On the Implementation of a Content–Addressable Network

Article · January 2010

Source: OAI

3 authors:

Alexandru Popescu
Blekinge Institute of Technology
25 PUBLICATIONS **79** CITATIONS

Dragos Ilie
Blekinge Institute of Technology
44 PUBLICATIONS **264** CITATIONS

Demetres D. Kouvatsos
University of Bradford
227 PUBLICATIONS **1,489** CITATIONS

Some of the authors of this publication are also working on these related projects:

Project — Modelling and Analysis of Networks Performance and Security Trade-offs View project

Project — A Novel Multiple Access QKD Network for Secure Communication View project

On the Implementation of a Content-Addressable Network

Alexandru Popescu
Dept. of Computing
School of Informatics
University of Bradford
Bradford, West Yorkshire, UK
E-mail: A.O.Popescu@Bradford.ac.uk

Dragos Ilie
Dept. of Telecommunication Systems
School of Engineering
Blekinge Institute of Technology
Karlskrona, Sweden
E-mail: dragos.ilie@bth.se

Demetres Kouvatsos
Dept. of Computing
School of Informatics
University of Bradford
Bradford, West Yorkshire, UK
E-mail: D.D.Kouvatsos@scm.brad.ac.uk

Abstract—Over the last years, the Internet has evolved towards becoming the dominant platform for deployment of new services and applications such as real time and multimedia services, application-level multicast communication and large-scale data sharing. A consequence of this evolution is that features like robust routing, efficient search, scalability, decentralization, fault tolerance, trust and authentication have become of paramount importance.

We consider in our paper the specific case of structured peer-to-peer (P2P) overlay networks with a particular focus on Content-Addressable Networks (CANs). An investigation of the existing approaches for structured P2P overlay networks is provided, where we point out their advantages and drawbacks. The essentials of CAN architecture are presented and based on that, we report on the implementation of a CAN simulator. Our initial goal is to use the simulator for investigating the performance of the CAN with focus on the routing algorithm. Preliminary results obtained in our experiments are reported as well.

I. INTRODUCTION

P2P overlay networks have become very popular over the last years due to features that makes them suitable for the development or deployment of new services like overlay multicast communication, large-scale data sharing and content distribution. P2P networks exhibit three fundamental features: self-organization, symmetric communication and distributed control [1]. Compared to the classical client-server data sharing, the P2P approach has the characteristics of a disruptive technology in the sense that it offers the opportunity for aggregation of very large storage and processing resources while minimizing entry and scaling costs. The main consequence of this is the presence of a much higher risk for failures and therefore different methodologies have been developed for increasing the robustness of P2P networks [1].

This paper is structured as follows. In Section II we provide a short overview of the state of the art research in P2P systems. This is followed in Section III by a description of how CANs work. Our own implementation of the CAN routing protocol along with simulation results are described in Section IV. In Section V we present some brief conclusions and ideas for future work.

II. P2P SYSTEMS: STATE OF THE ART

Some of the most important issues related to P2P systems that must be considered in the research and design of such systems are concerning naming, routing, congestion control and security. Other important research issue for P2P networks is the indexing system used in a specific P2P network. By an index we mean a $\{key, value\}$ pair in the general sense, where the key is a concise reference (*e.g.*, a name or numerical identification) to a data object and the value denotes a node or a collection of nodes that either store the data object or have knowledge of its whereabouts. Indexes are generally partitioned into three classes, namely local, centralized and distributed. Each of these classes has own advantages and drawbacks, as it is shown in [2]. A P2P local index means that a peer only keeps track of its own data and it does not receive any other reference for data at other nodes, *e.g.*, like in the case of the first Gnutella version. In a system with centralized index, there is a single server that keeps track of references to data existent on more peers, as it was the case of Napster. The most advanced and popular systems are however the ones with distributed indexes, where pointers to the target data reside on several nodes.

An important mechanism connected to indexing is the search system. There are a number of search systems existing in today's P2P networks, which can be partitioned into two main classes [2]–[5]:

- Semantic-free systems, which are data-centric. These systems are generally based on distributed hash tables (DHTs), which build a mathematical relatioship between the key and the value in the index. The DHT used induces a specific form of routing geometry such as Plaxton trees, rings, tori, butterflies, de Brujin graphs and skip graphs. Examples of applications using semantic-free systems are BitTorrent, e-Mule, Coral Content Distribution Network and OceanStore.
- Semantic-based indexes are human-readable (*e.g.*, they can be keywords) and able to capture object relationships such as those between the document name and the data itself. The drawback, in this case, is related to the fact that scarce data objects may not be found. Examples

of applications using semantic-based indexes are Napster and Gnutella.

If the network uses a DHT substrate, the node identification is typically mapped on the key space. Additionally, each participating node becomes responsible for a subset of the key space. In this scenario, searching for a key becomes similar to routing a message towards a node and we speak about key-based routing (KBR) or DHT-routing.

There are two main classes of P2P networks with reference to routing substrate or geometry [3]: unstructured networks and structured networks. Unstructured networks, also called "first generation" networks, used initially flooding and random walks for routing. More recently, these methods were enhanced to alleviate the amount of overhead traffic by using super-peers, clustering, selective forwarding or a combination thereof. Examples of such networks are Gnutella and Kazaa. The unstructured networks have the main advantage in the form of simplicity but they have several important drawbacks like, *e.g.*, large routing costs, risk of failure in finding the requested data objects, difficulties in providing scalability when handling increased rates of aggregate queries or when the system size increases. On the other hand, structured networks, also called "second generation" networks, use KBR schemes that reduce the routing cost and provide a bound on the number of hops required for localizing a target data item. A good definition of structured P2P networks is that the P2P network topology is tightly controlled and data objects are placed at specific locations selected such as to obtain better query performance. Examples of such networks are Plaxton, Pastry, Tapestry, Chord and CAN [1], [2].

Structured P2P systems have an architecture based on using the DHT as a layer, as shown in Figure 1. Information about data objects is deterministically placed in this layer, at peers identified by the key of the specific data object. Keys are mapped by the overlay network protocol to a unique peer in the overlay network. The search service is similar to a hash table, in the sense that $\{key, value\}$ pairs are stored in the DHT and every participating node can efficiently retrieve the value associated with a given key. DHT systems have several advantageous features, *e.g.*,, decentralization, scalability and fault tolerance.

The main criticism of structured systems surrounds the issue of peer transience (*i.e.*, churn), which has a serious impact on the network performance. Other drawbacks of these systems are significantly higher overheads than those associated with unstructured systems as well as the lack of support for keyword searches and complex queries. Because of this, decentralized unstructured P2P overlay networks seem today to be more popular [1]. However, recent efforts towards the development of a unification platform for different DHT-systems are making structured networks more attractive [6], [7]. Such a platform will provide a KBR-based application programming interface (API) coupled with a basic DHT service model to easily deploy DHT-based applications.

Another important issue is regarding the query mechanisms used in P2P systems and ways to optimize them. A query
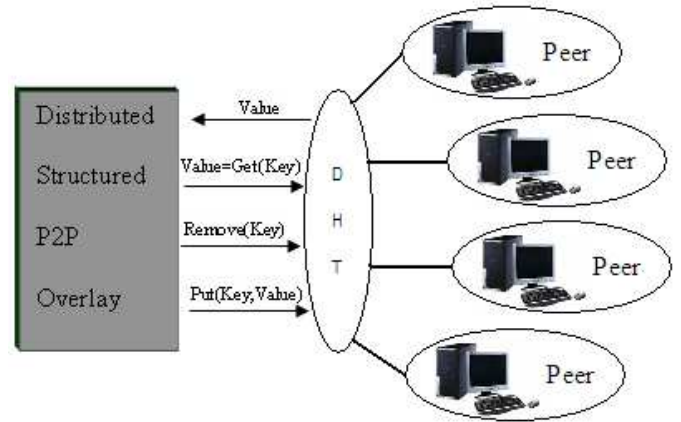

Fig. 1. Structured P2P system with DHT substrate

mechanism is used to construct efficient searches from user input. There are today several query systems, *e.g.*, range queries, multi-attribute queries, join queries and aggregation queries, each of them with own advantages and drawbacks [2]. The challenge is to develop new models for query optimizations for large networks in the order of thousands and tens of thousands of servers and millions of clients [8].

An interesting recent development is also the fact that the border between unstructured and structured networks seems to become less and less distinct. It has been found that most unstructured networks evolve to include some form of structure, typically hierarchical (*e.g.*, regular nodes and supernodes), in order to enhance their performance [2].

III. CAN ESSENTIALS

Content-Addressable Networks (CANs) are a robust, scalable, distributed systems designed for efficient search of data stored in a DHT. The key space of the CAN is a n-dimensional Cartesian coordinate space. The space wraps around the edges of each dimension, thus creating a n-dimensional torus geometry. For the remaining of this paper, we focus on a 2-dimensional coordinate space along the x, y coordinates with the implicit assumption that the concepts presented here can be extended to higher dimensions.

Each node in the CAN has an identifier that is mapped to a point P in the key space. In addition, the node is responsible for its *zone*, which is a rectangular portion of the key space that surrounds the point P. Furthermore, the node has information about adjacent zones and the nodes that are responsible for them. The nodes are able to route messages in the CAN overlay utilizing only information about neighbouring nodes and their zones. Since the CAN space is a 2-dimensional coordinate grid, this becomes a matter of routing along a straight line. We present our routing implementation in Section IV-A.

The construction of a CAN overlay consists of three steps: bootstraping, finding a zone and joining the overlay routing.

These correspond to the functions[1] BOOTSTRAP, FINDZONE and JOINROUTING outlined in Algorithm 1.

Algorithm 1 CAN construction

1: $S \leftarrow$ BOOTSTRAP
2: $c \leftarrow$ RAND(S)
3: $P \leftarrow$ RAND(X,Y)
4: $Z, N \leftarrow$ FINDZONE(c, P)
5: JOINROUTING(N)

6: **procedure** BOOTSTRAP
7:      Contact a DNS server d
8:      $b \leftarrow d$.LOOKUP(CAN domain)   ▷ bootstrap node
9:      $c \leftarrow b$.GETCANNODES   ▷ set CAN nodes
10:      **return** c
11: **end procedure**

12: **procedure** FINDZONE(c, P)
     Route JOIN message towards point P via node c
13:      $Z, N \leftarrow p$.GETZONE   ▷ P is in p's zone
14:      **return** Z, N
15: **end procedure**

16: **procedure** JOINROUTING(N)
17:      Send soft updates to all nodes N
18: **end procedure**

19: **procedure** LOOKUP($domain$)
20:      Lookup IP address ip associated with $domain$
21:      **return** ip
22: **end procedure**

23: **procedure** GETCANNODES
24:      **return** subset of known CAN nodes
25: **end procedure**

26: **procedure** GETZONE
27:      Give up half of own zone, Z, to calling node
28:      Collect the set N of neighbours to half-zone Z
29:      **return** Z, N
30: **end procedure**

The purpose of bootstraping is to enable the node to find the IP address of an existing CAN node. The authors of [9] do not insist on a particular bootstraping procedure, but recommend something similar to Your Own Internet Distribution (YOID) [10]. It is therefore assumed that a Domain Name System (DNS) lookup of the CAN domain will reveal the IP address of a bootstrap node. The bootstrap node is used to obtain a set of IP addresses corresponding to active CAN nodes.

---

[1] In the pseudocode presented in this paper we use the convention that a procedure name preceeded by a variable name and a dot indicate a remote function call (*e. g.,* n.LOOKUP means that function LOOKUP runs on node n).

Upon succesfull completion of the bootstrap procedure, the joining node must find its own zone. To do this, it starts by picking randomly the IP address of a node b from the list supplied by the bootstrap node. This is shown on line 2 of Algorithm 1. The function RAND shown there selects randomly one element from each set passed into its parameters. On line 3, the function RAND takes two arguments, X and Y, which are the set of points along the x- and y-axis, respectively. In this case RAND selects randomly a coordinate along the x-axis and another one along the y-axis and assignes the values to the point P. P becomes implicitly the identifier of the joining node.

The next objective of the new node is to find a zone in the CAN, which contains P. To do this, the new node assembles a JOIN message and then asks node c to route it towards point P. Upon receiving the JOIN message, node p, which is responsible for the zone where P belongs, executes the GETZONE procedure. The result of the procedure is the division of p's zone into two equal parts, where p keeps one part and relinquishes the other, Z to the new node. The procedure returns Z along with set of neighbours N responsible for zones adjacent to Z.

When the new node has found its zone, all nodes in the system send an immediate update to their neighbours to inform them if any change has occured in their zone. This is followed by periodic updates update messages where similar information is exchanged.

In the case a node leaves the CAN, one of its neighbours takes its zone. Zone information is then refreshed during periodic routing updates. The absence of routing messages from a neighbour is taken as indication of a node failure. In this case a takeover mechanism is initiated. The purpose of the takeover mechanism is to assign the zone of the failed node to one of its neighbours in a consistent manner (*i. e.,* preventing several nodes from attempting simultaneously to take over the zone) [9].

IV. CAN IMPLEMENTATION

We have implemented the CAN construction algorithm as well as basic routing functionality and are currently working on implementing the ability to store data (*i. e.,* values) at CAN nodes.

Our implementation creates a 2-dimensional $[0, 1] \times [0, 1]$ coordinate space for the CAN. The first node to join the CAN becomes the owner of the entire CAN space and it is assigned the coordinates $\{0,0\}$. Additionaly, it is selected as bootstrap node. This is just a matter of convenience and has no impact on anything else other than the bootstraping procedure. Each additional node to be added follows the description of Algoritm 2.

A. Routing in CAN

As previously mentioned, if the random point P is not located within the zone owned by the bootraping node, then a JOIN request must be routed through the CAN. Starting from the bootstrap node, the routing operation is accomplished

Algorithm 2 Add node to CAN

1: $P \leftarrow$ RAND(X,Y)
2: $N \leftarrow$ ADDNODE(P)
3: JOINROUTING(N)

4: **procedure** ADDNODE(P)
5:     **if** $P \in$ c **then**   ▷ P is in origo node c's zone
    Check if a neighbor of node c, owner of point P has a
    bigger zone, return biggest to split and save to p
6:         $p \leftarrow c$.PLUS1ZONE(P)
    Split p's zone, either still origo node or one of its neigh-
    bors
7:         $Z, N \leftarrow p$.GETZONE   ▷ P is in p's zone
8:     **else**   ▷ P is not in origo node c's zone
    Owner of zone where point P lie needs to be found
9:         $Z, N \leftarrow$ FINDZONE(c, P)
10:     **end if**
11:     **return** N
12: **end procedure**

13: **procedure** PLUS1ZONE(P)
    Compare zone area sizes of current node with its neigh-
    bors and return node p with the biggest zone area
14:     **return** p
15: **end procedure**

16: **procedure** FINDZONE(c, P)
    Route returns owner of point P
17:     $p \leftarrow c$.ROUTE(P)   ▷ Route through CAN
    Check if a neighbor of node p, owner of point P has a
    bigger zone, return biggest to split and save to p
18:     $p \leftarrow p$.PLUS1ZONE(P)
19:     $Z, N \leftarrow p$.GETZONE   ▷ P is in p's zone
20:     **return** Z, N
21: **end procedure**

22: **procedure** GETZONE
23:     Give up half of own zone, Z, to calling node
24:     Collect the set N of neighbours to half-zone Z
25:     **return** Z, N
26: **end procedure**

27: **procedure** ROUTE(P)
    Route JOIN message through CAN towards point P via
    node c, return owner of point P
28:     **return** p
29: **end procedure**

30: **procedure** JOINROUTING(N)
31:     Send soft updates to all nodes N
32: **end procedure**

---

by attempting to follow a straight line through the Cartesian space from source to destination. This is done simply by greedy forwarding to the neighbor whose zone extends furthest towards the destination. In a 2-dimensional coordinate space two nodes are neighbors if their coordinate spans overlap along one dimension and abut along the other dimension.

In our current CAN-implementation, the routing operation always starts from the bootstrap node located at the origo in the coordinate space. For each node on the path, the algorithm computes the extremities of each neighbor zone to find the one with the shortest distance to P along a straight line. For each neighbor zone the distance to P is computed from eight different points: four of them are the zone corners and the remaining four are the middle points on each zone border. The neighbour, whose zone contains the point with the shortest distance to P, is a candidate for the next hop on the path. To avoid loops, the algorithm never selects the same node twice. Therefore, if a candidate is already an intermediate node on the path, the algorithm selects the next eligible candidate.

When the number of CAN nodes grows, zones become rectangular areas of different size. In this scenario, there is an increasing probability that the algorithm will become trapped in a zone where all neighbor nodes are already intermediate nodes on the path. In such a case the routing algorithm enters a recovery mode that forces the JOIN message to backtrack its steps one at a time. For each step back all the neighbors of the node at that particular step are checked for an alternative path towards the destination. The recovery mode continues until a valid path is found.

B. Routing Performance

The performance of the routing algorithm outlined in the previous section depends on how often the algorithm enters the recovery mode. To get preliminary information of the algorithm performance we have run the two experiments.

In both experiments we start with an empty CAN. Then we add up to 100 nodes to the CAN in the first experiment, and up to 1000 nodes to the CAN in the second experiment. For each node n to be added we keep a variable a_n that counts the number of times the routing algorithm enters the recovery mode for that node. We run each experiment 100 times and compute the average number of times each node n enters the recovery mode. We call this statistic *mean routing failures per node n* and we denote it by A_n. More formally,

$$A_n = \frac{1}{100} \sum_{r=1}^{100} a_{n,r} \tag{1}$$

where $a_{n,r}$ indicates the number of times node n enters recovery mode during simulation run r. The plots for A_n when the number of simulated nodes is 100 and 1000, are shown in Figure 2 and Figure 3, respectively. We can observe that the performance of the routing algorithm degrades with an increasing number of nodes. This is an indication that our CAN implementation faces scalability problems unless the routing algorithm improves.
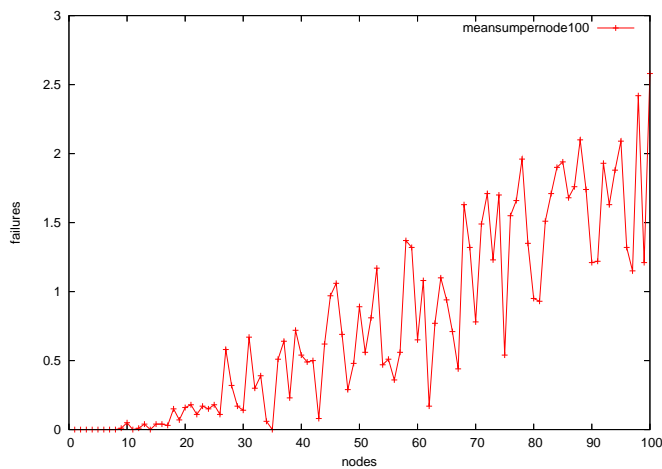
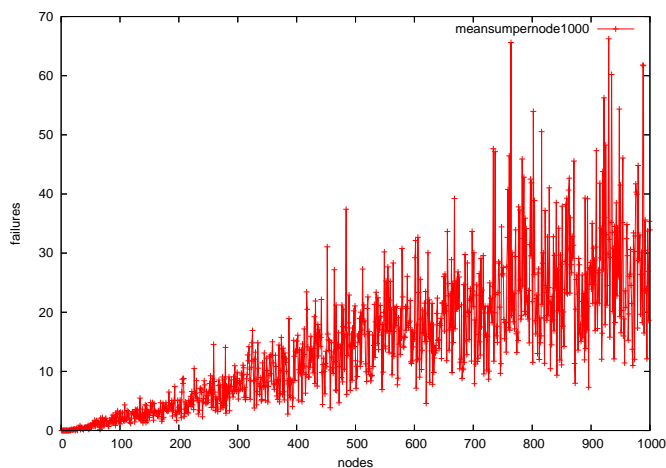Fig. 2. Mean routing failures: 100 nodes over 100 simulation runs

Fig. 3. Mean routing failures: 1000 nodes over 100 simulation runs

The reason behind the degrading performance is that when the number of peers increases, CAN zones become smaller. The routing algorithm becomes more likely to be trapped in a zone with neighbours already visited. In order to avoid a routing loop, the algorithm backs out incrementally, until it finds a neighbour not visited previously.

V. CONCLUSIONS

There is still much work remaining to complete this CAN implementation. It is imperative to address the routing performance issue in order to improve the overall scalability of the system. We have recently become aware of a more efficient way to perform CAN routing [11], based on identification of CAN zones with binary sequences. We plan to evaluate this algorithm as part of our future work.

In the long term we would like to move the CAN implementation into a simulator such as MyNS [12] or OmNeT++ [13]. This will allow us to study CAN's performance in the presence of large amounts of churn. Furthermore, we would like to compare CAN performance to the performance of other structured-based overlays such as Pastry and Tapestry.

REFERENCES

[1] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay networks schemes," *IEEE Communications Surveys and Tutorials*, vol. 7, no. 2, pp. 72–93, 2nd quarter 2005.
[2] J. Risson and T. Moors, "Survey of research towards robust peer-to-peer networks: Search methods," RFC 4981, Sep. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4981.txt
[3] H. Balakrishnan, F. M. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in P2P systems," *Communications of the ACM*, vol. 46, no. 2, pp. 43–48, Feb. 2003.
[4] K. P. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The impact of dht routing geometry on resilience and proximity," in *Proceedings of the ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003, pp. 381–394.
[5] B. Yang and H. Garcia-Molina, "Efficient search in peer-to-peer networks," in *Proceedings of ICDCS*, Vienna, Austria, Jul. 2002.
[6] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a common API for structured peer-to-peer overlays," in *Proceedings of IPTPS*, Berkeley, CA, USA, Feb. 2003.
[7] D. Ilie and A. Popescu, "A framework for overlay QoS routing," in *Proceedings of 4th Euro-FGI Workshop*, Ghent, Belgium, May 2007.
[8] D. Kossmann, "The state of the art in distributed query processing," *ACM Computing Surveys*, vol. 32, no. 4, pp. 422–469, Dec. 2000.
[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of ACM SIGCOMM*, San Diego, CA, USA, Aug. 2001, pp. 161–172.
[10] P. Francis, "Yoid: Extending the internet multicast architecture," Apr. 2000, unpublished paper. [Online]. Available: http://www.isi.edu/div7/yoid/docs/yoidArch.ps.gz
[11] J. Eberspächer, R. Schollmeier, S. Zöls, and G. Kunzmann, "Structured P2P networks in mobile and fixed environments," in *Proceedings of HET-NETs*, Ilkley, UK, Jul. 2004.
[12] "Myns simulator." [Online]. Available: http://www.cs.umd.edu/users/suman/research/myns/index.html
[13] "Omnet++ simulator." [Online]. Available: http://www.omnetpp.org/