European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies
A Network of Excellence funded by the European Commission

Investigating peer-to-peer meta-brokering in Grids

A. Kertész, P. Kacsuk
{attila.kertesz, kacsuk}@sztaki.hu
MTA SZTAKI Computer and Automation Research Institute
H-1518 Budapest, P.O. Box 63, Hungary

A. Iosup, D. H.J. Epema
A.Iosup@tudelft.nl, D.H.J.Epema@ewi.tudelft.nl
Delft University of Technology (TUD)
Mekelweg 4, 2628 CD, Delft, The Netherlands

CoreGRID Technical Report
Number TR-0170
August 31, 2008

Institute on Resource Management and Scheduling

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

Investigating peer-to-peer meta-brokering in Grids

A. Kertész, P. Kacsuk
{attila.kertesz, kacsuk}@sztaki.hu
MTA SZTAKI Computer and Automation Research Institute
H-1518 Budapest, P.O. Box 63, Hungary

A. Iosup, D. H.J. Epema
A.Iosup@tudelft.nl, D.H.J.Epema@ewi.tudelft.nl
Delft University of Technology (TUD)
Mekelweg 4, 2628 CD, Delft, The Netherlands

CoreGRID TR-0170
August 31, 2008

Abstract

Grid Computing has succeeded in establishing production Grids serving various user communities all around the world. The emerging Web technologies have already affected Grid development; the latest solutions from peer-to-peer networking also need to be considered in order to successfully transform the currently separated production Grids to a World-Wide Grid solving the big problem of Grid Interoperability. This paper gathers the promising peer-to-peer technologies and investigates their applicability in current Grids. We state the requirements of a scalable and adaptable high-level brokering architecture, and define the necessary routing mechanisms that can efficiently operate this peer-to-peer Grid meta-brokering architecture.

1 Introduction

Grid Computing has succeeded in establishing production Grids serving various user communities all around the world. Several world-wide Grid projects have delivered the expected developments in the area of high-performance computing, but we cannot sit back and watch idle, how users tweak scripts to get complex, computation-intensive applications run on various Grids still not mature enough for business purposes. The four year of intensive Grid research in CoreGRID (a leading European Network of Excellence project [1]) has created a sustainable European Grid Research Laboratory with 6 institutes that target different Grid research fields. They specify the Grid as a fully distributed, dynamically reconfigurable, scalable and autonomous infrastructure to provide location independent, pervasive, reliable, secure and efficient access to a coordinated set of services encapsulating and virtualizing resources (computing power, storage, instruments, data, etc.) in order to generate knowledge. Though most of the above stated adjectives hold for current Grids, Grid development is shifting and needs to focus more on cooperation with other successful and popular related fields. Web and peer-to-peer (P2P) computing have attracted great attention recently, and research in these areas has been fostered. NGG [2] has already envisaged the convergence of these research fields, therefore it is important to merge ideas, join research efforts in order to build the world of SOKUs, self-aware, adaptive and scalable Grid services. Life holds two certainties: death and taxes - this was the metaphor I. Foster et. al used in [4] to illustrate that Grids and peer-to-peer technologies are both needed to organize large-scale computational societies. Scalability and fault tolerance are two major issues Grids still cannot fully cope with. On the contrary P2P systems operate on millions of nodes and still show good resilience on continuous node changes and failures. To solve

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

the problem of Grid Interoperability, it is a must to investigate the application of yet proven successful peer-to-peer solutions. In this paper we gather promising peer-to-peer technologies for solving Grid interoperability problems and investigate the applicability of them. We target the resource management layer of the Grid middleware in our research, therefore we are looking for a high-level, interoperable brokering solution. The next section mentions the related approaches, Section 3. states the requirements of the new architecture, Section 4. gathers the alternative P2P overlays, finally Section 5. describes the mechanisms required to operate the proposed architecture.

2 Related work

In the introduction we have seen that Grids and peer-to-peer technologies need each other. I. Foster et. al. have further investigated in [5], how P2P approaches could be used in resource location. They proposed an unstructured overlay for resource discovery, and evaluated random-forwarding and learning-based propagation strategies with an own Grid emulator. They concluded that more sophisticated, learning-based search techniques are needed for Grids consisting of tens of thousands of institutes.

Similarly to this previous approach D. Talia et al. [3] have also worked on P2P resource discovery for Grids. They have written more technical reports in this field, their main contribution is a hybrid architecture of unstructured overlay with superpeers in a Chord ring. They used the Grid5000 [6] testbed to evaluate their solution. The evaluation results show that their solution is suitable for todays Grid systems, and would be able to substitute the currently more centralized information systems.

Uppuruli et. al proposed a framework for distributed resources management in [7]. They use an XML-based description to specify resource information. They use an unstructured architecture implementation based on Phex Gnutella with ultrapeers. They target the connectivity layer instead of the management layer to create their peer-to-peer system. The presented experiments show that their proposed approach is feasible to use P2P discovery in Grids.

Most of the related approaches propose a unique solution for resource discovery in Grids, which is the role of Grid information systems. Their novelty is that they use distributed, peer-to-peer architectures instead of the conservative centralized ones. We decided to take a step forward and extend resource discovery with brokering. In the following sections we state the requirements of this novel approach and explore its design space in order to arrive in an efficient and suitable solution.

3 Problem statement: Peer-to-peer meta-brokering

The term meta-brokering has been first introduced in [8]. The main reason for creating another level in Grid resource management on top of the existing brokers was to enhance Grid interoperability. The Grid Meta-Broker is a Grid service that utilizes currently used resource brokers by providing a uniform interface to users. In this way users of different communities can reach several Grids through their own brokers, without knowing about lower level details of each Grid middleware (description language, resource availability and different submission interfaces). In this work we build on and extend this model in order to complete the resource management layer of a future interoperable WWG [12]. To achieve this we need to interconnect Meta-Brokers, share user requests among them to balance load and maintain the whole architecture. This interconnected topology can be studied within the framework described in our previous work [9, 10, 11]. The goal of this paper is to investigate different peer-to-peer solutions that are suitable to be used in our approach.

Figure 1. shows our extended model. Brokers (B) operate Grids (G) or Virtual Organizations (VO) of resources (R) with certain properties (P). These brokers can have different capabilities (C) that serve specific user requests, jobs (J). Meta-brokers (MB) utilize more brokers and provide interface for different user communities to access different Grids. Meta-brokers are interconnected peers in an overlay network. This interconnection also needs a slight modification of the meta-broker peers: they maintain a local queue for incoming jobs and implements a routing mechanism to be described later in Section 4. We further state the following assumptions:

- meta-brokers can be operated by various providers that do not necessary trust each other, therefore not all the peers are interconnected in the network,
- as a result of lack of trust there is restricted information exchange among the peers,

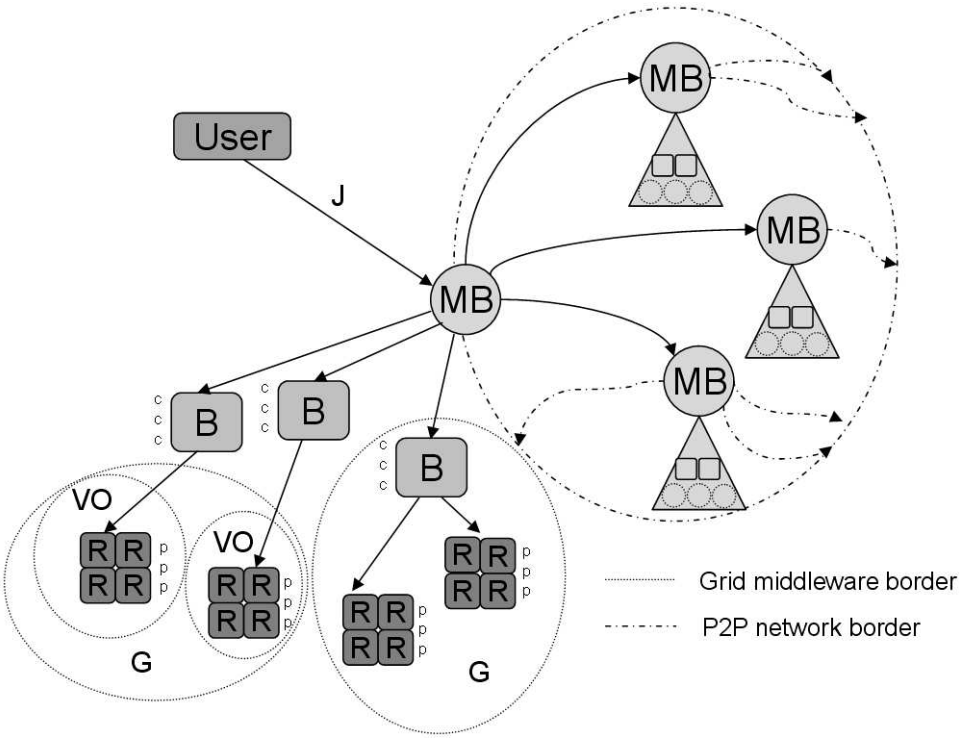

Figure 1: General P2P meta-brokering architecture

- the number of resources, brokers and meta-brokers may vary over time, but the system uses a fixed number of resource properties and broker capabilities; we refer to these as static attributes further on in the paper,
- the meta-brokers are also characterized by dynamic performances: examples of these are the number of job requests in the queue and the number of failed or successfully executed jobs of the utilized brokers.

The requirements of this new peer-to-peer architecture and the problem statement of this paper are the following:

- find a suitable and easily maintainable overlay topology,
- minimize or find an optimum number of links for the peers,
- limit the number of concurrent jobs at broker-level (with usage SLAs),
- and define a routing mechanism that efficiently operates the whole architecture by:
  minimizing the waiting time of the users,
  maximizing the throughput of the system,
  and minimizing the number of failed jobs.

4 Overview of existing peer-to-peer overlays

In this section we gather the related overlay topologies and provide a short summary of them. We focus on their structure, routing and maintenance methods. Before the survey, we state the general definitions. Peer-to-peer networks typically connect *peers* (also called as *nodes*) based on an overlay topology. Peers know each other in the network, if they are connected by *links*. *Neighbors* of a node are the ones that have a link to that node, so they know each other. Each node in the network has a *neighbor list* (*finger table* or *routing table*) to store information on its neighbors.

The basic classification of these networks is done according to their structure: they can be structured or unstructured. Structured overlays build on some geometry and usually have restrictions on the routing and node placement. *Routing* defines a mechanism, how messages are forwarded in the network. A *lookup* (*search* or *query*) is an operation that looks for a data in the network according to the routing mechanism. In general, the following maintenance operations can be done in these networks:

- *joining*: a new entity arrives and wants to be connected to the network. We suppose that these entities know how to contact at least one node in the network, called *bootstrap* node, which can initiate a process to connect the entity to the network in order to become a peer. This process directs the entity to some already existing node to establish a link with it, creates a routing table for this new node, and *updates* the neighbors routing tables with its own data.
- *leaving*: an existing node decides to disconnect from the network. Before disconnecting it needs to notify its neighbors to delete its data and if necessary add new neighbors to keep the overall connectivity. A special type of leaving is when a failure occurs on a node. In this case no notification is sent out to the neighbors: some overlay solutions can deal with this situation by regularly sending *keep-alive* messages to the neighbors.

4.1 Unstructured overlay topologies

In general, an unstructured network is formed when the overlay links are established arbitrarily. There can be an upper and lower bound defined for the number of entries in each nodes neighbor table. A joining node may use a *random walk* (hops through randomly selected links) starting from a bootstrap node, to find other nodes to fill its neighbor table. We classify routing mechanisms in unstructured overlays as *deterministic* or *probabilistic*. A deterministic routing is *flooding* (sending messages to all neighbors), or a *rank-based* routing that decides the next neighbor according to an algorithm (more detailed in Section 4). Probabilistic routings are random walks with a predefined *TTL* (time-to-live: the maximum number of hops a message can go through) number. Flooding has the disadvantage of too many message exchanges, while random walks cannot guarantee the success of the search and also have long response times. An enhanced probabilistic approach is the k-walker random walk, where the query is sent to k randomly selected neighbors. In this case the routing delay can be reduced.

Beside these general approaches we consider two other solutions for our problem: one is the BitTorrent [bittorrent] protocol. When a new peer *joins* a BitTorrent network, it asks the tracker (the only centralized component of the system that keeps track of the peers currently involved in the search) a list of peers to connect to and cooperate with (usually randomly chosen). This set of peers forms the *neighbor list* (peer set) of the new peer. This peer set will be augmented by peers connecting directly to this new peer. Such peers are aware of the new peer by a request to the tracker. Each peer reports its state to the tracker regularly, or when disconnecting from the network. When the number of peers in the peer set of the new peer falls below a predefined threshold, this peer will contact the tracker again to obtain a new list of peers. Moreover, a peer should not exceed a threshold number of initiated connections among the known peers at each time (this threshold is usually half of the known peers). As a consequence, the remaining connections should be initiated by remote peers. This policy guarantees a good interconnection among the peer sets and avoids the creation of cliques. Each peer knows which pieces each peer in its peer set has. The consistency of this information is guaranteed by the exchange of messages among the peers, which is governed by two core algorithms: the choke and the rarest first algorithms. We are interested in the choke algorithm to select the active peers from the neighbor table to forward messages to for a certain time period. In our case it could be used to penalize peers that never have good resources for user requests. By changing the default parameters of this algorithm it is possible to increase the size of the active peer set and the number of optimistic unchokes.

The second is the OurGrid [ourgrid] approach, which is also called as a network of favors. They assign priorities to peers based on their historical data: peers that contribute more to the community are prioritized when they request resources. This approach particularly useful for bag-of-tasks applications (their tasks are independent). Furthermore it would be interesting to see, how the usage of the OSPF internet routing protocol would perform in our architecture. It is specifically designed to operate with larger networks and does not impose a hop-count restriction. It maintains a routing table by exchanging link-state information with its neighbors.

4.2 Structured overlay topologies

Figure 2. shows simple lookup examples for each overlays. The detailed description of them follows below.

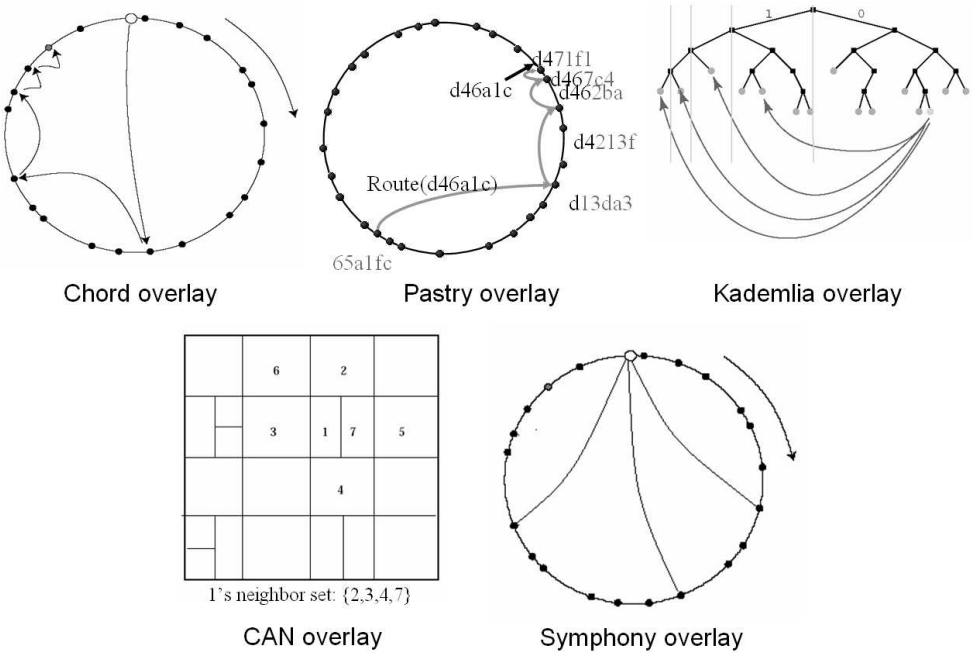

Figure 2: Alternatives of structured overlays

In *Chord* [15] overlay structure peers are organized in ring geometry. Each node has an *m*-bit identifier, which is usually chosen by hashing the nodes IP address. Certain keys represent data stored at different nodes. Each key also has a hashed identifier (ID). IDs are ordered in an identifier circle modulo 2m. Key *k* is assigned to the first node, whose ID is equal to or follows *k* in the ID space. This node is called the successor node of *k*. Each node stores information about only a small subset of the nodes in the system in its routing table. The *routing process* is the following: the search for a node moves progressively closer to identifying the successor with each step. A search for the successor of *t* initiated at node *b* begins by determining if *t* is between *b* and the immediate successor of *b*. If so, the search terminates and the successor of *b* is returned. Otherwise, *b* forwards the search request to the largest node in its finger table that precedes *t*; call this node *p*. The same procedure is repeated by *p* until the search terminates. When a node *n joins* the network, some keys that were previously assigned to *n*'s successor are now assigned to *n*. When node *n leaves* the network, all of its assigned keys are reassigned to *n*'s successor. It has efficient directory operations, but high maintenance costs. When a node *n fails*, nodes whose routing tables include *n* must find *n*s successor. The key step in failure recovery is maintaining correct successor pointers. Therefore to support fault tolerance, each node should maintain a successor-list of its *r* nearest successors on the ring. If node *n* notices that its successor has failed, it replaces it with the first live entry in its successor list. At that point, *n* can direct ordinary lookups for keys for which the failed node was the successor to the new successor. As time passes, this stabilization will correct routing table entries and successor-list entries pointing to the failed node.

Pastry [16] uses a tree geometry, but the peers are mapped to a ring. Each node has a randomly assigned unique *m*-bit identifier (typical value is 128). Each node maintains a routing table (*R*), a neighborhood set (*M*) and a leaf set (*L*). *L* consists of the *L*/2 closest peers by node ID in each direction around the circle. *M* represents the *l* closest peers in terms of the routing metric. Although it is not used directly in the routing algorithm, the neighborhood list is used for maintaining locality principals in the routing table. There is a configuration parameter *b* (with a typical value is 4), which determines the number of entries in *R*. The higher this value is, the more fault tolerant the network is. The routing algorithm assumes that both the node IDs and the keys are sequences of digits with base 2b. Pastry routes messages to the node, whose ID is numerically closest to the given key. So the *routing* goes as follows: the node first checks to see if the search key falls within the range of the IDs covered by its leaf set. If so, the message is forwarded directly to the destination node, namely the node in the leaf set whose ID is closest to the key. If the key is not covered by the leaf set *L*, then message is forwarded to a node in *R* that shares a common prefix with the key by at least one

more digit. In certain cases, it is possible that the appropriate entry in the routing table is empty or the associated node is not reachable, in which case the message is forwarded to a node that shares a prefix with the key at least as long as the local node, and is numerically closer to the key than the present nodes ID. When a new node n arrives to *join* the network, it needs to initialize its state tables, and then inform other nodes of its presence. We assume the new node knows initially about a nearby Pastry node b, according to the proximity metric that is already part of the system. Node n then asks b to route a special join message with the key equal to n. Like any message, Pastry routes the join message to the existing node t whose id is numerically closest to n. In response to receiving the join request, nodes n, t, and all nodes encountered on the path from n to t (the actual *route*) send their state tables to n. Finally n informs any nodes that need to be aware of its arrival. A Pastry node is considered *faile*d when its immediate neighbors in the ID space can no longer communicate with the node. To replace a failed node in the leaf set, its neighbor in the ID space contacts the live node with the largest index on the side of the failed node, and asks that node for its leaf table. This set partly overlaps the present nodes leaf set, and it contains nodes with nearby IDs not present in it. Among these new nodes, the appropriate one is then chosen to insert into the list, verifying that the node is actually alive by contacting it. To repair a failed routing table entry, a node contacts first node of the same row, and asks for that nodes entry for the missing one. In the event that none of the entries in the same row have a pointer to a live node with the appropriate prefix, the node next contacts an entry in the next row, thereby casting a wider net. Furthermore a node attempts to contact each member of the neighborhood set periodically to see if it is still alive. If a member is not responding, the node asks other members for their neighborhood tables, checks the distance of each of the newly discovered nodes, and updates it own neighborhood set accordingly.

Kademlia [17] uses a symmetrical tree overlay, and its routing is based on XOR metric. The nodes have random m-bit identifiers (typically 160). The Kademlia algorithm is based on the calculation of a distance, which is a bitwise exclusive or between two node IDs as an integer. It treats nodes as leaves in a binary tree, with each nodes position determined by the shortest unique prefix of its ID. The closest leaf to an ID x is a leaf, whose ID shares the longest common prefix with x. If there are empty branches in a tree, there can be more leaves with the same longest prefix. The routing table is a binary tree, whose leaves are k-buckets. Each k-bucket contains nodes with some common prefix of their IDs, where the prefix is the position of the k-bucket in the binary tree. When a new node *n joins* it is inserted to the k-bucket according to its ID, when it is full, the bucket is split into two buckets, or the least recently node is pinged and replaced if it does not reply. To join the network, a node n needs to have a contact to an already participating node b. n inserts b into its appropriate k-bucket, then performs a lookup for its own node ID. Finally n refreshes all k-buckets further away than its closest neighbor. Refreshing means picking a random ID in the buckets range, and performing a node search for that ID. During the refreshes, n both populates its own k-buckets and inserts itself to other nodes k-buckets as necessary. To cope with node *failures* and *leaving*, Kadelmia republishes each key-value pair once an hour.

CAN (Content-Addressable Network) organizes nodes into a logical d-dimensional Cartesian space (a d-torus). This space is partitioned into zones, with a node (a peer) serving as owner of the zone. An object o is mapped to a point p(o) in the space (its key) with a uniform hash function. A node t responsible for o is the one which has p(o) in its zone. To find t, we compute p(o) and contact any node in the network with a request to route a message to t. Routing from the contacted node to t boils down to routing from one zone to another in the Cartesian space. If the point p(o) is not owned by the queried node or its immediate neighbors, the request must be routed through the CAN infrastructure until it reaches the node in whose zone p(o) lays. *Joining* a node n corresponds to picking a random point in the Cartesian space, routing to the zone that contains the point, and splitting the zone with its current owner. Node *removal* amounts to having the owner of one of the neighboring zones take over the zone owned by the departing node. Since many different paths exist between two points in the space, so even if one or more of a nodes neighbors were to crash, a node can automatically route along the next best available path. Furthermore under normal conditions a node sends periodic *update* messages to each of its neighbors giving its zone coordinates and a list of its neighbors and their zone coordinates. The prolonged absence of an update message from a neighbor signals its failure. Once a node has decided that its neighbor has died it initiates the takeover mechanism and starts a takeover timer running. Each neighbor of the failed node will do this independently, with the timer initialized in proportion to the volume of the nodes own zone. When the timer expires, a node sends a takeover message conveying its own zone volume to all of the failed nodes neighbors. On receipt of this message, a node cancels its own timer if the zone volume in the message is smaller that its own zone volume, or it replies with its own takeover message. In this way, a neighboring node is efficiently chosen that is still alive and has a small zone volume. Increasing the number of dimensions implies that a node has more neighbors, and the routing fault tolerance improves as a node has more potential next hop nodes along which messages can be routed in the event that one or more neighboring nodes crash.

Table 1: Comparison of overlay topologies

Name	Topology	Lookup and Maintenance	Routing table length and flexibility	Scalability	Fault tolerance
Flooding Random walk OurGrid BitTorrent	unstructured	$<$ TTL, $O(TTL)$	$O(N)$, dynamic	fair fair good good	good
Chord	ring	$O(logN)$, $O(logN)$	$O(logN)$, fixed	good	fair
Pastry	tree	$O(logN)$, $O(logN)$	$O(logN)$, fixed	fair	fair
Kademlia	tree/XOR	$O(logN)$, $O(logN)$	$O(logN)$, dynamic	fair	good
CAN	d-torus	$O(d * N^{1/d})$, $O(d*N^{1/d})$	$O(d)$, dynamic	good	good
Symphony	ring/small world	$O(1/k * log^2 N)$, $O(log^2 N)$	$O(1)$, dynamic	good	fair

The *Symphony* protocol [19] is a variation of Chord that exploits the small world phenomenon. The nodes are placed along a ring, and each node has two short links to its immediate neighbors and a few, constant number of long distance links (this number is not necessary identical for all nodes) built in a probabilistic way – in contrast to Chord and Pastry, which require a number of links (to fixed positions) which depends on the total number of nodes in the system. These links are chosen randomly according to harmonic distributions, which favor large distances in the identifier space in systems with few nodes, and smaller distances as the system grows. During routing a query is forwarded to the node with the shortest distance to the destination key. Symphony also employs a 1-*lookahead* approach. The lookahead list of each node records those nodes, which are reachable through the successor, predecessor, and long distance links. A node forwards messages to its direct neighbor, which promises the best progression to the destination. When a new node *n joins*, it contacts a member *b*, then chooses a random real number between 0 and 1 as an ID and it establishes a link to the node responsible for this ID. Then it uses the probability distribution function to select the long distance links (the number is determined by an estimator), and establish connections to them. When a node *x leaves*, all outgoing and incoming links to its long distance neighbors are broken. Other nodes, whose outgoing links to *x* were just broken, reinstate those links with other nodes. The immediate neighbors of *x* establish short links between themselves to maintain the ring. To tolerate *failures*, additional redundant short links need to be maintained, therefore the overhead of redundant links for fault tolerance is significantly less for Symphony than other protocols.

4.3 Summary and discussion

Table 1. shows a summary of the above described overlay architectures. We gathered the relevant performance indicator numbers for each candidate and classified them according to their routing table flexibility (freedom of neighbor selection), scalability (dramatic increase of the number of nodes makes minimal effect on performance and availability) and fault tolerance (recovery on node failures).

We can state that unstructured are more natural candidates for our problem. We have more freedom to deploy our own routing mechanism in these overlays, and all the related peer-to-peer approaches in Grid information systems chose an unstructured network. Furthermore placing new nodes with various attributes is completely unrelated to the overlay topology. On the other hand, we are interested to see, how the well studied structured approaches would affect our proposed meta-brokering system. CAN promises low maintenance and good resilience, and would leave us enough space for neighbor selection as well as Symphony. To use these structured approaches we also need to specify

a partitioning of the attribute space of all meta-brokers. The next subsection deals with this problem.

4.4 Attribute space partitioning in structured overlays

In the general case, we are looking for suitable multi-attribute range queries, and we would like to partition the attributes according to the selected solution. Latest approaches use locality preserving hashing and space-filling curves for partitioning. Researchers in the area of Grid information systems have already proposed solutions for similar problems. MAAN [20] performs multi-attribute range queries on Chord overlay. They use uniform locality preserving hashing to map attribute values to the Chord identifier space. In this approach an iterative, attribute dominated query routing algorithm is used to resolve multi-attribute based queries. This means they execute a query for each of the attributes, then merge the results. This is definitely an inefficient solution. Sword [21] is a resource discovery service for distributed systems that supports multi-attribute range queries in PlanetLab. It uses an XML-based query language with Bamboo protocol. It has dedicated DHT servers to compute hash keys for each attribute. Therefore similarly to the previous solution each attribute needs to be searched separately. LORM [22] is built on a DHT called Cycloid, which is a lookup efficient constant-degree ring overlay. Instead of collecting resource information of all values of an attribute in a single node, it lets each node be responsible for information of a specific attribute within a value range by taking advantage of the hierarchical structure of Cycloid. Though they also compared their solution to MAAN, Sword and Mercury, and reported lower overhead, they still do searches for each attribute, and tightly coupled to the Cycloid protocol. Squid [23] proposes range queries for Chord with partitioning based on Hilbert space filling curves. With this technique they can map d-dimensional attribute space to 1-dimensional index space, which can be mapped to Chord. MIDAS [24] also uses space filling curves to map a resource with d attributes to an m-bit key. To process a range query, it transforms the query into a number of search keys using a d-to-one mapping function, then it performs DHT lookups only for available keys, i.e. search keys that represent available resources. Though it only does searches for available keys, it still needs to execute more lookups. Mercury [25] researchers found DHTs inappropriate for range queries, therefore they introduced routing hubs for logical collection of nodes in the system. Queries are passed to exactly one of the hubs corresponding to the attributes that are queried, while a new data item is sent to all hubs for which it has an associated attribute. It organizes each routing hub into a circular overlay of nodes and places data contiguously on this ring: each node is responsible for a range of values for the particular attribute. In this way data partitioning among nodes can become non-uniform, thus it requires an explicit load-balancing mechanism.

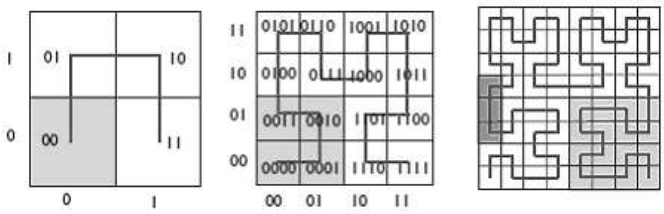

Figure 3: Hilbert SFC approximations

To simplify our problem, we discard real range queries by requiring users to specify minimum attribute values instead of ranges. In this way we can restrict our attention to multi-attribute queries. Our basic approach to solve this problem chooses CAN. It is a natural choice, since CAN builds on a d-dimensional Cartesian coordinate space. We use this space to fill with values of d attributes. Each zone of this space is stored in a chunk of a hash table. A user request consisting of n attribute-values is converted to a hashed key. To lookup this key, it will be routed in the network to the node responsible for the zone containing that key. When a new node joins a CAN network, it randomly chooses a point in the space and contacts the node responsible for that zone, then the zone is spited and the new node becomes responsible for one half zone. In our approach the point is not randomly defined, because the static attributes of the new node determines the point in the space. The other maintenance operations can be done in the same way as in CAN. But the lookup routing needs to be modified to ensure load balance: when the node is found responsible for the zone of the searched key, an additional routing step is made according to the predefined routing mechanism (to be defined in the next section), to find the most suitable neighbor for the request.

The other approach we take into account is a slightly modified usage of the Hilbert space-filling curves in Squid. With the use of this approach we can map our d-dimensional attribute space into one dimension. The transformation is done with a recursive refinement algorithm through approximations (shown is Figure 3).

We can use the Squid [23] approach (shown in Figure 4) to map this one-dimensional identifier space to any ring topology. In Squid the node IDs are randomly generated, and data elements are assigned to the nodes by using the SFC mapping to construct the data elements index that is placed on the node with the same (or the following closest) ID. In our case the nodes (Meta-Brokers) already have certain attributes. Therefore we place the n available attribute-values to the d-dimensional space (n <= d, each attribute is pre-mapped to one field). Then we use the SFC algorithm to generate its ID, and place the node to the ring according to the ID. In this case we also discard range queries, therefore user requests can be transformed to search keys with the SFC algorithm and directed with the routing of the chosen overlay to the node having the minimal required properties.

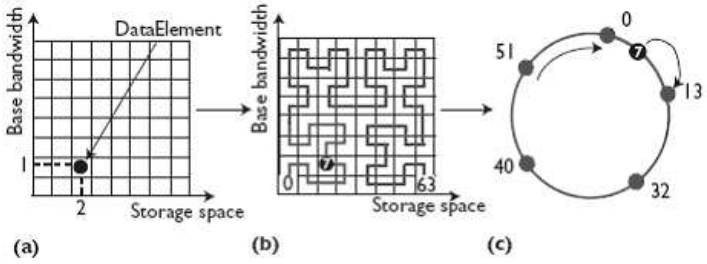

Figure 4: The approach of Squid to map 2D space to Chord

5 Routing mechanisms for P2P Meta-brokering systems

We define a general routing mechanism that takes into account the special properties of the Grid meta-brokering architecture, but still fits into the peer-to-peer topologies described above. Our overall architecture need to accomplish two goals:

- minimize the response time of finding a Meta-Broker peer that utilizes a broker with all the required attributes of the user request,
- and balancing load among the peers.

5.1 General routing architecture

We propose a general routing architecture shown in Figure 5. A typical routing scenario is the following: a user submits a job description to a known Meta-Broker. The request is placed into the local queue of the contacted peer. If the queue size (the local load) reaches a certain threshold, the request will be delegated (described later). When the Meta-Broker gets the request from the queue, it performs a local matchmaking. If a suitable broker is found, the broker description is sent back to the user. After the user performed the job submission to the selected broker, it notifies the Meta-Broker about the result of the submission. When the local match cannot find a suitable broker, the request is delegated. Delegation means a search in the overlay network. The Meta-Broker looks at its routing table to find a neighbor with the required attributes. If more peers are suitable, one is selected according to its dynamic information. If no suitable peer is found, the selection is made according to the current dynamic attributes (this field may contain a single value, or separate values for each static attribute). After a neighbor is selected, the request is sent to it. The selected peer placed the request into its queue, and performs the above described matchmaking or routing operation according to its current state. When a match is found, it is propagated back to the user through the peers participated in the lookup route, which process also updates information in the seen-jobs table of the peers. This table is used to ensure usage SLAs: each user is allowed to have a maximum predefined number of requests in the system.

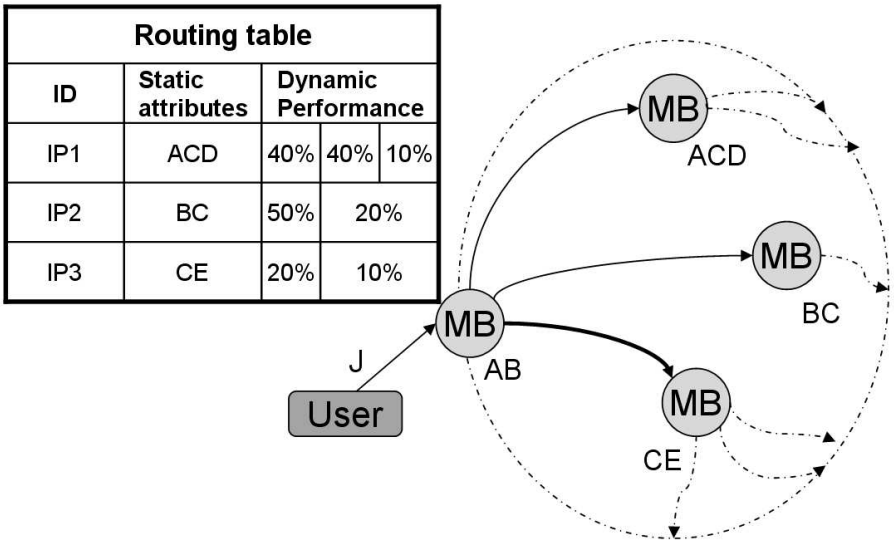

Figure 5: Routing architecture

Table 2: Family of routing mechanisms

	WHAT	WHEN	WHERE
Routing mechanisms	PARAMETER: Heuristics	PARAMETER: Threshold	PARAMETER: Prediction or Ranking function

5.2 Family of routing mechanisms

The general routing architecture described above also defines a family of mechanisms. Though we fixed that two kinds of information is stored in the routing table (static attributes and dynamic performance data), different queue thresholds, load values and scheduling policies for neighbor selection can be defined. Table 2. defines the mechanisms belonging to this family. Each mechanism needs to give an answer for the following questions:

- *What* to route? – this question means which request should be delegated. If the local queue of the actual peer is full, the request is always delegated.
- *When* to route? – this question looks for the answer, what time or under what circumstances should a request be routed to another peer. If the local matchmaking is failed, of the job has failed for all the matched brokers of the peer, the request is always delegated.
- *Where* to route? – the answer tells which peer should be contacted first to start the delegation. In unstructured networks we are free to choose the neighbor we want to contact, but in structured overlays the routes are usually fixed. In these cases we only consider this method, when we reached the target peer of the routing and still could not satisfy the request. We also not that before applying the method the peer always checks, whether the required static attributes are fulfilled by some neighbors or not. If there are some who does, only those peers are taken into consideration.

The answer for each question lays in the given parameter. Varying these parameters we can derive different mechanisms, setups. In the following we define the applicable parameters:

- *Heuristics*: by default the *FCFS* (first come first served) policy is used, so the first request in the queue is processed. The other approach is a *backfilling* based on the number of attributes of the request: the first request

Table 3: Routing mechanisms for P2P meta-brokering

	WHAT	WHEN	WHERE
RM1	Heuristic: FCFS	Threshold: 70%	Prediction: shortest wait time according to queue length
RM2	Heuristic: Backfilling with least attributes	Threshold: 70%	Prediction: shortest wait time according to queue length
RM3	Heuristic: FCFS	Threshold: 70%	Ranking function: $R = (L - L_A)^2 + (L - L_B)^2 + ...$, where L is the queue length, and L_A, L_B, ... are the number of requests with attributes A, B, ... in the queue
RM4	Heuristic: Backfilling with least attributes	Threshold: 70%	Ranking function: $R = (L - L_A)^2 + (L - L_B)^2 + ...$, where L is the queue length, and L_A, L_B, ... are the number of requests with attributes A, B, ... in the queue

with the least number of required attributes is processed.

- *Threshold*: the local queue length of the peers for the incoming requests is limited. The system need to avoid the overloading of the peers, therefore this parameter defines a queue length threshold that the number of incoming requests should not exceed. When a request arrives and the queue length is below the threshold, the request is put into the queue, otherwise it is delegated.
- *Prediction*: based on the dynamic information in the routing table the neighbor with the shortest predicted *wait time* can be selected according to the local *queue lengths* or *the number of similar requests*.
- *Ranking function*: based on the dynamic data in the routing table or according to additional cashed neighbor performance data ranking functions can be defined. After computing the ranks, the neighbor with the highest rank is selected. An example for this ranking function is based on the relation of waiting requests with the same attributes to the total number of waiting requests of a peer.

Table 3. describes the selected routing mechanisms that we will use in the evaluation of the selected approaches.

6 Conclusion and future work

In this paper we have stated the general requirements of peer-to-peer meta-brokering that aims to serve various user communities in future Grids. We examined the well studied peer-to-peer overlay topologies, and compared their general performances. Taking into account meta-brokering properties and requirements, we selected the following overlays for further evaluation: Flooding, random walk, BitTorrent, CAN, Symphony and we are further interested in experimenting with the OSPF protocol. We defined a family of mechanisms to be used for delegating user requests among the peer-to-peer network. Finally we specified four mechanisms of this family that we will apply and evaluate in each overlay topology. Our future work aims at implementing simulators for the selected peer-to-peer overlays and mechanisms, and comparing their performance results to conclude in the most appropriate solution for peer-to-peer meta-brokering in Grids.

7 Acknowledgement

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

[1] CoreGRID Project website: http://www.coregrid.net

[2] Next Generation Grids Report: Future for European Grids: GRIDs and Service Oriented Knowledge Utilities – Vision and Research Directions 2010 and Beyond, December 2006 (NGG3)

[3] H. Papadakis, P. Fragopoulou, P. Trunfio and D. Talia. Design and Implementation of a Hybrid P2P-based Grid Resource Discovery System. CoreGRID Technical report, TR-0105, Institute on Architectural issues: scalability, dependability, adaptability, CoreGRID – Network of Excellence, August 2007.

[4] I. Foster, A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), February 2003.

[5] I. Foster, A. Iamnitchi. A peer-to-peer approach to resource location in Grid environments. Grid Resource Management: State of the Art and Future Trends, J. Nabrzyski, J. M. Schopf, and J. Weglarz, Eds. Kluwer Academic Publishers, Norwell, MA, pp. 413-429, 2004.

[6] R. Bolze, F. Cappello, E. Caron, M. Dayde , F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi and T. Irena. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. International Journal of High Performance Computing Applications, 20 (4) 481-494, November 2006.

[7] P. Uppuluri, N. Jabisetti, U. Joshi, Y. Lee. P2P grid: service oriented framework for distributed resource management. IEEE International Conference on Services Computing, vol. 1., pp. 347-350, 2005.

[8] A. Kertesz, P. Kacsuk. Grid Meta-Broker Architecture: Towards an Interoperable Grid Resource Brokering Service. CoreGRID Workshop on Grid Middleware in conjunction with Euro-Par 2006, Springer-Verlag LNCS, Volume 4375/2007, pp. 112-115, 2007.

[9] A. Kertesz, P. Kacsuk. Meta-Broker for Future Generation Grids: A new approach for a high-level interoperable resource management. CoreGRID Workshop on Grid Middleware in conjunction with ISC'07 conference, Dresden, Germany, June 25-26, 2007.

[10] A. Iosup, D. H. J. Epema, T. Tannenbaum, M. Farrellee, and M. Livny. Inter-operating grids through delegated matchmaking. ACM/IEEE Conference on High Performance Networking and Computing (SC), p. 13. ACM Press, 2007.

[11] A. Iosup, D. H. J. Epema, T. Tannenbaum, M. Farrellee, and M. Livny. Inter-operating grids through delegated matchmaking. J. of Sci Prog, Special Edition "Best Paper Award at SuperComputing 2007", pp. 121, Feb 2008.

[12] P. Kacsuk, T. Kiss. Towards a scientific workflow-oriented computational World Wide Grid. Technical report, TR-115, CoreGRID – Network of Excellence, October 2007.

[13] N. Andrade, W. Cirne, F. V. Brasileiro, P. Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. Job Scheduling Strategies for Parallel Processing, 9th International Workshop, pp. 61-86, 2003.

[14] B. Cohen. Incentives build robustness in bittorrent. Workshop on Economics of Peer-to-Peer Systems, Berkeley, USA, May 2003.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. Technical Report TR-819, MIT, March 2001.

[16] A. Rowstron, P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pp. 329-350, 2001.

[17] P. Maymounkov, D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. Proceedings of IPTPS02, Cambridge, USA, March 2002.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. A Scalable Content-Addressable Network. Proceedings of ACM SIGCOMM 2001.

[19] G. S. Manku, M. Bawa, P. Raghavan. Symphony: Distributed Hashing in a Small World. 4th USENIX Symposium on Internet Technologies and Systems (USITS 2003), 2003.

[20] Min Cai, Martin Frank, Jinbo Chen, Pedro Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. Fourth International Workshop on Grid Computing, 2003.

[21] D. Oppenheimer, J. Albrecht, D. Patterson, A. Vahdat. Scalable Wide-Area Resource Discovery. UC Berkeley Technical Report UCB/CSD-04-1334, July 2004.

[22] H. H. Shen, A. Apon, C. Xu. LORM: Supporting low-overhead P2P-based range-query and multi-attribute resource management in grids. 13th International Conference on Parallel and Distributed Systems (ICPADS'07), vol. 1, no. 1, pp. 1-8, 2007.

[23] C. Schmidt, M. Parashar. Enabling flexible queries with guarantees in P2P systems. IEEE Internet Computing, Volume 8, Issue 3, pp. 19-26, 2004.

[24] V. March, Y. M. Teo. Multi-Attribute Range Queries on Read-Only DHT. 15th International Conference on Computer Communications and Networks (ICCCN), pp. 419-424, 2006.

[25] A. R. Bharambe, M. Agrawal, S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. ACM Special Interest Group on Data Communication (SIGCOMM), 2004.