An Overview of Content Distribution and Content Access in Peer-to-Peer Systems

Gregor von Bochmann and Guy-Vincent Jourdan

School of Information Technology and Engineering
University of Ottawa
800, King Edward Avenue
Ottawa, ON, K1N 6N5 CANADA
{bochmann,gvj}@site.uottawa.ca

RÉSUMÉ. Il existe différents schémas de distribution de contenu dans les systèmes pair à pair actuels. Ces différents schémas reposent sur des architectures diverses et permettent différents types d'accès. Nous faisons un survol de ces architectures et de leur impact sur l'efficacité de la recherche de contenu, ainsi que sur la flexibilité de cette recherche. Nous donnons des exemples de systèmes pair à pair courants pour chacune de ces architectures. Nous proposons ensuite une étude d'un système pair à pair récent, eQuus, qui groupe les pairs en cliques. Nous identifions quelques-unes des limites de cette approche, et proposons des solutions.

ABSTRACT. There are several different models for content distribution in current peer-to-peer (P2P) systems, based on different architectures and allowing different types of access. We give an overview of these architectures and their impact on content search efficiency and flexibility. We give examples of current P2P systems for each approach. - We then provide an overview of a recently proposed clique-based system, eQuus. We identify some of its limitations and propose solutions to these shortcomings.

MOTS-CLÉS : systèmes pair à pair, architecture des réseaux pair à pair, schémas de distribution de contenu, regroupement géographique.

KEYWORDS: Peer-to-peer systems, peer-to-peer networks architecture, content distribution models, geographical grouping.

1. Introduction

There are several different models for content distribution in current peer-to-peer (P2P) systems, based on different architectures and allowing different types of access. In this paper, we propose an overview of these architectures and their impact on content search efficiency and flexibility. We give examples of current P2P systems and their approaches.

We then provide an overview of a clique-based system, eQuus, a recently introduced P2P system. We identify some limitations with this system and propose some solutions.

2. Peer Architectures in P2P Systems

P2P systems are by essence decentralized, distributed systems used e.g. for data storage, data distribution, fault tolerance or censorship fighting. The architectures of the peers can be very different from one system to the next, including the level at which they are really decentralized. We can distinguish systems that are *purely decentralized* and systems that are *hybrid* (Pourebrahimi *et al.*, 2005).

First generation P2P systems were of hybrid type, with a centralized server orchestrating the indexing of peers and data, while the data exchange was occurring between the distributed peers. We can identify two types of hybrid systems:

- Centralized indexing systems have a central directory of all peers and the data each peer stores. Each peer informs the central server about the data it stores, and queries that server when trying to locate data. The central server puts the peer requesting the data and the peer holding a copy of it in contact and does not intervene in the actual data transfer process. The once popular Napster was built following this model.

- Hybrid systems introduce the concept of *super-peers* which maintain a centralized index of a group of peers which they manage. These super-peers have the task of routing requests towards the target, but again, once the two end-peers are identified, the data exchange occurs between these two peers only. The system Kazaa is an example of such an architecture.

Systems having a centralized server can be searched easily and rapidly, and they are easier to build and maintain. However, this solution suffers from several limitations, in particular the fact that it is not scalable beyond what the centralized server can handle, and the fact that the temporary unavailability of the centralized server puts the entire system to a halt. In addition, with this architecture the centralized server is an unwanted potential global control point, making the entire P2P network vulnerable to “easy” censorship.

The second generation P2P systems such as Freenet (Clarke *et al.*, 2002), Chord (cite), CAN (site), Pastry (Rowstron *et al.*, 2001) and Tapestry (Zhao *et al.*, 2004) are purely decentralized and to not require any central server. They are inherently scalable and can offer a much better defense against censorship. Having no centralized information creates new challenges, in particular for offering efficient routing and for ensuring that the architecture is globally maintained as peers join and (silently) leave the network. In the following, we survey the different types of architectures that are commonly found in these second generation P2P systems.

2.1. *Peer vs Host vs Data*

Before describing the P2P architectures, we should clarify the difference between data, peer and host.

The data (or data items) is what is ultimately stored and exchanged in the system. A peer can be responsible for several data items (although usually not solely responsible, P2P system having some built-in redundancy): they store the items, and they provide copies of the data items when requested. Peers can also initiate queries for other data items, and will participate in the routing of such queries toward the peer that has a copy of the requested data item.

Finally, it may be possible to group several peers onto a single host (a computer). Depending on the system, the peers may know that they are sharing a host with other peers. This location information can be relevant for systems that attempt to provide efficient routing in terms of distance traveled by the routing request.

The goal of the P2P system is thus, on the one hand, to successfully and efficiently route a request for a data item to a peer holding a copy of that data, and on the other hand, to maintain the structure as peers join and leave the network at rapid pace.

2.2. *Hypercube*

The first type of architecture that can be found in second generation P2P systems is the *hypercube*. In this model, peers can be seen as nodes of some incomplete hypercube (that is, some of the nodes of the hypercube may have no peers associated with them). Depending on the P2P system, the dimension of the hypercube can be fixed or may adapt to the current number of peers in the system.

Hypercube-based systems can be efficiently searched. Indeed, in a hypercube of n nodes, the distance between two nodes is at most $log_2(n)$. So, provided that the routing is efficiently performed, this architecture allows to route a message toward its goal in at most $log_2(n)$ hops.

Efficient routing in an hypercube can be performed by assigning identifiers of size $log_2(n)$ to nodes corresponding to their location in the hypercube, from *00…0* for the bottom of the hypercube to *11…1* for the top. Efficient routing is typically achieved by routing a request for an ID toward a neighboring node with an ID having more common digits with the target node than the current node. It is easy to see that on a complete hypercube, at least one such neighbor always exists, and the entire routing from any node to any other node is bounded by $log_2(n)$ hops.

In practice, current systems base their routing on the work of Plaxton, Rajaraman and Richa (Plaxton *et al.*, 1999), and the routing is *prefix based*, where at each hop the node attempts to expend the common prefix between the ID of the current node and the target ID. In addition, most system do not necessarily use a base *2* but a base 2^b for some *b* (typically, $b=3$ or $b=4$), with 2^b nodes in each direction of the hypercube. In this case, search can be done in no more than $\mathrm{log}_2\mathrm{b}(\mathrm{n})$ hops.

In terms of information storage, maintaining the hypercube information requires also $log_2(n)$ links per node ($b.log_2(n)$ links for a base 2^b), which is in practice a good trade off between efficient routing and amount of information storage.

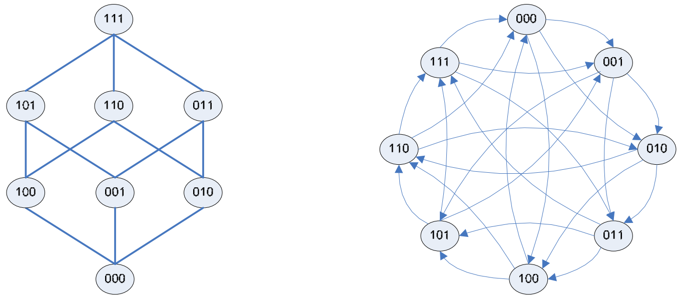

Figure 1: Hyper cube architecture (left) and skip list architecture (right) of the same 3 bits ID spaces of 8 elements

2.3. *Skip Lists*

The other type of architecture which is commonly found in P2P system is based on a simple *linked list*, were each peer keeps a pointer towards the next peer on the list. Much like with the classical data structure, having a simple linked list allows for simple implementation but is inefficient in terms of access, since the entire list might have to be traversed in order to locate an element. One solution to this problem is to enhance the list with additional links that can be used to speed up the searches. *Skip lists* (Pugh 1989) are one example of enhanced linked lists, in which “every $(2^i)^{th}$ node has a pointer 2^i nodes ahead” for any i within the linked list (Figure 1, right). Such a skip list allows searching any element in a list of length *n* in $O(log_2(n))$.

In order to adapt this scheme to P2P systems, the first modification required is to work for a circular list, in order to avoid having a “head” and a “tail”. This can be achieved fairly easily, and all jumps are done modulo the size of the list. There are however two more significant problems in a skip list. First, the load is not evenly shared among the nodes: half of the node maintain a single pointer to their direct successor, a quarter of the nodes maintain two pointers, one toward their successor, and one 2 nodes ahead, one eights of the nodes maintain three pointers (one 1 node ahead, one 2 nodes ahead and one 4 nodes ahead) etc. The second problem is that maintaining the structure of a perfect skip list is costly when nodes are added and removed.

The solution to these problems is to adapt the concept of skip lists so that each node maintains the same type of information, that is, a set of *h* pointers skipping ahead 2^i nodes, for $0 \leq i < h$ (modulo the size of the network). This still allows for a $O(log_2(n))$ search, while the amount of routing information maintained on each node (peer) is also $O(log_2(n))$.

In practice, P2P system based on linked lists work with a fixed circular ID space (e.g. 160 bits). With singly-linked lists, these P2P networks will operate in one direction around the ID space, always forwarding the message clockwise (or always counterclockwise) around the circular ID space.

Beside the good performance for peer look-up, insertion and deletion, this type of architecture provides two major advantages:

- The only information that is really necessary to ensure accurate routing is the linked list structure, that is, that each peer correctly points at its direct successor in the list. As long as this information is maintained, messages will be successfully (although possibly inefficiently) routed towards their destination peers. The other information in the routing table are "long jumps" that are necessary for efficient routing, but not needed for correct routing. As a consequence, it is sufficient to maintain the "next peer" information at all times and only ensure that the other information is reasonably accurate using a background process.
- Since peers are inserted in the list according to the ID, it is possible to efficiently search ID ranges in skip lists. It is sufficient to find the beginning of the range, and then move to the next peers until reaching the end of the range. However, this characteristic might be relevant only if the peers' IDs have some significance. If ID are randomly assigned to peers, the search of ID ranges is not useful.

In contrast, architectures based on simple lists do lack the capacity to handle geographic proximity. The rooting algorithm will, at each hop, forward the request toward a peer whose ID is closer to the search key, regardless of the geographic location of that peer.

2.4. *(Multi Dimensional) Cartesian Coordinate Space*

Another architecture for P2P systems uses multidimensional Cartesian coordinates. In this scheme, each peer is located somewhere in the space, and is responsible for the data items located within a geographical region (a *zone*) to which the peer belongs. It is aware of the boundaries of its zone, and knows about the peers in the neighboring zones as well.

Routing in such an architecture is done greedily, each peer forwarding the message in the direction of the destination, to a neighbor peer whose zone is closest to the goal, until a peer is reached that is in charge of the zone containing the destination. When a peer joins the network, a location *l* is attributed to it (for example randomly). The new peer then contacts the peer currently in charge of the zone containing *l*, and the zone is split among the two peers. When a peer fails or leaves the network, its zone can be merged with the zone of another neighbor peer.

The one particular strength about such an architecture is that the amount of information stored by each peer does not grow with the size of the network, but is bounded by $O(d)$, where d is the dimension of the coordinate space. On the other hand, the path lengths for routing requests is bounded by $O(n^{1/d})$ for a n peers network, which is larger than the logarithmic scale of the other architectures.

3. Data Management

In most P2P systems, data items are indexed in either of two ways: with a name that can be controlled by the data publisher, or by a mechanism under which the publisher has no control. In the first case, the name under which the data item is accessed in the P2P network can (and usually will) have some semantic meaning. Typically, it can be the actual file name in a file sharing service, or contain the name of the organization publishing the data item. In the other case, the name is usually obtained as a hash value of the “semantic” name, and thus is deterministically generated from that name, and cannot be controlled.

Having a hashed index is sometimes a necessary condition for the efficiency of the P2P network. This approach is used to ensure that the name space will be randomly and uniformly filled. On the other hand, it can be argued that such a hashed-index approach breaks some of the natural relationship between data items and the peers holding them, and among data items themselves. Indeed, with a hashed index, data items end up usually being handled by some random peer that just happens to have drawn the closest ID to the data ID so far in the system. And the data items that are “close” (e.g. direct neighbor) have no particular relationships, except a random one. For these reasons, semantics-preserving indexing is sometimes favored. However, the names are no longer uniformly distributed in the name space in this case, thus the underlying architecture must be performing well without this characteristic. Preserving the semantics of the names can sometimes allow for data locality, when the publishing peer decides what peers are going to be responsible for the data. More often, it is done to allow range-queries where routing is not based on a single ID value, but but on a range of ID values. This can be used to either reach (or search) a set of contiguous items intentionally, or to look for a possible match to a partially specified query. Note, however, that this can usually only be done for ranges sharing the same prefix, and only with one possible ordering semantic (the one selected to order the items).

In addition to what is stated above, the name used in the P2P system can be the actual data name, in which case the data item is retrieved directly from the peer to which the request was routed, or it can be meta-data, a pointer to the actual location of the name. In this case, the result of the query is the location of the real data, which can be the address of a peer or something totally outside the P2P network (URL etc.). The indirection of meta-data allows to have several names for the same data item, while having a single copy of that data.

4. P2P System Examples

4.1. *Freenet*

The Freenet system is a P2P system that primarily focuses on privacy and survivability (Clarke *et al.*, 2002). In this system, each peer provides storage space that can be used by other peers for data storage. Freenet is different from the other P2P systems surveyed here in that it does not have a well defined architecture; instead, peers are "trained" and learn to better route queries throughout their lifetime.

In this system, each file is associated with a global unique identifier (GUID, calculated by the SHA-1 hash function from the file content). When a peer joins the network, it sends an announcement message to an existing peer located through some out-of-band method. This peer, in turn, announces the new node to another node randomly chosen within the list of peers it knows. This announcement is dispatched inside the network to randomly chosen peers for a certain number of hops, as set by an application level parameter. At that point, a GUID is selected for the new peer, and this GUID is added to the routing tables of all the nodes that have participated in the peer addition process.

Each peer maintains a routing table of the peers it knows, along with the GUID of the data it "thinks" they have. When a new request for data (request for a GUID) reaches the peer, the peer forwards the request towards the peer that handles GUIDs that are numerically closest to the one requested. If this attempt fails, the peer tries the next best candidate peer in its list, and so on until either the data is found or the search is abandoned. One crucial point is that in case of successful location of the GUID, all the peers on the path between the requester and the peers holding the GUID update their routing table with the newly acquired knowledge of the address of the peer holding the requested GUID. This is the route training principle, which will ensure that subsequent requests for that GUID will be more efficiently routed and that requests for similar GUID will also be routed toward this peer.

When some new data is inserted in the network, the GUID insertion algorithm follows the same steps, thus ensuring that the peer that will end up holding a copy of the data is the peer that would have been contacted if the GUID had been looked up. This ensure a "specialization" of the peers that tend to be responsible for a set of closely related GUIDs, and known to be a reliable source for these GUID.

4.2. *Pastry*

Pastry is a hypercube-based decentralized P2P network (Rowstron *et al.*, 2001). In this system, the peers and the data are assigned a uniformly and randomly chosen 128 bits GUID. This GUID is represented as a sequence of digits in base 2^b, where b is a configurable parameter whose value is typically 4.

A peer routing table is made of two parts. The first part is a table of $128/2^b$ rows and 2^b columns. The entries of the n^{th} row contain the IP addresses of peers whose GUID has the same initial n digits as the GUID of the current peer. They are ordered by their $n+1$ GUID digit, with the entry at column m holding the IP address of a peer whose GUID's $n+1$'s digit is m. If no such peer is known, then the entry is left empty in the routing table. Since the 2^{128} namespace is sparsely filled and the distribution of peers is uniformly random, a peer's routing table will be typically completely filled in the first few rows, and will have fewer and fewer entries for the rows corresponding to longer prefixes.

In addition, the routing table also contains a *leaf set,* which is a set of l nodes that are numerically closest to the peer's GUID ($l/2$ above and $l/2$ below, for some configurable value of l). The initial definition of Pastry also included a neighborhood set, but this has been removed in the current version of the system (Castro *et al.*, 2002).

In order to route a request, a peer tries to forward the request to another peer whose GUID common prefix with the search key is one digit (b bits) longer than its own common prefix. To achieve this, it simply checks its routing table to see if the corresponding entry is populated, and if so simply forwards the request. If no such entry exists in the routing table, then the message is forwarded to another peer that has the same common prefix with the key, but that is numerically closer. Such a peer must be found, if only in the leaf set (this shows that the routing will always succeed unless the $l/2$ peers in the leaf set fail simultaneously).

When a peer joins the network, it randomly chooses a GUID, and contacts an existing peer through some out-of-band method. From that initial entry point, it locates a "nearby" node by using the initial node's leaf set (whose peers should be geographically randomly and uniformly distributed) and trying to find closer and closer peers in these peers' routing tables (see (Castro *et al.*, 2002) for details). Once a nearby node is located, the new peer sends a join message with its GUID. The join algorithm follows the same steps as the routing algorithm, except that at each step m, the m^{th} row of the current peer's routing table is copied into the m^{th} row of the routing table of the new peer. At the end of the process, the node will have built an entire routing table with the required properties. Finally, the last step of the process consists of informing the existing nodes about the new peer, so that their routing table can be updated if required.

One important aspect of this scheme is the selection of a nearby node and the maintenance of proximity information. Thanks to the way the routing table is constructed, it tends to be filled up with peers that are geographically close (among the possible peers for a particular table entry). As a consequence, the routing of a message tends to stay close to the initial node for most of the hops, and the final overall distance traveled will be on average dominated by the last routing step. "*As a result, the average total distance traveled by a message exceeds the distance between source and destination node only by a small constant value*" (Castro *et al.*, 2002) .

4.3. *Tapestry*

Tapestry is another P2P system with a hypercube architecture (Zhao *et al.*, 2004). Like Pastry, but unlike other systems such as Chord or Can, Tapestry takes the traveled distance into account when routing a message, and does not merely minimize the number of hops.

The routing strategy of Tapestry is similar to that of Pastry, that is, the goal is to always increase the common prefix between the current node ID and the key. Peer IDs and data IDs do share the same name space, a 160 bits GUID usually coded in hexadecimal, leading to a 40 digits hexadecimal ID. The routing table is similar to the one of Pastry, with the m^{th} entry of the n^{th} row having the address of the closest peer with a GUID sharing a prefix of size n with the current node and whose $n+1$'s GUID digit is m. It does not, however, have a leaf set, and uses instead a slightly different routing strategy, called *surrogate routing*. When routing a message, if a perfect match cannot be found for the next level of prefix, a Tapestry peer then looks for a "close" digit in its routing table instead. This surrogate routing strategy ensures that the message is routed toward a peer with a GUID which is "close enough" to the key. This peer is called the *identifier root* and is unique (Hildrum *et al.*, 2002).

One particularity of Tapestry is the relationship between data and peers. In Tapestry, the peers decide to "publish" data items for which they have a copy. The GUID of the data item is generated automatically and has no direct relationship with the GUID of the peer owning the data. In order to advertise the data, the peer sends a *publish* message into the network, announcing the data GUID and its own GUID. The publish message is routed in the network, using surrogate routing, toward its identifier root (the GUID of the data). The peers along the path from the publishing peer to the identifier root store the *location mapping* of the data GUID and its publishing peer. Note that several peers can thus publish the same data (same GUID), and the various publish messages will follow different path but will all eventually converge, if only at the identifier root.

When a peer wants to locate data, it again uses surrogate routing for the data GUID, which will eventually reach the identifier root of the data. Along the way, if the data is currently being published, the locate message will go through a peer that was involved in the publish message of the data and that has a record of the publishing peer address in its location mapping table (again, at worst, this peer is the identifier root). If more than one peer is publishing the data, then the locate message will cross the path followed by the publish message of the closest peer first, thus the requesting peer will be directed towards this closer peer.

4.4. *Chord*

Chord (Stoica *et al.*, 2001) is a P2P system with an architecture based on the skip list idea described in Section 2.3. This is a system that is simple to understand and which provides one basic operation: “given a key, it maps the key onto a node” (Stoica *et al.*, 2001). It operates with a circular node ID space (typically, but not necessarily, a 160 bits SHA-1 hash) and ensures that a peer can be located with $O(log_2(n))$ messages (where n is the number of nodes), while each peer maintains information about $O(log_2(n))$ other peers. In addition, with high probability, this information can be maintained as peers join or leave the network with $O(log^2(n))$ messages for each update.

Like other P2P systems based on this architecture, Chord only requires that one piece of information per peer (namely the address of the next peer in the ID space) is correct in order to guarantee correct routing. On the other hand, it is unaware of network topology and the message can travel a total distance far greater than the actual distance between the source and the destination peer. In addition, since Chord’s peer IDs are randomly and uniformly assigned, the network does not get any significant benefit from the architecture’s ability to do range queries on node IDs. Chords’ advantage remains its simplicity.

In Chord, each peer maintains a routing table with at most *m* entries if the node IDs have *m* bits. This routing table is called a *finger* table, and the i^{th} entry in the table contains the address of the first peer with an ID at least 2^{i-1} away from the current peer’s ID (modulo 2^m). Clearly, a peer knows more about peers that are close to it in the ID space than about peers that are far away. It is also clear that in general, a peer cannot directly route a query to the destination peer because it does not have enough information in its finger table. The actual routing algorithm consist of finding in the current peer’s finger table the address of the peer that is the closest to the key being searched, without getting past that key value, and sending the request to that peer. The message is sent from peer to peer until it reaches the destination peer.

A Chord peer joins the network by first contacting a current node (through some out-of-band method). It can then construct its finger table by using this node to find the successor of its own node ID (similar to routing a request) and the other (NodeID+2^i modulo 2^m) entries. In order to inform efficiently the other peers about the arrival of the new node, the system will have to traverse the network *backwards* to the immediate predecessor of a node. This link is also maintained by Chord.

4.5. *SkipNet*

Like Chord, SkipNet is a P2P system based on a skip-list architecture (Harvey *et al.*, 2003). However, unlike Chord, SkipNet makes use of the range query capacity of skip lists to provide *content locality* and *path locality*: a peer is able to keep the control of the data it publishes and if an entity such as a commercial organization has several peers in the system, it can ensure that messages exchanged between its peers will be routed through its own peers only

SkipNet achieves its goal by having two separate (but related) node ID spaces: the string *name ID* which records the actual name and identifiers, and the *numeric ID* which is the hash value of the actual name and identifiers. As expected, the numeric IDs are randomly and uniformly distributed, and name IDs are not. Yet, SkipNet can efficiently route searches by both name ID and numeric ID (that is, in $O(log_2(n))$ messages either ways, for a network of *n* peers).

In order to achieve this, SkipNet works with "rings" of different level. The level 0 ring includes all the peers, the two level 1 rings include each about ½ of the peers, and the 2^i level *i* rings include about $1/2^i$ peers each. There are up to *m* levels of the numeric ID, coded on *m* bits, accommodating up to 2^m peers.

The level 0 ring is the usual circular linked list, where the peers are sorted according to their name IDs. In other words, in this ring, each peer knows of the peer with a name that comes directly after its own name.

Peers use their numeric ID to decide which higher level ring to join. The first *i* bits of the numeric ID give 2^i possible values, and there are 2^i level *i* rings. Thus, it is enough to look at the first *i* bits of the numeric ID and use it to assign the peer to a ring of level *i* in a random and uniform way. This implies that peers belonging to the same ring at level *k* have the same *k* bit prefix in their numeric IDs.

At the first level ring, neighbor peers have contiguous names in the name ID space, but are far apart in the numeric ID space. This situation reverses as we climb the levels, and at the last level ring, neighbor peers have contiguous numeric ID in the numeric ID space, but are far apart in the name ID space. Note that with SkipNet, those lists are doubly-linked, so it is possible to move both ways on each ring (but peers have to store twice as much routing information).

In order to route a message by name ID, SkipNet uses the usual skip list routing algorithm, that is, it looks for the next peer that is closest to the destination, without ever going beyond the destination, and using as high a ring as possible in order to jump as "far" as possible at once in the name ID space.

Routing by numeric ID is slightly different. The idea is to expand the matched prefix at each step. So initially, on ring level one, routing is done from neighbor to neighbor until a peer is found that shares the same first numeric ID bit. The message search is then moved up from the ring at level 1 to which the peer belongs, and a search for a peer sharing the first 2 bits is initiated, and so on, until the final level ring is searched. This search can always be done in $O(log_2(n)$ steps.

Since the primary "sorting key" is the name ID, it is possible for the peer to use their organization's name as prefix of their name, and to prefix by the same name the data they want to publish, to guaranty that:

- Peers belonging to the same organization (starting with the same name) will be contiguous in the level 0 ring.
- Data published with an identifier matching the organization's name will be store on a peer of that organization (content locality).
- Requests from within the organization will never be routed outside the organization since the requests are never sent past their destination.

These characteristics are interesting from a security viewpoint, since it allows the organization to keep control of its own data and ensure that traffic exchanged within the organization will be routed though the organization's peers only. It has also a nice consequence in terms of fault tolerance: if one assumes that failures typically happen at the organization boundaries (faulty router/proxy, misconfiguration of gateway etc.), the typical consequence of a failure is that the entire organization is (temporarily) disconnected from the internet. Since all the peers in the same organization are stored contiguously in SkipNet, it means that such a failure disconnects a set of contiguous nodes at once, and that the connectivity for intra-organization traffic is preserved.

4.6. *GosSkip*

GosSkip is another P2P system based on skip lists (Guerraoui *et al.*, 2006). In this system, the locality of the content semantics is preserved, in other words, peers are totally ordered according to a relationship that is external to the system (for example, alphabetical ordering of the names used for peers and content). This allows for efficient range queries across peers or data.

GosSkip is a system that directly builds the first ring of the overlay in the usual way, with peers inserting themselves at the correct location in the level 0 ring by routing a query towards their name in the exiting network. The higher level rings, which are only needed for efficiency, are built using a *gossip* protocol which periodically transmits information between neighbors and can be piggy-backed onto existing application messages. This approach ensures that the overlay is self-organizing and continually rebuilt from the existing level 0 ring.

The gossip protocol works as follows: regularly, each peer sends information to their immediate neighbor on a ring, say the level *i* ring. This information contains a list of pairs (peer ID, index). This list contains the peer's own identity with the index value 0, and the identity of all the other peers collected since the last gossip message at that level, with an index value increased by one compared with the value received. When receiving such a message, the peer can learn two things:

- The peer associated with the index value 1 in that message is the current immediate predecessor of the current peer for the ring at level *i*. If needed, the peer can create or correct its current information for this ring.
- The peer associated with the index value *k* in that message, where *k* is the number of contiguous peers along the ring at level *i* that are skipped in one long jump on the ring at level *i+1* (typically, *k=2*) is the current immediate predecessor of the current peer for the ring at level *i+1*. If needed, the peer can create or correct its current information for this ring. The information about the peer with index value *k* is then discarded from the gossip message, which thus never carries more than *k* pieces of information.

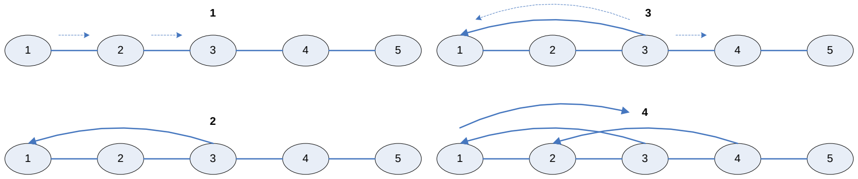

Figure 2: GosSkip gossip messages

As an example, consider Figure 2: at Step 1, a gossip message is sent from Peer 1 to Peer 2 with the information about Peer 1 at index value 1. The message is then gossiped from Peer 2 to Peer 3 with the information about Peer 2 at index 1 and Peer 1 at index 2. When this information is received by Peer 3 (Step 2), it learns that its predecessor on Ring 1 is Peer 1, and creates the link. Two messages can now be gossiped (Step 3), one along Ring 0 from Peer 3 to Peer 4 with the information about Peer 3 at index 1, Peer 2 at index 2 (information about Peer 1 has been discarded), and one along Ring 1 from Peer 3 to Peer 1 with the information about Peer 3 at index 1. At Step 4, Peer 4 learns about Peer 2 as its immediate predecessor along Ring 1, while Peer 1 learns about Peer 3 as its immediate successor along Ring 1. The spreading of information continues, towards the right on Ring 1, towards the left on Ring 2.

GosSkip is also equipped with a "spreading" algorithm which allows efficient broadcasting of range queries (or single queries reaching a range of equal-valued peers). This algorithm is invoked once the normal routing algorithm has reached one of the target peers. In order to efficiently distribute the request throughout the range of peers that should receive it, the spreading algorithm uses the higher level rings to determine how far away the request should be dispatched. The message is then sent at once to all neighbor peers for the concerned rings. Each of these peers receives the message together with instructions about how far to spread the message.

4.7. *CAN*

The Content-Addressable Network (CAN, Ratnasamy *et al.*, 2001) is a distributed hash-table mechanism based on multidimensional Cartesian coordinates. CAN works with a *d*-dimensional space, each peer being responsible for one zone in this space.

Each peer is assigned a random *d*-dimensional location and maintains the list of its current neighbor in the network. When routing a request, the peer greedily forwards the message towards the neighbor that is closest to the searched key. A new peer sends a routing message toward the location it has been randomly assigned, and thus contacts the peer in charge of the zone containing that location. The peer then splits its zone in half and hands over half of it to the new peer. The local information of the peer is enough to build the neighboring information for the new peer. A message is then sent one hop away to the neighboring peer to alert them of the new configuration. This very localized protocol is sufficient to update the network. When a peer leaves (or is identified as failing by its neighboring peers), the neighboring peer in charge of the smallest zone takes over the zone of the departed peer.

As already mentioned, this mechanism allows each peer to store a fixed amount of information, regardless of the size of the network, but the number of hops of routing messages grows as $O(n^{1/d})$ for a network with *n* peers. Also, CAN does not ensure short traveling distances and, since the data is stored according to its hash value, significant range queries are not efficient.

5. A Discussion of eQuus

5.1. *Overview of eQuus*.

eQuus is a recently introduced P2P system that follows the hypercube architecture (Locher *et al.*, 2006). In eQuus, a new concept called *clique* is introduced. Each peer within a clique is responsible for the same set of data items. This simple idea provides robustness, since data remains available as long as at least one peer in the corresponding clique is up and running.

In eQuus, new peers are added to cliques based on geographic proximity: when a peer joins the network, it is automatically added to the geographically closest clique. This locality is the reason for the second advantage of eQuus, namely, that data sharing within a clique is fast thanks to the geographical closeness of the peers in a given clique. The relationship between cliques mimics the peer architecture of the already surveyed P2P system Pastry. However, in eQuus, a relationship between two cliques is maintained by having each peer of the first clique maintain *k* links to randomly selected peers in the second clique. Within a clique, peers share the same ID (the *clique ID*), and are responsible for the data items having an ID between their clique ID and the ID of the next clique (data IDs are randomly uniformly distributed).

When a clique has reached a given size (set by a system-wide parameter), the clique will split in two, half of the peers joining a new clique with an ID numerically half way between the ID of the split clique and the ID of its immediate successor. The responsibility for data items will be shared (on average equally) between the two cliques. For example, in a new network all the peers are initially part of the same clique (with clique ID 00…0) and the entire set of data items are the responsibility of this clique. Once enough peers are in the system, a new clique with ID 10…0 is created, and half of the initial clique's peers keep the responsibility for data items 00…0 to 01…1, while the second half has responsibility for data items 10…0 to 11…1. During such a split, the peers' proximity within each clique is preserved.

Finally, when a clique's size reduces to a certain threshold (set as a system-wide parameter), this clique will merge with the preceding clique in the clique ID space. This way, eQuus guaranties that the level of redundancy does not go below a given safety level.

5.2. Limitations of eQuus.

One criticism that can be made about eQuus is that clique management is entirely done based on the clique size. It ignores other factors such as the clique's load. We have identified the following two scenarios that could be problematic for this system.

First, the unbalanced clique allocation strategy may create a seriously unbalanced workload. For example, assume that an eQuus P2P network is built out of many European peers and a few Canadian ones. If the Canadian peers and some European peers first join the system such that two cliques are set up, we will end up with one clique having the Europeans peers that are responsible for half of the data, while the Canadian peers are put into the second clique, responsible for the other half of the data. From that point on, if more and more peers are added on the European side of the network, the number of cliques will grow and the data items located on the "European side" of the ID space will be shared between an ever increasing number of cliques, while the other half of the data will always remain the responsibility of the a single Canadian clique and its peers.

Second, if for some reason a particular data item is particularly "popular" and often requested, the clique in charge of it will bare the cost of this load, regardless of the level of activity of the other cliques.

5.3. *Hints at possible solutions*

We believe that eQuus can be improved to better handle the situations described above. Details are beyond the scope of this paper, but we claim that split and merge activities should not be based only on the size of cliques. For example, in the case of unbalanced data ID ranges, the clique in charge of the larger range could move forward in the ID space, so as to shift the responsibility of some of its data items to its predecessor clique[1].

When a clique of eQuus is in charge of a larger amount of data (or in charge of a very popular data item), it could momentarily "hire" a set of peers from the successor and predecessor cliques (by letting some their peers join the clique) in order to help coping with the current load. In this case, additional mechanisms to monitor the cliques' load and to decide when a clique can spare a peer will have to be added to eQuus.

6. Conclusion

In this paper, we have provided a survey of several different models for content distribution in current peer-to-peer (P2P) systems. We have in particular identified the hypercube architecture and the skip list architecture, and have identified some of the strengths and weaknesses of each of these approaches. We have also provided a survey of the architecture and performance of numerous P2P systems. Finally, we have provided an overview of a recently proposed clique-based system, eQuus, for which we have identified apparent weaknesses and proposed some possible solutions.

Bibliography

Pourebrahimi B, Bertels K.L.M, Vassiliadis S., "A Survey of Peer-to-Peer Networks", *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, ProRisc 2005, November 2005.

Plaxton C. G., Rajaraman R., Richa A. W., "Accessing nearby copies of replicated objects in a distributed environment", *Theory of Computing Systems*, 32:241-280, 1999.

Clarke I., Miller S. G., Hong T. W., Sandberg O., Wiley B, "Protecting free expression online with Freenet", *IEEE Internet Computing*, 6(1):40 – 49, Jan./Feb. 2002.

Rowstron A., Druschel P., "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems". *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, pages 329-350, November, 2001.

[1] This could be done without much impact on the current eQuus by simply doing a "fake" split, then a merge back, effectively halving (in average) the amount of data items handled by the clique.

Castro M., Druschel P., Hu Y. C., Rowstron A., "Topology aware routing in structured peer-to-peer overlay networks," *Tech. Rep. MSR-TR-2002-82*, Microsoft Research, One Microsoft Way, Redmond, WA 98052, 2002.

Zhao B. Y., Huang L., Stribling J., Rhea S. C., Joseph A. D., Kubiatowicz J. D., “Tapestry: A Resilient Global-scale Overlay for Service Deployment”, *IEEE Journal on Selected Areas in Communications*, Vol 22, No. 1, January 2004.

Hildrum, K., Kubiatowicz, J. D., Rao, S., Zhao, B. Y., “Distributed object location in a dynamic network”, *Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Manitoba, Canada, August 10 – 13, 2002.

Pugh, W, "Skip Lists: A Probabilistic Alternative to Balanced Trees", *Workshop on Algorithms and Data Structures*, 1989

Stoica, I, Morris, R, Karger, D, Kaashoek, F, Balakrishnan, H, “Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications”, *ACM SIGCOMM*, San Diego, California, USA, August 27 – 31, 2001.

Harvey, N. J. A., Jones, M. B., Saroiu, S., Theimer, M., Wolman, A., “Skipnet: A scalable overlay network with practical locality properties”, *USITS, Fourth USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, March 2003.

Guerraoui, R, Handurukande,S, Huguenin, K, Kermarrec, A-M, Le Fessant, F, Riviere, E,, "GosSkip, an Efficient, Fault-Tolerant and Self Organizing Overlay Using Gossip-based Construction and Skip-Lists Principles", *Sixth IEEE International Conference on Peer-to-Peer Computing*, 2006

Ratnasamy, S, Francis, P, Handley, M, Karp, R, Shenker, S, “A scalable content-addressable network”, , *ACM SIGCOMM*, San Diego, California, USA, August 27 – 31, 2001.

Locher, T, Schmid, S, Wattenhofer, R, “eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System” , *Sixth IEEE International Conference on Peer-to-Peer Computing*, 2006.