Babelchord: a Social Tower of DHT-Based Overlay Networks

Luigi Liquori, Cédric Tedeschi, Francesco Bongiovanni

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 protocol to distribute and retrieve information at large scale. It builds a large but rigid overlay network without taking into account the social nature and the underlying topology of large platforms, made of the interconnection of many independent smaller networks. Thus, new approaches are required to build overlay networks. In this paper, we propose Babelchord, a more flexible and social overlay interconnecting different Chord networks, which are floors *of a* social tower. *Peers can belong to several floors, allowing this interconnection. By connecting smaller structured overlay networks in an unstructured way, it provides a cost-effective alternative to hierarchical structured P2P systems requiring costly merging. Routing of lookup messages is performed as in Chord within one floor, but a peer belonging to several floors forwards the request to the different floors it belongs to. These co-located peers act as a sort of* neural synapse. *Results from simulations show that Babelchord scales up logarithmically with the number of Babelchord peers. Moreover a small number of synapses is enough to ensure a high exhaustiveness level.*

1 Introduction

A significant part of today's Internet traffic is generated by peer-to-peer (P2P) applications, originally developed for file sharing, it today extends to real-time multimedia communications or high performance computing. Distributed hash tables (DHTs) like the well known Chord [8] protocol, have become the breaking technology to implement scalable, robust and efficient Internet applications. DHTs provide a lookup service similar to a basic hash table. Information is stored as (key, value) pairs and evenly distributed among peers. Any peer can efficiently retrieve the value associated with a given key. DHTs are extremely scalable in the sense that both the number of hops to reach any peer of the network and the size of the routing table scale logarithmically with the number of peers. Periodic mechanisms are used to detect and correct problems following departures and failures of peers, thus ensuring a minimal disruption in dynamic environments.

Chord, like other DHTs, was built to maintain a global overlay network on top of the physical interconnection of many heterogeneous computers. It builds a large but rigid overlay network (a *global* ring) without taking into account the social nature and the underlying topology of large platforms, made of the interconnection of many independent smaller networks. New approaches are required to build overlay networks. Another drawback is its inability to cope with network partitions resulting in two (or more) separated networks. Recovering a single overlay requires to merge all these sub-networks back together, which appears to be particularly costly in terms of time and messages. Moreover, if several distinct Chord networks wish to aggregate their resources, then they also have to merge. To do so, they have to decide which ring will absorb the other one, and which hash function will be used, leading to critical security issues.

More generally, some distant networks may want to cooperate to offer the aggregated set of their resources in a transparent way to the community, without giving the opportunity for one DHT to alter other DHT's data. To overcome the previously described drawbacks of a global overlay, an emerging and promising paradigm is the interconnection of smaller independent networks. In this paper, we propose the Babelchord framework, connecting smaller Chord networks in a simple *unstructured* way via peers co-located in several networks playing the role of *neural synapses*. We build a *social and flexible tower* of independent Chord's floors.

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

Suitable applications. In addition to DHTs traditional use, Babelchord provides a groundwork for some newly introduced classes of applications. We here give two strong examples. (1) Many applications and networks (Psiphon, Tor, . . .) have been recently developed in order to bypass the censorship on the Internet. Babelchord could support such applications by taking advantage of inter-floor routing to bypass software barriers. (2) Social networks, such as Facebook or LinkedIn are still based on a client-server architecture; very often those sites are down for maintenance. Babelchord could represent a scalable and reliable alternative to decentralize such social networks.

Related work. Apart from Chord protocol, our proposal is inspired by the Arigatoni overlay network [1, 5] built over a number of *agents*, organized in *colonies*, and ruled by a broker *leader*, democratically elected or imposed by system administrators. An agent asks the broker to log in the colony by declaring the resources it offers to the community. Colonies can recursively be considered as evolved agents who can log in an outermost colony governed by a *super-leader*. Once logged in, an agent can ask the broker for other resources. Brokers route requests by filtering their resource routing table, and forwarding the request first inside its colony, and second outside, via the proper super-leader. Once the requesting agent receives the information on the requested resources, the real resource exchange is performed directly between agents, in a pure P2P fashion.

Hierarchical overlay networks have been recently intensively studied. Brocade [9] is a two-level DHT whose key idea is to build local DHTs inside which some leaders are elected, according to metrics such as CPU or bandwidth, to enter an *interdomain* DHT connecting local DHTs together. Authors in [3] generalize it to an arbitrary number of levels, and adapted it to the IP-numbering [4]. A different way, introduced in [6] consists in using a set of distributed reliable peers, called *landmarks*, used to dispatch peers in virtual *bins* considering a given metric, such as the latency. Each peer computes the latency between itself and each landmark, sorts them and thus finds its own bin. The intuition behind this is that peers that are close to each other have similar landmark measurements. To achieve exhaustiveness, hierarchical approaches require mergers. Authors in [7] focused on merging several similar overlays together. However, as argued in [2], such mechanisms generate a significant communication overhead, not to mention the time required before converging towards a usable single overlay.

2 Babelchord's social tower

We here describe the Babelchord's features. Joining and creating floors is governed by negotiations and social behaviors, as encountered in the recent social networking phenomena. Babelchord extends Chord in the following points:

Peers and floors. Peers can belong to several distinct floors. A peer wishing to join a floor comes with a list of resources it offers to this floor's community. The rationale is then simple. The more (relevant) resources the peer injects into the floor, the higher the probability to successfully enter it is. This operation can be based on a tit-for-tat strategy, commonly used in economics or social sciences. It is clear that the more floors the peer is registered to, the larger its routing table will be. Nevertheless, we can assume that the numbers of floors a peer belongs to will be pretty low. Moreover it is the peer's choice to belong to more floors, thus it knows it has the capacity to deal with a routing and storage overhead. Each floor has a proper hash function in order to perform consistent hashing of peers and keys within it. Peer variables used to perform routing, like its predecessor on the ring (`pred`), its successor on the ring (`succ`) and the entries of its routing table (`fingers`), must be upgraded in order to take into account the multi-floor extension.

Multi-floor routing. When a peer lookups a resource on a given floor (using the floor's hash function), a Babelchord routing is launched. A unique tag identifier for the query is created. If the routing goes through a synapse peer, then the lookup is forwarded in parallel to all the floors the synapse peer belongs to, otherwise the routing goes on as in a standard Chord ring. To do this, each peer needs to know the hash function of the floor. This means that keys need to be hashed at every floor change. The rationale of this propagation is simple: the more floors you explore, the higher the probability of success will be. It is important to notice that while the searchwithin a single floor lookup is *exhaustive* and *logarithmic* in the number of peers, 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).

Limiting cycles during lookup. In order to avoid lookup cycles when doing cross-floors search, each peer maintains a list of already processed requests' tag in

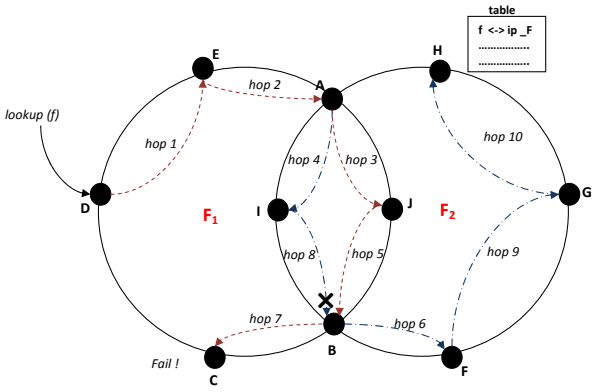

Figure 1. Multi-floor routing example

order to discard previously seen queries. Also, each lookup process has a Time-To-Live (TTL) value which is decreased each time we cross floors' boundaries. These two features prevent 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 exhaustiveness.

Peers and floors selection. Each peer maintains a list of *hot peers* involved in successful lookups and a list of *hot floors* similarly responsible for a significant amount of successful lookups. Periodically, every peer can either *(i)* select a new *hot* floor to join thus increasing the current peer's connectivity, *(ii)* select a *hot* peer to invite in one its floors, *(iii)* create a new floor.

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`. Peers are depicted by capital letters, *i.e.* `A,B,..I`, and `J`. Assume that every peer offers a single resource, *i.e.* `a,b,...,i`, and `j`. The two Babelchord floors intersect via two synapses peers `A` and `B`. Hops are labeled by an integer `i` denoting a global time of arrival on a given peer. Since intra-floor routing is performed as in Chord, fingers and hand tables 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 peers and resources via the two different hash functions. A `lookup(f)` is received by peer `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 peers `I`, `J`, and `B` (hops `3,4,5` and `8`, respectively). Since peer `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 peers `C` (routing failure on floor `F1`), on peers `F,G`, and finally terminate on peer `H` hosting the value `ip_F` offering the resource `f`.

3 Babelchord's protocol

We present the details of the algorithms to build Babelchord. First, we focus on how Chord's variables are extended to support a multi-floor architecture. Then, in Section 3.2, we give the Babelchord lookup. Finally, in Section 3.3, we give an idea on how peers can negotiate joins and creation of floors. Recall that words *floor* and *ring* refer to the same object.

3.1 Data structures for every peer

Let `f`, `g` represent 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 previously seen requests. 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. The `succ`, `pred`, and `hands` structures contain predecessor, successor and finger information for each floor. Finally, `hotpeers` and `hotfloors` contain information collected through lookups about peers for potential collaborations and floors for potential participation. Here is the detailed list of variables:

Node's Data structures

`f(-)`	$\stackrel{def}{=}$	`hash(f)(-)` with `hash:int->Sha1`	hash function
`tag`	$\stackrel{def}{=}$	`(int)`*	list of unique tags identifying a Babelchord packet
`res`	$\stackrel{def}{=}$	`(r)`*	list of resources offered by the current peer
`table`	$\stackrel{def}{=}$	`(r,(ip)`*`)`*	the associative array of key, value pairs
`succ`	$\stackrel{def}{=}$	`(f,ip)`*	associative array of successors at floor f
`pred`	$\stackrel{def}{=}$	`(f,ip)`*	associative array of predecessors at floor f
`fingers`	$\stackrel{def}{=}$	`[ip]`	array of ip addresses
`hands`	$\stackrel{def}{=}$	`(f,fingers)`*	associative array of fingers at floor f
`hotpeers`	$\stackrel{def}{=}$	`(ip,(f)`*`)`*	associative array of peers view by some floors
`hotfloors`	$\stackrel{def}{=}$	`(f,(ip)`*`)`*	associative array of floors view by some peers

```
The Babelchord's protocol
1.01 on receipt of LOOKUP(r) from ip do                                looking for peers 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,⊥,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 g ∈ this.dom_hands() do                               for all floors of current peer
1.20       if g(r) ∈ (g(this.ip),g(this.get_succ(g))]                  test if arrived, as in Chord
1.21         send FOUND(g,this.get_succ(g)) to ip;                     found a peer hosting an entry for r
1.22         exit forall;                                              stop the routing: ''game over''
1.23       else
1.24         ip4 = this.closest_preceding_node(g,r);                   internal Chord routing
1.25         send FINDSUCC(t,g,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 peer ip
```

Figure 2. 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 peer
2.10       send FINDSUCC(join,f,r,this.ip) to ip;                      find the peer hosting the table entry for r
2.11       receive FOUND(f,ip5) from ip4;                              waiting for response
2.12       if ping(ip5)                                                test the aliveness of ip5
2.13         update_table(ip5,r,this.ip);                              the table stored on ip5 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 3. Pseudocode for join and join request

3.2 The lookup protocol

The multi-floor lookup is illustrated in Figure 2. Lines 1.01 to 1.04 initiate a `lookup` on a resource `r`. After creating a new unique tag for this request, the current peer 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 peer storing the sought key. On receipt of a `FINDSUCC` message (Line 1.10), the current peer distinguishes two types of messages:

Join routing (Lines 1.12-1.16). When the first element of the message is a `join` tag, it means that the lookup serves a join purpose. The request is then routed as in Chord's `join`, and corresponds to the routing process of either a resource registration or a peer insertion. A `FOUND` message is then 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

```
Runned periodically, in order to make some inter-floor business
Join a hot floor (increase local, i.e. peer, 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 peer 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 peer 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 peer (strategy left free to impl.)
3.07    select ip ∈ this.dom_hotpeers();                           select one hot peer 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 4. Pseudocode for negotiating new joins

current peer 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, otherwise, the request tag is saved. When a search is successful, a `FOUND` message (containing the address of the peer responsible for the requested information) is sent back to the lookup's initiator. On receipt of `FOUND` (Lines 1.05-1.09), the initiator checks the aliveness of the peer and updates its hot peer and hot floor lists according to its satisfaction. Finally, it remotely reads the values wanted (Line 1.09). Lines 1.26-1.30 detail one local routing step (finding the closest preceding peer among my fingers) for the next step, as in Chord, but it includes the floor information.

3.3 New floor creation (tower building)

Figures 3 and 4 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 peer `ip`, already member of `f`. On receipt, the peer decides whether it is a *good deal* or not to join `f` (Line 2.02). If this is the case, the current peer 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 peer 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 peer evaluates the advantages and drawbacks of accepting a new peer at floor `f` and sends an invitation in the case of a positive evaluation.

Through Figure 4, 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 peer of this floor. `invite_new_node` (Lines 3.05 to 3.09) selects a peer to invite at a given floor (this peer 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 the strategy for peers and floors selection can be based on any criteria (performance, churn rate, resources' relevance...).

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 peers) N, (ii) a fixed number of floors denoted F, (iii) a fixed global *connectivity*, *i.e.*, the number of floors each peer belongs to, denoted by C. As a consequence: (i) the peers are uniformly dispatched among the floors, *i.e.*, each peer belongs to C floors uniformly chosen among the set of floors, (ii) each resource provided by peers 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 peer storing 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 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 5 shows the number of synapses vs. the lookup success rate. Only 5% of the whole population is a synapse connecting 2 (resp. 3, 5, 10) floors. However, this is enough to achieve more than 50% (resp. 60%, 80%, 95%) of exhaustive lookups in the Babelchord network.

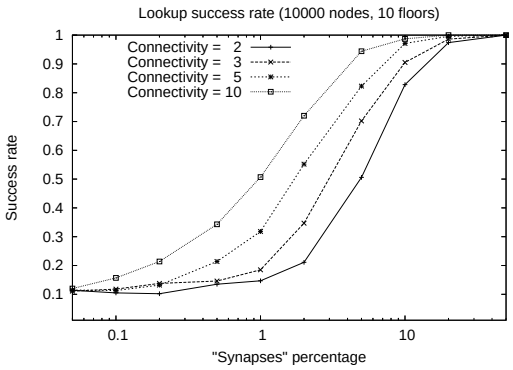

Figure 5. Exhaustiveness, N=10000

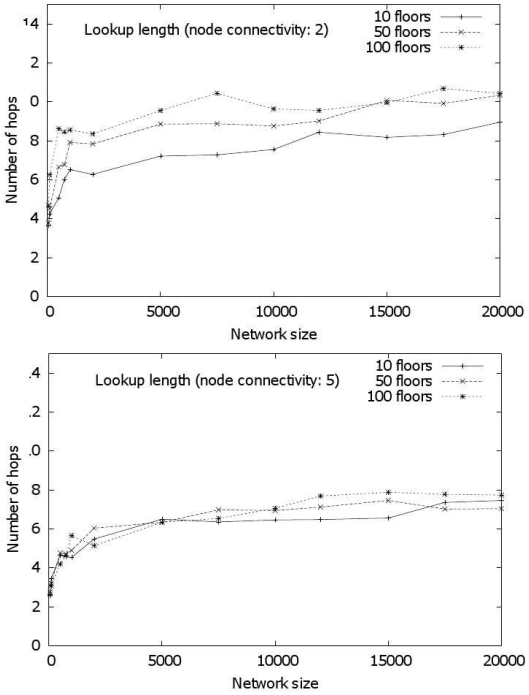

Figure 6. Lookup length, C=2 **and** C=5

5 Conclusion

Babelchord is a social tower of Chord rings. It provides a simple method to build the flexible and social next generation of overlay networks. Babelchord aggregates smaller *structured* overlay networks in an *unstructured* fashion based on intersection peers, called *synapses* allowing to explore the whole overlay without the need for hierarchical systems or costly merging. Simulations show that Babelchord is scalable while offering an exhaustive search at the cost of only few synapses, thus establishing the relevance of connecting smaller structured network in an unstructured fashion. Next steps are a complete analysis of such topologies, more simulations, and an actual implementation followed by real deployments of this promising paradigm.

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. Erice, E. Biersack, K. Ross, P. Felber, and G. Keller. Hierarchical P2P Systems. In *Euro-Par*, 2003.

[4] L. Erice, K. Ross, E. Biersack, P. Felber, and G. Keller. Topology-Centric Look-up Service. In *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. 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. Zhao, Y. Duan, L. Huang, A. Joseph, and J. Kubiatowicz. Brocade: Landmark Routing on Overlay Networks. In *IPTPS*, 2002.