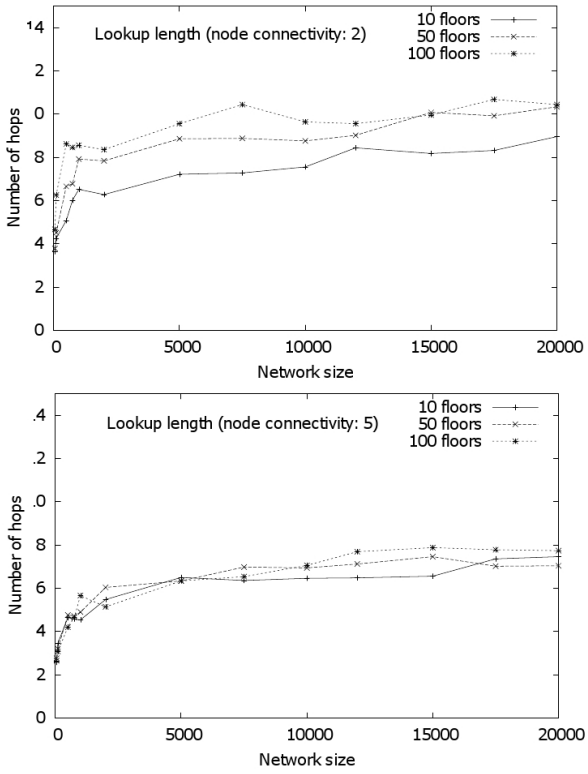

Figure 6. Lookup length, C=2 and C=5

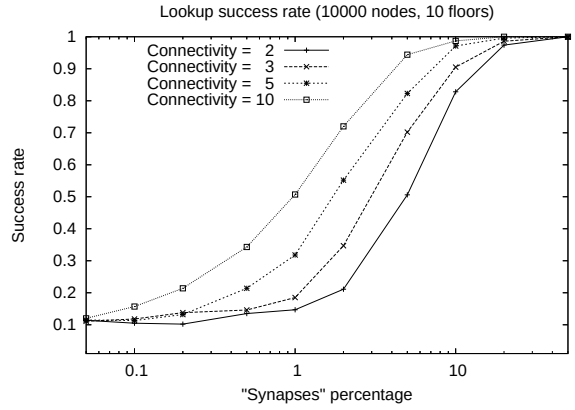

Figure 7. Exhaustiveness, N=10000

5 Conclusions

We have presented Babelchord, a social tower of Chord-based DHTs. Babelchord proposes a simple method for aggregating DHTs (floors) based on intersection nodes, called synapses. Synapses can be created by keeping a list of hot peers and floors and by negotiating between nodes following some "good deal" criterion, left free to every node implementation. Babelchord proposes to aggregate small *structured* overlay networks in an *unstructured* fashion and is an alternative approach to rigid hierarchical peer-to-peer systems.

Simulation confirms that Babelchord scales well with the number of nodes and floors, exhibiting logarithmic behavior. It illustrates the relevance of connecting smaller structured network in an unstructured fashion, within which each node is free to select its neighbors according to its own policy. From a networking point of view, the rationale is simple: *the more synapses we have, the more exhaustive and fast the routing is!*

References

[1] R. Chand, M. Cosnard, and L. Liquori. Powerful resource discovery for Arigatoni overlay network. *Future Generation Computer Systems*, 1(21):31–38, 2008.

[2] A. Datta and K. Aberer. The challenges of merging two similar structured overlays: A tale of two networks. In *Proc. of IWSOS*, 2006.

[3] L. G. Erice, E. W. Biersack, K. W. Ross, P. A. Felber, and G. U. Keller. Hierarchical p2p systems. In *Proc. of Euro-Par*, 2003.

[4] L. G. Erice, K. W. Ross, E. W. Biersack, Pascal A. Felber, and G. U. K. Topology-centric look-up service. In *Proc. of NGC*, 2003.

[5] L. Liquori and M. Cosnard. Logical Networks: Towards Foundations for Programmable Overlay Networks and Overlay Computing Systems. In *TGC*, volume 4912 of *LNCS*, pages 90–107. Springer, 2007.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Adressable Network. In *ACM SIGCOMM*, 2001.

[7] T. M. Shafaat, A. Ghodsi, and S. Haridi. Handling network partitions and mergers in structured overlay networks. In *Proc. of P2P*, pages 132–139. IEEE Computer Society, 2007.

[8] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications. In *ACM SIGCOMM*, pages 149–160, 2001.

[9] B. Y. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiatowicz. Brocade: Landmark Routing on Overlay Networks. In *IPTPS 2002*.