Handbook of Research on P2P and Grid Systems for Service-Oriented Computing:
Models, Methodologies, and Applications

Nick Antonopoulos
University of Surrey, UK

George Exarchakos
University of Surrey, UK

Maozhen Li
Brunel University, UK

Antonio Liotta
University of Essex, UK

Volume II

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE

Hershey · New York

Director of Editorial Content: Kristin Klinger
Director of Book Publications: Julia Mosemann
Development Editor: Christine Bufton
Publishing Assistant: Kurt Smith
Typesetter: Carole Coulson
Quality control: Jamie Snavely
Cover Design: Lisa Tosheff
Printed at: Yurchak Printing Inc.

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com/reference

Library of Congress Cataloging-in-Publication Data

Handbook of research on P2P and grid systems for service-oriented computing : models, methodologies and applications / Nick Antonopoulos ... [et al.].
p. cm.
Includes bibliographical references and index.
Summary: "This book addresses the need for peer-to-peer computing and grid paradigms in delivering efficient service-oriented computing"--Provided by publisher.
ISBN 978-1-61520-686-5 (hardcover) -- ISBN 978-1-61520-687-2 (ebook) 1. Peer-to-peer architecture (Computer networks)--Handbooks, manuals, etc. 2. Computational grids (Computer systems)--Handbooks, manuals, etc. 3. Web services--Handbooks, manuals, etc. 4. Service oriented architecture--Handbooks, manuals, etc. I. Antonopoulos, Nick.
TK5105.525.H36 2009
004.6'52--dc22
2009046560

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 25
Data Replication in P2P Systems

Vassilios V. Dimakopoulos
University of Ioannina, Greece

Spiridoula Margariti
University of Ioannina, Greece

Mirto Ntetsika
University of Ioannina, Greece

Evaggelia Pitoura
University of Ioannina, Greece

ABSTRACT

Maintaining multiple copies of data items is a commonly used mechanism for improving the performance and fault-tolerance of any distributed system. By placing copies of data items closer to their requesters, the response time of queries can be improved. An additional reason for replication is load balancing. For instance, by allocating many copies to popular data items, the query load can be evenly distributed among the servers that hold these copies. Similarly, by eliminating hotspots, replication can lead to a better distribution of the communication load over the network links. Besides performance-related reasons, replication improves system availability, since the larger the number of copies of an item, the more site failures can be tolerated. In this chapter we survey replication methods applicable to p2p systems. Although there exist some general techniques, methodologies are distinguished according to the overlay organization (structured and unstructured) they are aimed at. After replicas are created and distributed, a major issue is their maintenance. We present strategies that have been proposed for keeping replicas up to date so as to achieve a desired level of consistency.

INTRODUCTION

In a peer-to-peer (p2p) system, a large number of nodes share data with each other. The participation of nodes is highly dynamic; nodes may enter and leave the system at will. Since, it is not possible to maintain links with all nodes, for performance as well as for privacy and anonymity reasons, each node maintains links with a selected subset of other nodes, thus forming an *overlay network*. A message

DOI: 10.4018/978-1-61520-686-5.ch025

between any two nodes in the p2p system is routed through this overlay network which is built on top of the physical one. Thus, two nodes that are neighbors in the overlay network may be many links apart in the physical network.

The overlay network is built to facilitate the operation of a p2p system. In data sharing p2p systems, a basic functionality is discovering the data of interest. A look-up query for data items may be posed at any node in the overlay. The query is then routed through the overlay to efficiently discover the nodes that hold the requested data items. Such query routing must be achieved by contacting as "small" a number of nodes in the overlay as possible and by maintaining as "little" state information at each overlay node as possible.

There are two basic types of overlays: structured and unstructured ones. To assist lookup, *structured overlays* map (keys of) data items to nodes. In structured overlays, the mapping is usually done by hashing the key space of the data items to the id space of nodes. Thus, each node in the overlay maintains a partition of the data space. In structured overlays, lookup reduces to locating the node in the overlay that is responsible for the corresponding data partition. In *unstructured overlays* on the other hand, there is no correlation between nodes and data items.

There are a number of issues regarding the design of a structured p2p system. One design dimension refers to the geometry of the overlay, that is, its structural characteristics. Structured overlays usually follow a regulated topology, such as a ring, tree or grid. Then, upon entering the p2p system, each node takes a specific position in the overlay network. Another design choice is how data are mapped to nodes. The mapping must be fair so that nodes receive similar loads even when the data sets or the operations are skewed. All designs aim at supporting efficiently the basic operations of the overlay, that is, its construction, its incremental maintenance when nodes enter or leave the system, and its search. Efficiency must be achieved even in the case of high churn, where maintaining the overlay structure incurs a high cost. Most structured overlays guarantee lookup operations that are logarithmic in the number of nodes. Finally, overlays differ with respect to the range of different types of queries that they support.

In unstructured overlays, the topology is not a rigid one. Unstructured overlays are formed by the nodes as they join the system by either selecting randomly a node from a known list of participating peers or by following some loose rules regarding this selection. The resulting topology may have certain properties, however there is no assumption regarding the way the data space is mapped to the address space of the nodes in the overlay. To locate data of interest, a node queries its neighbors in the overlay, which in turn query their neighbors, and so on, until the query hits on a node holding the item. However, this procedure provides no guarantees on the complexity of search.

The topology of an unstructured overlay is built up over time in a decentralized manner as peers join and leave the system. In many existing systems, upon joining the network, a peer selects to connect to another peer essentially at random. In these systems, topologies often tend towards a power-law degree distribution, where some long-lived peers have many connections, while most other peers have a few connections.

To improve performance of lookup, caching or replication[1] of either data or search paths (or both) is possible. Besides improving search, replication may also assist in providing load balancing. Further, replication improves fault tolerance and the durability of data items.

Replication increases the number of copies for each shared piece of data in the system. By doing so, the probability that some or all the data is temporarily or permanently lost significantly decreases, thus the *dependability* of the system in terms of *reliability* and *availability* is increased. Additionally, by having more copies for popular data items, the load for routing and answering

queries can be evenly distributed among the servers that hold the copies. This way, the *performance* of query processing is increased in terms of *throughput* and *response time*, since congestions in "hot" servers are avoided. Placing data closer to their requesters, as done by replication, also improves the performance of query processing.

Replication may also improve *data recall*. In structured overlays data recall is not an issue, contrary to unstructured overlays. While unstructured overlays which adopt flooding-based techniques are effective for locating popular data, they are poorly suited for locating rare data. Thus, by replicating the rare data the probability of locating it during query processing increases, consequently increasing data recall. However, replication may affect *data freshness*. P2p systems consist of autonomous peers that can arbitrarily delete or update their content, thus the replicated data can become stale if not updated properly. Updating replicas and cache entries in a p2p system mainly aims at providing soft-state guarantees. Hence, query processing might encounter out-of-date copies of data, thus failing to achieve result freshness.

The cost associated with replication includes the storage cost as well as the maintenance cost in the case of updates. Updates are either initiated by the owner of the copy that was modified (push-based updates) or by the holder of the copy (pull-based updates). Often entries are associated with a time-out value, whose expiration signals the removal of the entry, thus providing only soft-state consistency.

There are two main choices on *what* to replicate: peers may replicate the data items themselves or their location information (index). There is an obvious difference in the required storage space among the two choices. Also, replicating indices does not improve reliability or availability since it does not lead to more physical copies. However, both the placement and the update mechanisms that have been proposed are applicable to both data and index/location replication. Thus we will treat the two cases as equivalent and shall not differentiate between them in what follows.

The rest of this chapter is organized as follows. Replication in unstructured p2p systems is considered first, followed by replication in structured overlays. The results and techniques differ since unstructured p2p systems are treated in essence as "unknown" complex networks, while in structured p2p systems the topology is more or less known in advance and replication schemes may take advantage of it. Next, the problem of keeping the replicas up-to-date is considered where, again, we distinguish between the techniques used for the two types of overlay topology. A final section summarizes the chapter.

REPLICATION IN UNSTRUCTURED P2P NETWORKS

Unstructured topologies evolve in more or less unpredictable ways, as nodes leave and join the overlay at arbitrary positions. As a consequence, they can be effectively modeled as a random graph, or *Erdős-Rényi random graph*. Such a graph with N nodes can be equivalently described in two ways: it is a graph where each of the possible $N(N–1)/2$ edges is present with some fixed probability p; or it is a graph selected uniformly at random among all possible graphs on N nodes and M edges. This is a simple but powerful model, where it can be shown that the degrees of participating nodes follow a Poisson distribution. However, real-world networks have exhibited a behavior that is closer to power-law degree distributions: there are a few highly connected nodes and a large number of nodes with small degree. Thus power-law random graphs or simply *power-law graphs* have been quite popular as a model for p2p networks. Although many replication schemes do not depend on the particular model used for the p2p topology, the choice of this model has strong implications on the analysis of some replication objectives (e.g. determining an optimal number of replicas).

Since in unstructured p2p systems there is no information on where the desired piece of data is placed, mostly "blind" search procedures are used and look-up queries get propagated through the network so as to locate peers offering the data. The usually employed blind search strategies are flooding and random walks, along with their variations, and proposed methods for replication routinely depend on the particular search strategy used for locating items. In *flooding*, a node that receives a query message propagates it to all its neighbors, unless it knows about the item in question. Flooding usually results in fast responses but can easily overload the network with messages, a large percentage of which are unnecessary duplicates. In variations of flooding the query message is sent only to a selected subset of a peer's neighbors, based on certain rules.

During a *random walk* on the other hand, a contacted peer propagates the query to exactly one of its neighbors, selected uniformly at random. Proposed variations bias the selection of the neighbor according to various criteria. This type of search is particularly efficient, especially in certain topologies, but usually results in slow responses. Things can be significantly improved if multiple random walkers are deployed simultaneously. It is a known fact that using random walks, the probability of visiting a node is proportional to its degree, i.e. random walks go through high-degree nodes more frequently (since they are more easily reached due to their large number of neighbors).

To avoid high delays and an overwhelming number of messages, queries are allowed to propagate for a limited number of steps, which is the so-called *time-to-live* (TTL) parameter. The TTL value is included in every query message and gets decremented by 1 at each visited peer. Peers that observe a zero TTL value stop propagating the search any further. It should be obvious, that TTL-limited query propagation may result in *unsuccessful* searches.

In this section we present replication schemes for unstructured p2p networks. The section consists of two parts. In the first part, we discuss theoretical results regarding the optimum number of replicas of each item in the system and practical ways of achieving this. In the second part, we examine methods for placing the generated replicas onto the p2p system nodes.

Number of Replicas

Assume that there are N peers participating in the network and m different data items to be shared among peers. Each peer on average has a storage capacity for storing ρ replicas of data items and the network has a total budget of R copies overall ($R=N\rho$). The *query rate* or *popularity* of item i, q_i, is the probability that any arbitrary peer issues a request for item i.

The problem of determining what is the optimal replica configuration is discussed by Cohen & Shenker (2002), for overlays that are modeled as Erdos-Renyi random graphs. Specifically, the authors deal with the problem of how many replicas of each data item should exist in the network so that the search overhead for locating the item is minimized, with the constraint of fixed storage capacity in the network. Given the query rates for each data item, the objective is determining which fraction p_i of R should be allotted to each data item i, so that the expected search size (ESS), i.e. the number of peers probed during the search process is minimized.

Two natural ways of replicating data items, namely uniform and proportional replication, are shown to be suboptimal under the above assumptions. In *uniform replication* (UR) the same number of replicas is created for each data item, regardless of its query rate. In *proportional replication* (PR) the number of replicas for each data item is proportional to the popularity of the item, so that $p_i \propto q_i$. Although it seems natural to create more replicas for more popular data items so

as to favor most common queries, this is done at the expense of rare ones. In fact, it can be shown that the ESS for a successful query is the same for both uniform and proportional replication strategies. The optimal configuration, proved to minimize the expected search size, is *square-root replication* (SR), where the number of replicas of each data item is proportional to the square root of its query rate, i.e. $p_i \propto \sqrt{q_i}$.

Since global knowledge is unavailable at each peer, the authors also consider ways of realizing square-root replication using simple distributed protocols. In one of the simplest, the number of copies created after a successful search is equal to the size of the search, i.e. the number of peers probed during search. At steady state, and under reasonable assumptions, this simple strategy can be shown to converge to SR. The only critical assumption is that the fixed storage capacity of each node is managed through replacement policies that do not depend on the identity and the query rate of the stored items. As such, at a full node, the item that must be deleted so as to make room for another replica cannot be given by usage-based policies such as LRU or LFU but rather by policies like FIFO or random deletions.

Notice that although the idea is quite simple, the size of the search is normally not known. Lv, Cao, Cohen, Li & Shenker (2002) discuss two practical strategies that try to approximate the search size, namely owner and path replication. In *owner replication*, which is used in Gnutella (2003), when a search for a data item is successful (only) the peer that initiated the search process stores a replica of the data item. In *path replication*, each query keeps track of the path it follows from the peer that issues the request to the peer that offers the data item. When the search succeeds, all peers in this path are forced to keep a replica of the data item. Clearly, path replication comes quite closer to approximating the search size and experimental results show that it comes close to achieving SR. Path replication is used on Freenet (Clarke et al, 2002) where all nodes along the search path are forced to create a replica using an *insert* message. Nodes keep both the item and a pointer to the original data holder of the file. The replacement policy used to manage the finite storage space at each node is LRU. Subsequent incoming requests of evicted files, however, can still be served for much longer since the node also holds a pointer to the original holder. It is worth noting that LRU was shown, through detailed simulations (Zhang, Goel, & Govindan, 2004), to be responsible for deteriorating Freenet performance under heavy loads. By just altering the replacement policy so as to force local data clustering, the authors managed to achieve high percentages of successful queries and with small hop counts, even under heavy traffic.

Path replication works only for search strategies based on random walks. Even in such cases however, it may fail to discover the search size. If multiple walkers are used (Lv, Cao, Cohen, Li & Shenker, 2002), only the successful ones will be used to create replicas while the others will be ignored, creating a number of replicas smaller than the total number of visited nodes. To closely approximate the number of probes, Leontiadis, Dimakopoulos & Pitoura (2006) propose the *Pull-then-Push* (PtP) strategy, where replica creation becomes a responsibility of the inquiring peers. PtP replication consists of two phases: the *pull phase* during which the requesting peer is trying to locate the desired data item and the *push phase* which begins after a successful search whereby the requesting peer transmits the data item and causes other peers to hold replicas of it. In order to achieve SR, the number of peers that are probed during the push phase should be equal to the number of peers that where probed during the pull phase. Therefore, it is essential that the same search strategy is used both for searching for the data item (pull) and the data item transmission (push) and with the same hop limit (TTL). Finally, every peer that is probed during the push phase is forced to hold a replica of the data item. PtP works for both flooding and random-walker based strategies and leads easily to SR.

For Erdos-Renyi random graphs, if flooding-based search is used and if the objective is to minimize the search *time* (as opposed to search size) then proportional replication is the optimal configuration as shown by Tewari & Kleinrock (2005). Search time is the shortest distance from the inquiring node where a replica of the queried item is found. Optimality is achieved under the assumption of an ideal "controlled" flooding strategy where search stops immediately when the data item is located. A practical but slightly suboptimal search mechanism that approximates controlled flooding is the expanding rings method described in Lv, Cao, Cohen, Li & Shenker (2002). PR has additional benefits as well, e.g. the minimization of used network bandwidth (estimated as the average number of links traversed per download). Tewari & Kleinrock (2006) additionally consider practical ways of achieving PR. They basically follow owner replication (an inquiring node keeps a copy for itself), which should naturally lead to a number of replicas proportional to the request rates of data items. Again, a crucial factor is the replacement strategy used in managing each node's fixed storage space. Experimentally, all known strategies have good but not optimal performance, with LRU and LFU the better ones. Almost perfect PR can be achieved with a replacement strategy based on random evictions combined with additional replica creations even if the item is found in the inquiring node's storage space.

Placement of Replicas

The works in the previous section deal mostly with determining the optimum number of replicas and with ways to achieve this number, under certain assumptions and constraints. Another approach is to determine *where/how* to place the replicas (without striving for a particular number of them) so as to optimize some objective. For example, the objective may be the minimization of search size or the maximization of the percentage of successful searches.

Gia (Chawathe, Ratnasamy, Breslau, Lanham & Shenker, 2003) has been proposed as an improvement of Gnutella to exploit peer heterogeneity and includes mechanisms that dynamically adapt the overlay topology and the search algorithms. The topology adaptation mechanism ensures that high-capacity nodes are the ones that have high degree. Gia follows *one-hop replication*: an index of the content of every peer is replicated to its immediate neighbors. The rationale behind this is that since high-degree nodes are visited more frequently and high-degree nodes are the ones with high capacity, having them know the content of their neighbors will make them capable of providing answers to a greater number of queries.

Jia, Pei, Li & You (2005) compare various mechanisms for the problem of replica placement in power-law networks. They consider replication of location information (i.e. not the actual data) so as to maximize the overall performance of search queries. The spread mechanisms considered are flooding, percolation-based (randomized) flooding, random walks and high-degree random walks (HDRW). The later is a variation of random walks where a visited peer selects the next peer randomly among its highest-degree neighbors. By spreading location information along an HDRW, more information reaches high-degree nodes more quickly. As a result, because it is well known that search queries gravitate towards the high-degree nodes in the network, potentially more searches will be resolved successfully and quickly. This was confirmed through simulations which showed that for the same message overhead, spreading replicas by HDRW results in better search performance than the other mechanisms, under both flooding-based and random walk-based search.

Morselli, Bhattacharjee, Marsh & Srinivasan (2005) propose *LMS* (Local Minima Search), a search method and replication protocol. Assuming that both peers and data items obtain ids uniformly at random from a given large set (so as to guarantee uniqueness with high probability), the replication mechanism tries to replicate an item

with id *i* to peers with id 'close' to *i*. Such a node is called a local minimum for item *i* in that its id is closest to *i* among the ids of all peers in the node's *h*-hop neighborhood, where *h* is a given parameter. A random walk is used first, followed by a deterministic walk that progresses towards the closest local minimum node by selecting at each step the neighbor with the smallest distance from *i*. When this random local minimum is reached, a replica is created if there is not one there already; otherwise, the process is repeated with a random walker of double length. For locating the item, the same procedure is used. A local minimum that receives the query replies with the replica or with a failure message depending on whether it stores the item or not. To improve success rate and response time, multiple such walkers can be utilized. The protocol can achieve quite high query success probabilities but at the expense of a possibly large number of replicas ($O\left(\sqrt{n / d_h}\right)$, where d_h is the minimum size of an *h*-hop neighborhood), which can be a problem if the storage space in each peer is limited.

Maximization of the probability of success is also the subject of the work by Sozio, Neumann & Weikum (2008). They consider the problem of replica placement in arbitrary networks that are searched by random walks. Given the peer capacities and the query rates q_{ij}, i.e. the fraction of all queries (issued in the whole network) for data item *i* by peer *j*, the problem of finding an assignment of replicas to peers so as to maximize the probability of a successful query is shown to be related to the multi-knapsack problem, where there is a set of bins with given capacities and a set of elements each with size and profit and the aim is to find a feasible packing that maximizes the profit. The problem can be tackled by good approximation algorithms, which however are centralized. The authors present *P2R2*, a distributed algorithm to solve the problem, which is based on each peer *j* keeping a special counter for each data item *i*, r_{ij}. The counter r_{ij} is incremented for each query about *i* that passes through node *j* and is unsuccessful or is satisfied by a peer with larger id. This requires that certain information is piggybacked on the query messages and that random walks are always unfolded to their maximum length even if the item is located at some step earlier than the expiration of TTL. P2R2 leads to a probability of query success which is within a factor of 2 from the optimal.

Summary of Replication in Unstructured P2P Systems

Replication methods that are applicable to unstructured p2p systems provide answers to the questions of how many replicas are created for each data item, according to which optimization criteria, and where those replicas are placed. Table 1 summarizes how replication methods described above deal with each of these issues.

REPLICATION IN STRUCTURED P2P NETWORKS

In structured peer-to-peer networks, data items are stored at specific nodes of the overlay. The mapping of data items to overlay nodes is in general achieved through appropriate hash functions that support a hash table interface with primitives put(key, value) and get(key), where key is the identity of a data item (for instance, the file name). In addition, the nodes in the overlay are organized in rigid topologies, such as a multidimensional ring, grid or an n-dimensional cube. This way, items can be located efficiently. The routing messages for locating an item follow a deterministic path from the requester to the owner of the item.

Ideally, hashing should be such that peer are responsible for roughly the same number of data items. The common underlying assumption for achieving this is that data keys, and in some cases node identifiers, are randomly chosen. However, due to skeweness in the data population, this is

Table 1. Summary of replication methods for unstructured p2p systems

	How many	Where/How	What	Goal
Sqaure-root Replication (Cohen et al, 2002)	Proportional to the square root of the query rate of each data item	-	Data items	Minimum expected search size
Owner Replication (Lv et al, 2002)	Proportional to the query rate of each data item	Only to the requesting peer	Data items	Minimum expected search size
Path Replication (Lv et al, 2002)	Proportional to the number of probes for locating the item	Along the path from the requesting peer to the provider peer	Data items	Minimum expected search size
Pull-then-Push Replication (Leontiadis et al, 2006)	Proportional to the number of probes for locating the item	-	Data items	Minimum expected search size
Proportional Replication (Tewari et al, 2005)	Proportional to the query rate of each data item	-	Data items	Minimum expected search time
Gia (Chawathe et al, 2003)	Equal to the degree of each node	1-hop neighborhood	Location information	Maximum success rate
HDRW (Jia et al, 2005)	Proportional to the number of probes for locating the item	Along a degree-biased search path	Location information	Good success rate and search size
LMS (Morselli et al, 2005)	-	At peers considered as local minima for a data item	Data items	Good success rate and search size
P2R2 (Sozio et al, 2008)	-	At peers resulting in greatest success rate for a data item	Data items	Maximum success rate

not always the case. Furthermore, even when the data items are evenly distributed among the peers of the overlay, non-uniform query workloads may lead to an uneven workload distribution among the peers, resulting in potentially overloading the peers that maintain popular items. Thus, replication techniques are central in achieving both data and workload balance in structured peer-to-peer systems. Furthermore, as in unstructured p2p systems, replication is used to handle peer failures and departures and increase availability. Scalability and performance are also central goals of replication in this context as well.

Most structured p2p systems provide search time logarithmic to the number of nodes in the overlay. Enhancing the basic structured overlays with replication can lead to achieving constant search time in most cases. Since the search path followed for locating an item is deterministic, this can be achieved by proactively placing replicas of each item on appropriate nodes on its search path.

Another common mechanism for implementing replication in DHT-based p2p systems is based on replicating each data item at the k neighbors of the node holding it. Nodes close to each other on the overlay are not likely to be physically close to each other, since the id of a node is based on a hash of its IP address. This provides the desired independence of failures. Besides availability, these replicas can be used to improve query latency; they allow choosing among the k replica holders the one that has the lowest reported latency. Fetching from the lowest-latency replica has also the desired side-effect of spreading the load of serving a lookup over the replicas.

An alternative mechanism for realizing replication in DHTs is by using multiple hash functions. By doing so, a data item is mapped and stored at more than one node. This results in increasing availability as well as in improving load balancing. Furthermore, latency may be improved by selecting at each routing step the neighborhood or route that is closest either to the query or to the current node.

Finally, caching in DHTs is based on placing copies on the lookup path, similar to path replication in unstructured p2p systems or to the requestor of an item similar to owner replication.

Replication in Representative Structured P2P Systems

CHORD

Chord (Stoica, Morris, Karger, Kaashoek & Balakrishnan, 2001) is a popular DHT-based p2p system. The Chord protocol uses a variant of consistent hashing to assign to each node and data key an m-bit identifier. The identifier of a node is chosen by hashing its IP address, while the identifier of an item is produced by hashing its key. Identifiers are ordered in an identifier circle modulo 2^m. A data item with key i is assigned to the first node whose identifier is equal to or follows i in the circular space. This node is called the *successor* of i.

Each Chord node maintains two sets of neighbors, its successors and its fingers. The successor nodes immediately follow the node in the identifier space, while the finger nodes are spaced exponentially around the identifier space. Each node has a constant number of successors and at most m fingers. The i-th finger of the node with identifier p is the first node that succeeds p by at least 2^{i-1} on the identifier circle, where $1 \leq i \leq m$. The first finger node is the immediate successor of p, where $i = 1$. For a Chord network with N nodes, the number of routing hops for a lookup is O(log N), while each node only needs to maintain pointers to O(log N) neighbors.

The core Chord system does not provide for replication or caching. Instead, the replication mechanisms are left as a responsibility of the higher layer applications that use Chord. A typical method for an application to replicate data items in Chord is by using multiple hash functions to store each data item under distinct Chord keys. Furthermore, an application can store replicas of a data item with key i at the k nodes succeeding the successor of i. This is facilitated by the *successor-list* mechanism supported by Chord. In Chord, each node maintains a successor-list with its r nearest successors on the Chord ring. When a node notices that its successor has failed, it replaces it with the first live entry in its successor list. The fact that a Chord node keeps track of its successors means that it can notify the application when successor nodes fail or recover and thus when the application should propagate new replicas.

CAN

The *Content Addressable Network (CAN)* (Ratnasamy, Francis, Handley, Karp & Shenker, 2001) is another popular DHT-based structured p2p network. It uses a virtual d-dimensional Cartesian coordinate space or d-torus to store (key, value) pairs. Upon entering the system, each node is assigned a zone of this space. The key of each data item is mapped onto a point in the coordinate space using a uniform hash function. Then, the item is stored at the node that owns the zone within which the point lies. Each CAN node maintains a coordinate routing table that holds the IP address and virtual coordinate zone of each of its immediate neighbors in the coordinate space. In a d-dimensional coordinate space, two nodes are neighbors if their coordinates overlap along d–1 dimensions. Each node routes a message for an item with key i towards its neighbors whose coordinates are closer to that of i. Intuitively, routing works by following the straight line path through the Cartesian space from source to destination coordinates.

CAN supports a variety of replication mechanisms. A node that is overloaded with requests for a specific data item can replicate the data key at each of its neighboring nodes. The key of a popular data item is thus eventually replicated within a region surrounding the original storage node. A node holding a replica of a requested data key may, with a certain probability, choose to either satisfy the request or forward it. The second mechanism is based on the observation that each node can maintain multiple, independent coordinate spaces and be responsible for a different zone in each coordinate space. Each such coordinate space is a

called a *reality*. For a CAN with *r* realities, a single node is assigned *r* coordinate zones and holds *r* independent neighbor sets. This form of replication improves data availability. Multiple realities also improve routing fault tolerance, because if routing fails in on one reality, messages can continue to be routed using the remaining realities. It also provides for neighbor selection. To forward a message, a node can check all its neighbors on each reality and forwards the message to the neighbor whose coordinates are the closest to the destination. Thus, using multiple realities reduces the path length and hence the overall CAN path latency. Yet another replication mechanism for improved data availability is to use *k* different hash functions to map a single key onto *k* points in the coordinate space. In this case, a (key, value) pair becomes unavailable only if all *k* distinct nodes become simultaneously unavailable. In addition, queries for a particular hash table entry could be sent to all *k* nodes in parallel thereby reducing the average query latency. Instead of querying all *k* nodes, a node may choose to retrieve an entry from that node which is closest to it in the coordinate space. Finally, with *zone overloading,* a zone may be assigned to more than one node. Each node maintains a copy of all items mapped to the zone to increase availability.

In addition to replication, CAN also supports caching. A CAN peer maintains a cache of the data keys it has recently accessed. Thus, before forwarding a request for a data key towards its destination, a peer first checks whether the requested data key is in its own cache and if so, it can itself satisfy the request without forwarding it any further.

PAST

Pastry (Rowstron & Druschel, 2001a) is a peer-to-peer routing substrate for supporting a variety of applications. In Pastry, each node is assigned a quasi-random 128-bit node identifier (nodeId). The nodeId is used to indicate the position of the node in a circular identifier space, which ranges from 0 to 2^{128} - 1. Both nodeIds and data keys are treated as a sequence of digits with base 2^b. Pastry routes messages to the node whose nodeId is numerically closest to the given key.

To support routing, each node maintains a routing table, a neighborhood set and a leaf set. The routing table of each node *p* is organized into $\log_{2^b} N$ rows with 2^{b-1} entries each. The 2^{b-1} entries at row *i* of the routing table refer to those nodes whose nodeId has the same first *i* digits with the nodeID of *p*, but a different $i + 1$ th digit. Each such entry contains the IP address of one of the potentially many such nodes, usually the one physically closest to *p*. If no such node is known, the routing table entry is left empty. The leaf set includes the set of nodes with the *L*/2 numerically closest larger nodeIds to *p*, and the L/2 nodes with numerically closest smaller nodeIds to *p*. Lastly, the neighborhood set contains the nodeIds and IP addresses of the *M* nodes that are physically closest to *p*. This set is not normally used for routing, but it is used for maintaining physical locality properties. Typical values of *L* and *M* are 2^b or 2^{b+1}.

Given a message, each node *p* first checks whether the key falls within the range of nodeIds covered by its leaf set. If so, the message is forwarded directly to this node. Otherwise, the routing table is used to forward the message to a node that shares a common prefix with the key by at least one more digit (or *b* bits) than the current node *p*. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix that is as long as the one shared with *p*, but is numerically closer to the key than the nodeId of *p,* by using the leaf set.

In Pastry, replication is not implemented directly. Instead, Pastry provides to the applications built on top of it the functionalities necessary for implementing replication. In particular, applications can use the information maintained in the routing table and the leaf and neighborhood sets to decide where to place replicas. Further, Pastry

provides mechanisms for handling peer failures, such as periodically exchanged keep-alive messages.

In particular, *PAST* (Rowstron & Druschel, 2001b) is a p2p file storage system that relies on Pastry to provide routing file queries, multiple replicas of files, and caching for additional copies of popular files. For improved availability, PAST creates k replicas of each file and places them to k different peers whose nodeId is numerically closest to the 128 most-significant-bits of the identifier of the file (fileId) among all live nodes, where k is the replication factor. Since by the way identifiers are assigned to nodes, there is no correlation between these identifiers and the geographic location, network connectivity, ownership or jurisdiction of the nodes, the k nodes selected for storing the replicas are highly likely to be diverse in all these aspects and thus unlikely to conspire or be subject to correlated failures or threats.

For maintaining good system-wide storage utilization, PAST uses replica and file diversion. Replica diversion is achieved by allowing a peer that is not one of the k numerically closest peers to the fileId of a file to maintain a replica of it, if it is in the leafset of one of those k peers. This improves utilization within the nodes in the leaf set. File diversion is performed when the entire leaf set of a node is reaching capacity. A file is diverted to a different part of the identifier space by choosing a different salt in the generation of its fileId.

Replication in PAST aims mainly at improving fault-tolerance and partly at balancing the query load or reducing latency. Creating additional copies for popular files is achieved through caching. PAST uses a form of path replication: copies of files are cached along the search path for the file. In caching a file, however, PAST also considers the storage available at a node. A file is cached at a node only if its size is less than some fraction c of the current cache size of the node.

Kademlia

Kademlia (Maymounkov & Mazieres, 2002) is a distributed DHT-based p2p system that employs 160-bit identifiers for both participating nodes and file keys. Every node maintains information about (key, value) pairs "close" to itself. The distance between two objects (keys or nodes) in the 160-bit key space is measured as the bitwise XOR of their ids interpreted as an integer.

Each Kademlia node maintains a routing table that consists of 160 buckets. The ith bucket of a node contains up to k entries pointing to nodes in distance between 2^i and 2^{i+1}. The buckets are kept constantly updated, as for every received message the node either enters the sender's id in the tail of the appropriate bucket (possibly discarding another entry) or rearranges the entries in the bucket (by refreshing its contact with the least recently seen node).

Lookup is implemented by contacting nodes that have ids close to the id of the requested item. In particular, the inquiring node selects some of the closest nodes from its routing table and queries them, learning about other nodes even closer to the id in question, and so on. The end result is that within O(log N) steps (with high probability) the node forms a list of the k closest nodes to the requested id.

Replication in Kademlia exploits the lookup procedure. To store a (key, value) pair, a Kademlia node first locates the k closet nodes to the key, as described above. Then, it sends them a STORE message, creating k replicas of the item. Replicas are additionally created dynamically: after each successful search, a replica is placed in the closest node to the key that did not contain the item. The reason behind this is the unidirectionality property of the XOR distance metric which ensures that all searches for an item converge along the same path, no matter where they originate from; placing replicas on the lookup path leads to faster searches, avoiding at the same time hot spots. To ensure the freshness of replicas, Kademlia requires periodic re-publishing of the (key, value) pairs.

P-Grid

P-Grid (Aberer, Cudré-Mauroux, Datta, Despotovic, Hauswirth, Punceva & Schmidt, 2003; Aberer, Datta, Hauswirth & Schmidt, 2005) is a p2p data management system based on building a virtual distributed trie. Data keys are composed by a number of bits. The data key space is recursively bisected so that the resulting partitions carry approximately the same load. One or more peers are associated with each partition. Each partition is uniquely identified by a bit sequence. The bit sequence of a partition is called the path of the peer associated with the partitions. These bit sequences induce a trie structure which is used to implement prefix routing by resolving a key lookup a bit at a time. Each peer maintains for each bit position of its path one or more randomly selected references to a peer that has a path with the opposite bit at this position.

P-Grid implements two forms of replication for fault-tolerance. First, multiple peers are associated with the same key space. This is called structural replication. Then, multiple references are kept in the routing tables, thus providing alternative access paths.

General Replication Strategies

Selective Placement to Reduce Latency

Beehive (Ramasubramanian & Sirer, 2004) is a general replication framework that operates on top of any DHT that uses prefix-routing, such as Chord. In such systems, routing is performed by successively matching a prefix of the data identifier against node identifiers. In general, at each routing step, the query reaches a node that has one more matching prefix with the query than the previous node on the path. A query traveling k hops reaches a node that has k matching prefixes. The central observation behind Beehive is that the length of the average query path will be reduced by one hop when a data item is proactively replicated at all nodes logically preceding that node on all query paths. For example, replicating the object at all nodes one hop prior to their successor node decreases the lookup latency by one hop. This can be applied iteratively to disseminate items widely throughout the system. Replicating an item at all nodes k hops or lesser from the successor node will reduce the lookup latency by k hops.

Beehive controls the extent of replication in the system by assigning a replication level to each item. An item at level i is replicated on all peers that have at least i matching prefixes with the item. Queries to data items replicated at level i incur a lookup latency of at most i hops. Data items stored only at their successor peers are at level $\log(N)$, while items replicated at level 0 are cached at all the peers in the system. The goal is to find the minimal replication level for each item such that the average lookup performance for the system is a constant C number of hops. Naturally, the optimal strategy involves replicating more popular items at lower levels (on more peers) and less popular items at higher levels. An analytical model provides Beehive with closed-form optimal solutions indicating the appropriate levels of replication for each item. In addition, a monitoring protocol based on local measurements and limited aggregation estimates the relative item popularity and the global properties of the query distribution. These estimates are used, independently and in a distributed fashion, as inputs to the analytical model which yields the locally desired level of replication for each item. Finally, a replication protocol proactively makes copies of the items around the network.

PopCache (Rao, Chen, Fu & Bu, 2007) is based on the observation that in structured p2p system, each node can be seen as the root of a tree. In particular, each node p can be treated as the root of a k-ary tree with its k direct neighbors of p connected by k links as the first level children, the neighbors of neighbors of p added with k^2 links as the second-level children and so on until level $\log_k(N)$, where N is the number of nodes.

Let us denote with T_p the tree having node p as its root. The search from some node to p is the process of greedily approaching the root p along the bottom-up path of T_p. For each data item i, PopCache utilizes the tree T_p that is rooted at the node p responsible for item i. Assume that we want to create a total of m copies for i. First, k replicas of i are placed on the first level of T_p, then k^2 copies of i are placed on the second level and so on, until all m replicas are created. For deriving the optimal number of copies per item, two optimization criteria are considered (a) given a maximum number of copies, how to minimize the average latency per query (MAX PREF), and (b) given a targeted threshold how to minimize the number of replicas (MIN COST). The first criterion is similar to that use in unstructured p2p systems, however, in this case, the optimal number of copies per item follows a different proportional principle.

Range Queries

HotRoD (Pitoura, Ntarmos & Triantafillou, 2006) uses replication over Chord to provide fair load distribution in the case of range queries. The key of this implementation is the use a locality-preserving hash function that preserves the ordering of data by mapping consecutive data values to neighboring peers. A range query is pipelined through those peers that store ranges of entries that overlap with the query range. HotRoD detects overloaded peers and distributes their access load among other, under-loaded ones, through replication. In particular, each peer keeps track of the number of times, it was accessed during a time interval T, and the average low and high bounds of the ranges of the queries it has processed during this interval. A peer is characterized as overloaded or *hot*, if this number exceeds a system-defined threshold. When a hot peer is detected, replication is initiated. Instead of replicating the content of a single peer, HotRoD replicates *arcs of peers*, where an arc consists of successive neighbors that correspond to a certain range. This range is defined by the average low and high bounds of the range of the queries processed by the hot peer during the time interval T. Replication is achieved by using a multi-rotational hash function to randomly place the replicated arcs on the ring.

Sahin, Gupta, Agrawal & El Abbadi (2004) propose an extension of CAN for caching the results of range queries. In particular, the authors consider a 2-dimensionsal CAN. Each range query [low, high] is hashed at the point (low, high) in the virtual hash space.

Load Balancing

The *LAR protocol* (Gopalakrishnan, Silaghi, Bhattacharjee & Keleher, 2004) primarily addresses replication for load balancing of both the routing load as well as the load of the server holding the item and serving the request. Instead of creating replicas on all peers on a source-destination path as in path replication, the protocol relies on individual server load measurements to precisely choose replication points. The routing process is augmented with lightweight hints that shortcut the original routing and direct queries towards new replicas. Zhu, Zhang, Li & Huang (2007) propose a load prediction algorithm for estimating the load at each peer as well as multiple load thresholds for appropriately adjusting the number of replicas according to the load status of each node.

Alqaralleh, Wag, Zhou & Zomaya (2007) study three replica placements algorithms that can be implemented on top of any prefix-based DHT overlay. They were tested on top of FreePastry. The first algorithm, called CDN-QueryStat, places replicas on peers where queries frequently come from. In the second proposed algorithm, termed CDN-Rand, a peer randomly selects another peer from the id space. Thus, this algorithm tends to distribute replicas uniformly across the network. The third algorithm, CDN-PR, is a priority based approach that tries to minimize the number of peers that store replicas. The motivation is to re-

duce the overhead of maintaining load statistics. Peers are initially selected to hold replicas as in CDN-QueryStat. A new peer is chosen to hold replicas only if the previously selected peers get saturated with copies. Load balancing is applied, when the access frequency exceed a threshold. Then, a procedure is activated for replica creation and query forwarding.

Datta. Schmidt & Aberer (2007) propose using the query redundancy, that is, the existence of multiple search paths, to achieve better load balancing of both the routing load and the answering load of the server holding the item. They show through simulation, using the P-Grid topology, that, just replicating items and then routing to any of the replicas results in high statistical variation of the query and answering load. Proportional replication was used. To address this imbalance, they propose exploiting the redundant routing table entries used for fault tolerance. To choose among the available peers at each routing step, a cumulative load measure was used where answering queries weighted more than forwarding ones.

P2P-Based Storage and Caching

Dabek, Kaashoek, Karger, Moris & Stoica (2001) have proposed *CFS*, a storage layer based on a DHT that consists of two layers, namely the DHash (a distributed hash table) and Chord layers. The DHash layer performs block fetches for the client and distributes the blocks of each file among the servers. It uses the Chord distributed lookup system to locate the servers responsible for a block. CFS provides distinct mechanisms for replication and caching. Both caching and replication are performed at the level of a file block. CFS places the replica of a block at the *r* servers immediately after the successor of the block on the Chord ring. The placement of block replicas makes it easy for a client to select the replica likely to be the fastest to download. CFS also caches blocks to avoid overloading servers that hold popular data items. A block is cached at all peers on the search path after each successful look-up. Cached blocks are replaced in a least-recently-used order. This has the effect of preserving the cached copies close to the successor. In addition, it expands and contracts the degree of caching for each block according to its popularity.

Squirrel (Iyer, Rowstron & Druschel, 2002) is a decentralized p2p system that exploits resources from many desktop machines to achieve the functionality and performance of a dedicated web cache without requiring any additional hardware. Squirrel is built over the routing substrate of Pastry and uses its functionality for locating an object stored at the distributed client caches. It adopts two approaches to create copies: a home-store and a directory approach. In the home-store approach, Squirrel stores objects both at client caches and at their home peer. In the directory approach, the home peer remembers a small number of peers (up to k) that have recently accessed a certain object and keeps pointers to these peers. Then, each request is redirected to a randomly chosen peer among these (called the delegate), which is expected to have a copy of the object locally cached. Comparing these approaches in practice, the home-store method achieves better load balancing than the directory one, since popular objects are associated with rapidly changing directories.

Multiple Mappings

The *"power of two choices"* (Byers, Considine & Mitzenmacher, 2003) proposes a strategy for replica placement for Chord based on multiple hash functions. Each object is hashed using d ($d \geq 2$) hash functions to multiple ids and placed on the least loaded peer among candidates. To locate an item, instead of applying all d hash functions, the peers responsible for the item are connected with each other through *redirection pointers*. Using the redirection pointers, each request received by other peers (candidates) is redirected to the hosting peer. Although, using two or more choices for

placement improves load balancing, it still forces a static placement of the data items, which may lead to poor performance when the popularity of items changes over time. One way of addressing this issue is to use the re-direction pointers among the peers and allow items to choose a different peer for placing its replica by periodically re-inserting the items, if their previous choice has become more heavily loaded. The redirection pointers can also be used to facilitate a wide range of load balancing methods that react more quickly than periodic re-insertion, such as allowing an under-utilized peer to perform load-stealing or an overloaded one to attempt load shedding.

Symmetric Replication (Ghodsi, Alima & Haridi, 2005) can be applied to any structured peer-to-peer system. The basic idea is to associate each identifier in the system with *f* other identifiers. If identifier *i* is associated with identifier *r*, any item with identifier *i* should be stored at the peers responsible for identifiers *i* and *r*. Similarly, any item with identifier *r* should also be stored at the peers responsible for the identifiers *i* and *r*. Thus, effectively an identifier space of size *N* is partitioned into *N*/*f* equivalence classes such that identifiers in an equivalence class are all associated with each other. To replicate items with symmetric replication, the peer responsible for identifier *i* stores all *f* items with an identifier associated with *i.* For example, if the identifier 0 is associated with the identifiers 0, 5, 10, 15, any peer responsible for any of the items 0, 5, 10, or 15 has to store all of the items 0, 5, 10, and 15. Hence, if we are interested in retrieving item 0, we can ask the peer responsible for any of the items 0, 5, 10, 15. To implement symmetric replication, each peer in the system augments its routing table to contain for each routing entry *f* entries, one for each of the replicas of the routing entry. Symmetric replication can be used to send out multiple concurrent requests for an item and then picking the first response that arrives. The advantage of this is twofold. First, it enhances performance. Second, it provides fault tolerance in an end-to-end fashion, since the failure of a peer along the search path does not require repeating the request as it is likely that another one of the concurrent requests succeeds. It can also be used to achieve proximity neighbor selection in the following way. To route a message to the peer responsible for identifier *i*, each message in the routing process is augmented with a parameter *r* that specifies which of the *f* replicas of *i* is currently searched for. A peer that receives the request for a replica of *i* can calculate its distance to all of the *f* replicas and choose among the *f* peers the one that has a shorter distance to each respective replica of *i*. Then, it updates the parameter *r* in the outgoing message to reflect the new selection.

Locating Replicas

The *Replica Location Service (P-RLS)* (Cai, Chervenak & Frank, 2004) relies on Chord to build a mechanism for locating replicas. Each mapping from logical names (i.e., keys) to physical locations (i.e., replicas) is stored at the root peer of the mapping. P-RLS uses *successor replication*: the root peer replicates the mappings to its *k* successor peers in the Chord ring for successor routing reliability, where *k* is the replication factor. As a peer joins to network, it will take over some of the mappings and replicas from its successor peer. When a peer leaves the system, its predecessor will detect its departure, make another peer the new successor, and replicate mappings on the new successor peer adaptively. To avoid unnecessary replication of mappings, each mapping is associated with an expiration time. Besides fault tolerance, successor replication improves data load balance. In Chord, the number of mappings stored at each node is determined by the distance of the node to its immediate predecessor in the circular space, i.e. the "owned region" of the node. With adaptive replication with replication factor *k*, besides storing the nodes belonging to its own region, each node also replicates the mappings belonging to its *k* predecessors. Therefore,

the number of mappings stored on each node is determined by the sum of k+1 continuous owned regions before the node. If the node identifiers are generated randomly, there is no dependency among these continuous owned regions. Thus, intuitively, when the replication factor k increases, the sum of k+1 owned region is distributed more normally. To improve query load balance, P-RLS also proposes *predecessor replication*: replicating mappings at the predecessors of the root node. When a predecessor receives a query to the root node, it resolves it locally using its own replica of the mapping without forwarding the request to the root node, thus alleviating hotspots.

To reduce the number of replicas as well as the delay and bandwidth consumption for update propagation, Chen, Katz & Kubiatowicz, (2002) propose organizing the replicas on an application-level multicast tree, called replica dissemination tree (or d-tree) build on top of the overlay network, in their case, Tapestry. Each peer in the d-tree maintains state information only for its parent and its direct children. Two algorithms are proposed for dynamic replica placement. In the first algorithm, called naive placement, a peer stores a replica on the parent server of the requestor peer or on the overlay path server that is as close to the asked peer as possible. The second scheme, called smart placement, chooses as parent the peer with the lowest load among candidates. If more than one of them meets the requirements, then the replica is placed on the overlay path server that is as far from the requestor as possible.

Caching State

EpiChord (Leong, Liskov & Demaine, 2006) is a DHT similar to Chord. Instead of maintaining a finger table per node, EpiChord keeps a cache per node with a list of k successor and k predecessor node. Nodes populate their caches mainly from observing network look-up traffic, and cache entries are flushed from the cache after a fixed lifetime. In particular, each node updates its cache based on information returned by queries and adds an entry to the cache each time it is queried by a node not already in the cache. To lookup an entry, an EpiChord node initiates a number of parallel lookups to the successors and predecessors nodes in its cache. In addition, nodes communicate with their immediate successor and predecessor periodically, exchanging their entire successor and predecessor lists.

Summary of Replication in Structured P2P Systems

Approaches to replication differ on *what* is replicated. Replication may involve either replicating the item itself or its index (i.e. its location). In few cases, the routing table or information about neighbors is also replicated.

There are various methods for achieving replication in structured p2p systems. A very common approach is to place a number of replicas at the immediate neighbors of a node such as the successor nodes in CHORD, the nodes in the leafset in PAST or the peers at the neighboring zones in CAN. Such replicas are easily locatable. The primary reason for this form of replication is fault tolerance. Another approach is to use multiple hash functions to map an item to more than one node. Besides availability, applying multiple hash functions allows the employment of multiple search paths for an item and thus improves query latency and path fault tolerance. Path or owner replication can also be used to improve search for popular items. Finally, to achieve load balancing various approaches base their decision to create replicas purely on the load of each peer. Other approaches include: making more than one node responsible for the same identifier space (such as with zone overloading in CAN or structural relaxation in P-Grid), building multiple overlays (such as with multiple realities in CAN) or building a replica tree on top of the overlay (such as in d-tree).

The number of replicas created is either fixed for all items as a general replication factor of the

system (for instance *k* successors or nearest peers in the identifier space) or varies for each item or node depending on the current system parameters such as the item popularity or the load at the servers.

Table 2 summarizes the various replication approaches in structured p2p networks.

Table 2. Summary of replication strategies in structured p2p systems

	How many	Where/How	What	Goal
Applications built on top of Chord (Stoica et al, 2001)	*k* successors	Multiple hash functions, or At the successors of an item	Data items or keys	Failure recovery Load balance
CAN (Ratnasamy et al, 2001)	varies	Neighbor replication Multiple realities Multiple hash functions Zone overloading Caching (i.e. owner replication)	Data items/index entries	Response time Availability Neighbor selection Load balancing
PAST (Rowstron et al, 2001b)	*k* nearest identifiers	At the *k* peers whose identifier is numerically closest to the identifier of the file	Files	Fault tolerance
		Caching (path replication)	Files	Query load balance
Kademlia (Maymounkov & Mazieres, 2002)	*k* replicas plus 1 new per successful lookup	*k* closest nodes to the key and 1 to next-to--last node on the lookup path	<key, value> pairs	Handle failures and improve latency
P-Grid (Aberer et al, 2005)	-	Multiple peers per key space Multiple route paths	Data keys Routing	Load balancing Fault tolerance
Beehive (Ramasubramanian et al, 2004)	*k* per item where *k* depends on item popularity	At all peers *k* hops before the successor of the item	Data items	Achieves lookup of a constant number of *C* hops
PopCache (Rao et al, 2007)	Such that to achieve optimal average search (MAX PERF) or a targeted lookup threshold (MIN COST)	On the *k*-tree induced by the *k* neighbors of each node	Data items	Query latency
HotRoD (Pitoura et al, 2006)	Arcs of peers	Multi-rotational locality-preserving hash function	Popular data items on arcs of peers	Load balance for range queries
LAR {Gopalakrishnan et al, 2004)	adaptive	Load measurements by individual peers	Data items/index entries	Data and query balance
CDN (Alqaralleh et al, 2007)	adaptive	Frequently query peers, or at random peers or such that to minimize the number of peers holding replicas	Data items	Performance Load balancing Replica maintenance cost
CFS (Dabek et al, 2001)	varies	Caching Replication at the successor	File Blocks	Load balancing Performance
Squirrel (Iyer et al, 2002)	up to *k* pointers	Caching	Data items (home-store) or Pointers to their location (directory)	Query latency Fault tolerance
Power of two choices (Byers et al, 2002)	*d* hash functions	Multiple hash functions	Data items	Load balance
Symmetric Replication (Ghodsi et al, 2005)	*f* nodes	Equivalence classes of related identifiers	Data items	Response time Neighbor selection Fault tolerance
P-RLS (Cai et al, 2004)	*k* successors and /or predecessors	At the successors	Mappings	Failure recovery, Data balance
		At the predecessors		Query balance
d_tree (Chen et al, 2002)	varies	On a multicast tree built on top of the p2p overlay	Data items	Reduce storage Query latency Improve update efficiency
EpiChord (Leong et al, 2006)	varies	Caching during lookup	Routing state (predecessor and successors)	Query latency Reduce state per node

UPDATES

Replication introduces the overhead of maintaining the replicas of each data item up-to-date. A replica management protocol decides *where* (i.e. at which copies) updates take place, *when* updates propagate to other replicas and *how* the propagation of updates is achieved.

According to the *where* aspect, replication strategies can be classified broadly as single master or primary copy and multi-master or group. *Single master or primary copy* replication is the simplest approach in which each replicated item is owned by a single peer (or owner). The copy held by the owner is called the primary copy. All copies can be read but any update to an item must be first applied to its primary copy and then propagated to the other copies. The advantage of primary copy replication is its simplicity. However, the owner of an item may be a potential bottleneck as well as a single point of failure. The *multi-master* or *group approach* allows multiple peers to hold primary copies of the same data item. All replicas are regarded as equally authoritative. The multi-master approach avoids bottlenecks and single points of failures, however, it increases communication costs and system complexity, since it requires concurrent updates at different replicas to be coordinated and reconciled to solve any potential replica divergences.

In terms of the *when* aspect, update propagation strategies can be implemented either synchronously or asynchronously. A synchronous propagation mechanism updates all replicas before a transaction commits. With the asynchronous strategy, only a subset of the replicas is updated.

Regarding the *how* aspect, most replication management techniques in p2p networks use a combination of push and pull methods to propagate updates as follows. The initiator of an update *pushes* the new value of its copy to a number of other peers in the system. A peer that holds a copy *pulls* other peers to be informed of any potential updates. Most update propagation mechanisms in p2p systems are *probabilistic* in the sense that they ensure that an update will eventually reach all copies of an item with a certain probability. The propagation of an update may involve a *notification* that the item has been updated, a *state transfer* where the actual new value of the modified data is transferred or an *operation transfer* where the update operation is propagated. Choosing a propagation method depends on the amount of data, bandwidth availability and various system- and application- related characteristics.

Finally, the consistency of replicas refers to the allowable divergence among the various copies of an item. *Strong* consistency does not allow any such divergence and guarantees that each read returns the most current value of an item. *Weak* consistency allows various levels of divergence among copies as well as reads that may return stale values of an item.

To address scalability and dynamicity, most update replication mechanisms in p2p systems support multi-master schemes, probabilistic update propagation and weak consistency.

Individual Peer Techniques

Before proceeding to describe update mechanisms for unstructured and structured p2p systems, we review some techniques that can be followed by individual peers in order to achieve a desired level of confidence or consistency for the items they are interested in.

Vecchio & Son (2005) adapt the traditional quorum consensus schemes to a dynamic p2p environment by letting each peer choose its own quorum level. In effect, each peer decides on the level of confidence of the item it accesses. This way, there is a tradeoff between the incurred message overhead and the achieved consistency levels; the higher the quorum values the higher the message overhead and the lower the possibility of accessing stale data.

The Controlled Update Propagation (CUP) protocol (Roussopoulos & Baker, 2003) allows

individual peers to receive and propagate updates only when they have a payoff to do so. Each peer registers with its neighbors for receiving updates only for the items it is interested in. Correspondingly, it propagates any received updates of an item *i* only to the neighbors that have registered their interest for *i*. A node decides whether it is interested in receiving updates for item *i* based on the "profit" it will have; receiving an update is justified if it will save the node the cost of handling queries, i.e. if the node receives frequently queries for item *i* then keeping an up-to-date replica of *i* will allow it to answer these queries immediately, avoiding the overhead of propagating the queries further. Clearly, these policies favor the popular items since these items generate queries most often.

Susarla & Carter (2005) let each peer express their consistency requirements as a vector of options along five different dimensions, on a per-access basis. They argue that different classes of distributed applications, such as file access, database and directory services, and real-time collaborative groupware, have a broad and diverse set of requirements with regards to replica handling. These requirements are classified along the following five, mostly orthogonal, dimensions: (1) concurrency - the degree to which conflicting accesses can be allowed, (2) replica synchronization - the degree to which replica divergence can be tolerated (termed coherence or timeliness) and the types of inter-dependencies among updates that must be preserved upon replica synchronization (termed consistency), (3) failure handling - how data access is handled when some replicas become unreachable or have poor connectivity (4) update visibility - the time at which modifications to local data are made visible globally, and (5) view isolation - the time at which remote updates are made visible locally. To cater for such diverse requirements with regards to replica updates, the *composable consistency model* is proposed along with an outline of its implementation in *Swarm*, a wide area p2p middleware file service. Swarm allows applications to specify the level for each of the five requirements at every search. Swarm assumes that there is a master server, termed custodian, per file that coordinates the consistency management protocols for the file. There can be more than one custodian per file for fault tolerance. Consistency is achieved through a combination of push and pull operations.

Updates in Unstructured P2P Networks

As mentioned above, the consistency mechanisms that have been proposed use a push-based and/or a pull-based propagation algorithm. One more possibility can be found in the work of Demers et al (1987), who have applied the theory of epidemics to the problem of update propagation in a distributed environment, proposing a number of generic methods. The first method the authors examine is *direct mail*, where the owner of a data item contacts (`mails') all the other peers at every update. This approach, although simple, can be overwhelming in a p2p network with a large number of nodes. In the *anti-entropy* method each peer regularly chooses a neighbor and by exchanging their content resolves any differences between them (if a newer version of an item is found, it updates its own replica). A peer can either push its content to the other peer letting it check for inconsistencies, or pull content, or even push and pull content at the same time. Another update spreading algorithm considered is *rumor mongering*: at first all peers are considered 'ignorant' when an update is out and the update becomes a 'hot rumor'. If a peer knows of such a rumor, it periodically chooses another peer and tries to communicate the rumor. If the peer sees that the rumor is no longer hot (i.e. most of the peers it contacts already know it), it stops propagating it any further.

If the direct mail method is to be used, a natural plan would be to know (most of) the peers that hold a replica of the particular data item (statefull replication) so as to only contact those upon an

update. A mechanism like this is assumed by Datta, Hauswirth & Aberer (2003). The authors study the performance of a generic hybrid push-pull consistency maintenance protocol for p2p environments where peers join and leave the network at a very high rate. At the push phase, the owner sends the updated item, along with its version number, to the peers that hold replicas. This requires knowledge of who holds replicas of what, but the update is not communicated through direct mail; it is rather propagated with a randomized flooding among the affected peers. The owner performs a selective push of its updates to a subset of the peers that will be affected by it (because they have a replica of the updated data item); each peer that receives the update also propagates it to a subset of affected peers it knows, and so on. To reduce the overhead, each message contains a partial list of the peers that have already been contacted. The method is accompanied by a pull phase that takes place whenever a peer is reconnected to the network after a disconnection or has not received updates for a long time (in the spirit of the anti-entropy method); during this pull phase, it contacts online peers with replicas of the items it stores, for their latest versions.

UPTReC – update propagation thought replica chain (Wang, Das, Kumar & Shen, 2007) – exploits similar pull and push mechanisms to scatter updates in decentralized and unstructured p2p systems. The peers that hold the replicas of an item i form a logical bi-directional chain, where each peer maintains information about the k closest peers in the chain in each direction. Peers may join (when a new replica is created) or leave (when removing a replica) by pushing messages at appropriate directions in the chain. Updates are similarly propagated by pushing messages at both directions, informing up to $2k$ nodes; at each direction the furthest known peer undertakes the responsibility of reaching the next bunch of k nodes in the chain and so on. Nodes that reconnect after a disconnection pull in order to synchronize. Maintaining such a chain for every item reduces the message overhead on updates while also providing better consistency levels than Datta, Hauswirth & Aberer (2003), as shown experimentally.

Wang, Kumar, Das & Shen (2006) consider multi-master replication where all replica holders (termed "replica peers" – RPs) are allowed to update the item. In particular, a subset of RPs become "virtual servers" (VRPs) for the data item. The set of VRPs changes dynamically over time, based on node availability. Any replica peer updating the item contacts a VRP to undertake the update coordination. This "master" VRP first enters an agreement phase with the other VRPs in order to commit the update. When agreement is achieved, the master VRP obtains the updated item from the replica peer and pushes it to the remaining VRPs and to a partial list of the other RPs. Among the other RPs, the update propagation is implemented using a combination of push and pull, where some RPs are only pushing while the others are only pulling. The protocol achieves one-copy serializability, i.e. the concurrent execution of updates on a replicated item has the same effect as a serial execution on a non-replicated item.

Update propagation in the last three methods occurs strictly among the interested peers; although this seems efficient in terms of overheads and consistency levels, it nevertheless incurs the extra state overhead of keeping track of all peers holding a replica of the data item, which could be prohibitive in an unstructured and dynamic p2p network. Three update propagation policies (two based on push and pull techniques and a hybrid one that combines the push and pull policies) are proposed by Lan, Liu, Shenoy & Ramamritham (2003) for practical networks. The authors assume a master-copy schema where the owner of the data item always has the most up to date version and all peers that hold a replica need to be kept consistent; the overlay network is unstructured and the owners do not know who/where replica holders are. To achieve consistency, each data item is associated with a version number which

is incremented by the owner every time an update occurs. In the push-based policy, the owner of a data item broadcasts an invalidation message when a data item is modified. The invalidation message is propagated through the network using a flooding algorithm, limited to a predefined number of hops (TTL). When a peer receives an invalidation message, it checks its cache. If it holds a replica of the data item and the stored version is smaller than the received version number, it invalidates the replica in its cache. In the proposed pull-based policy, a peer polls the owner of an item it holds in its cache to determine if the replica is stale or not. An *adaptive polling policy* is used to determine how frequently the peer should poll. It is based on a time-to-refresh (TTR) value associated with each item in the cache, which indicates when the next pull for the item should occur. The TTR is increased by an additive amount C (TTR = TTR+C) if the peer finds out that a data item has not been modified between two successive polls, otherwise TTR is reduced by a multiplicative factor D (TTR = TTR/D). A hybrid push and pull approach can also be used to combine both techniques. In this hybrid scheme, the owner propagates invalidation messages using a limited push. In addition, a peer that holds a replica may pull adaptively to make sure that the replica is valid. TTR can be further tuned by a factor that depends on the degree of a peer; the intuition behind this is that highly connected nodes should poll less frequently since they are potentially easier to reach by the owner push.

An alternative hybrid push/pull update propagation policy, *PtPU*, is proposed by Leontiadis, Dimakopoulos & Pitoura (2006). It is assumed that for the creation of replicas in the p2p network the Pull-then-Push algorithm was used where a peer that requests an item, after a successful search (pull phase) enters a push phase where it transmits replicas of the item using the same algorithm as in the pull phase. Given this replica creation approach, each peer that holds a data item is characterized as *owner* if it is allowed to apply updates, *responsible* if it has requested the data item and has forced the creation of replicas or *indifferent* if it has been forced to hold a replica without requesting the data item. In the PtPU policy, the owner performs a limited broadcast of the new version of a data item when an update occurs. If a peer that is characterized as responsible for an item receives the broadcast message with a new version of the data item, it undertakes the task of informing the indifferent peers. This is done by propagating the update message (*U-push phase*) exactly as in the push phase when the replicas were created. Apart from pushing the updates they receive from the owner, responsible peers also pull periodically in order to become aware of more updates. To determine the frequency of the pull, the adaptive polling policy is used, where a TTR value is increased or decreased depending on weather the data item has been changed or not between two successive poll periods.

Updates in Structured P2P Networks

Many update propagation mechanisms in p2p systems use a form of periodic pushing to inform of any updates other holders of data copies. This is particularly useful in terms of updates related to state or routing information. This is often referred to as *soft-state* updates. In P-RLS (Cai, et al, 2004) update propagation is implemented in two phases. In the first phase, the Replica Location Service (LRC) periodically sends soft state updates summarizing its state into the peer-to-peer network. Then, the root peer of each mapping updates its successors immediately to maintain the consistency of the replicated mappings. Soft-state updates are also applied in LAR (Gopalakrishnan et al, 2004) and CAN (Ratnasamy et al, 2001). In CAN, each peer sends periodic update messages to each of its neighbors giving information about its zone coordinates and a list of its neighbors with their zone coordinates. The same policy of periodic update messages is applied to CDN (Alqralleh et al, 2001).

Beehive (Ramasubramanian et al, 2004) exploits the structure of the underlying DHT to provide strong consistency. It ensures that any object modification is propagated to all replicas. In CFS (Dabek et al, 2001) cryptographic verification of updates and server id authentication are used and only owners of data can implement updates. For update dissemination in symmetric replication (Chen et al, 2002), replicas and caches self-organize into a *d*-tree and use application-level multicast to propagate updates. Replicas and caches are always kept up-to-date. P-Grid (Aberer et al, 2003) proposes an update mechanism based on a generic push/pull gossiping scheme that provides probabilistic guarantees for consistency.

Akbarinia, Pacitti & Valduriez (2007) proposed an interesting replica update mechanism for DHT-based p2p networks. Their objective is to provide a mechanism which returns efficiently a current replica of a data item given its key. The proposed update management mechanism relies on timestamps. Data items are replicated using multiple hash functions as in many structured p2p systems. The main difference is that, when a data item is mapped to a peer, each item is associated with a logical timestamp which is stored along with the item. Timestamps are generated through a distributed service that guarantees the monotonicity property for timestamps, i.e. two timestamps generated for the same key are monotonically increasing. This property allows ordering the timestamps generated for the same key according to the time at which they have been generated.

The distributed timestamp generation service uses the underlying DHT. In particular, a hash function is used to map each key with one peer that is held responsible for returning a new timestamp for that key. Each peer that needs a timestamp for an item with a specific key *i* uses the hash function to locate the peer responsible for generating timestamps for *i* and sends a timestamp request to it. Upon receiving the request, the responsible peer initializes some local counter to the value of the last generated timestamp for *i*.

Upon storing an item with timestamp t_s, in case the peer already has a replica of the item with timestamp t_p, the two timestamps are compared so that only the latest version (the one with the largest timestamp) is finally kept. In order to retrieve a data item, a peer first gets a timestamp from the responsible peer and compares it with the results it receives, so that it ensures its currency.

SUMMARY

In this chapter, we have presented replication techniques and mechanisms that have been proposed for p2p networks. Replication is a central mechanism for improving performance and availability in a distributed system. P2p systems introduce new challenges mainly because of their unprecedented scalability and dynamicity. In this chapter, we have focused on replication mainly for improving the response time of search and achieving load balancing.

Note that replication is just one method for achieving redundancy. Alternatively, erasure coding or a combination of replication and erasure coding can be used towards this end. An *erasure code* provides redundancy without the overhead of strict replication. Erasure codes divide an object into *m* fragments and recode them into *n* fragments, where $n > m$. The ratio m/n is called the *rate* of encoding. A rate *r* code increases the storage cost by a factor of $1/r$. The key property of erasure codes is that the original object can be reconstructed from any *m* fragments. Weatherspoon & Kubiatowicz (2002) have quantified the availability gained using erasure codes. Then, they show that erasure-resilient codes use an order of magnitude less bandwidth and storage than replication for systems with similar mean time to failure (MTTF). They also show that employing erasure-resilient codes increase the MTTF of the system by many orders of magnitude over simple replication with the same storage overhead and repair times. Recent research takes other issues

into consideration such as user download behavior (Chen, Qiu & Wu, 2008) and the characteristics of the overlay nodes (Rodrigues & Liskow, 2005) under which replication may outperform erasure codes. Erasure codes and related protocols are beyond the scope of this chapter, since our main focus is on search quality and load balance.

In unstructured p2p systems, most research has focused on determining the appropriate number of replicas as well as on developing practical mechanisms for placing the replicas. Most of the theoretical work on the subject is based on the results of Cohen & Shenker (2002) and assumes a network topology and a search strategy that allow uniform node sampling. Similar results are lacking for other search strategies and more realistic network models (e.g. power-law random graphs). A very limited number of practical algorithms have been presented for the placement of replicas so as to optimize some aspects of search performance. Replication in unstructured p2p systems is currently an area where further theoretical as well as experimental analysis is needed.

As with unstructured p2p, the main reasons for replication in structured p2p systems are availability and performance. In particular with structured p2p, replication is central in improving load balancing caused by skew in the mapping of items to nodes. Replica placement decisions in structured p2p systems often exploit the structure of the underlying overlay. Common choices for placing replicas include the immediate neighboring nodes as well as the nodes on the search path of an item. DHTs also offer alternative mechanisms for realizing replication that are based on the mapping of the data-key space to nodes. One way is by using multiple hash functions to map (store) the same item on multiple nodes. Another way is by assigning the same key space to more than one node (such as with zone overloading in CAN or structural relaxation in P-Grid). Finally, one may build multiple overlays (such as with multiple realities in CAN) or replica trees on top of the overlay (such as in d-tree).

Once replicas are created, a central point is maintaining the replicas up-to-date. Strategies that are based on global knowledge of the replica holders may have the potential to achieve good consistency levels and low message loads, but seem rather inappropriate for dynamic networks that evolve quickly. A logical thing to do in such a case is exploit the p2p network structure for an efficient update scheme, an approach that by nature fits some structured network topologies well. For all other networks most approaches aim at achieving some form of probabilistic consistency, relying on a combination of pushing the updates to neighbors and pulling copies from them.

Since no single optimal solution exists for replica placement or for replica updates, this is expected to be an active area of research for many years. Promising issues include (a) theoretical results regarding the optimal replica placement in various types of unstructured p2p systems, (b) gossiping protocols for update maintenance, (c) replication for fault tolerance, (d) replication to meet the requirements of specific storage-related applications and (e) replication techniques for highly dynamic and unpredictable environments. New research challenges also arise in the area of social networking and in mobile peer-to-peer environments.

REFERENCES

Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., & Schmidt, R. (2003). P-Grid: A Self-organizing Structured p2p System. *SIGMOD Record*, *32*(3).

Aberer, K., Datta, A., & Hauswirth, M. (2003). *The quest for balancing peer load in structured peer-to-peer systems* (Tech. Rep. EPFL No. IC/2003/32). Lausanne, Switzerland: Ecole Polytechnique Fédérale de Lausanne.

Akbarinia, R., Pacitti, E., & Valduriez, P. (2007). Data currency in replicated DHTs. In *Proc. SIGMOD 2007 ACM Int'l Conference on Management of Data,* Beijing, China (pp. 211-222).

Alqaralleh, B. A., Wang, C., Zhou, B. B., & Zomaya, A. Y. (2007). Effects of replica placement algorithms on performance of structured overlay networks. In *Proc. IPDPS 2007, IEEE Int'l Parallel & Distributed Processing Symposium,* Long Beach, CA, USA (pp. 1-8).

Byers, J., Considine, J., & Mitzenmacher, M. (2003). Simple load balancing for distributed hash tables. In *Proc. IPTPS 2003, 2nd Int'l Workshop on Peer-to-Peer Systems*, Berkeley, CA, USA.

Cai, M., Chervenak, A., & Frank, M. (2004). A peer-to-peer replica location service based on a distributed hash table. In *Proc. SC2004, ACM/IEEE Conference on Supercomputing,* Pittsburgh, Pennsylvania, USA (pp. 54-54).

Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., & Shenker, S. (2003). Making gnutella-like P2P systems scalable. In *Proc. of SIGCOMM'03,* Karlsruhe, Germany (pp. 407-418).

Chen, G., Qiu, T., & Wu, F. (2008). Insight into redundancy schemes in DHTs. *The Journal of Supercomputing*, *43*, 183–198. doi:10.1007/s11227-007-0126-4

Chen, Y., Katz, R. H., & Kubiatowicz, J. (2002). Dynamic replica placement for scalable content delivery. In *Proc. IPTPS 2002, 1st Int'l Workshop on Peer-to-Peer Systems* (pp. 306-318). Boston, MA, USA.

Clark, I., Miller, S. G., Hong, T. W., Sandberg, O., & Wiley, B. (2002). Protecting free expression online with Freenet. *IEEE Internet Computing*, *6*(1), 40–49. doi:10.1109/4236.978368

Cohen, E., & Shenker, S. (2002). Replication strategies in unstructured peer-to-peer networks. In *Proc. of SIGCOMM'02,* Pittsburgh, Pennsylvania, USA (pp. 177-190).

Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., & Stoica, I. (2001). Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symposium on Operating Systems Principles,* Chateau Lake Louise, Banff, Canada (pp. 202-215).

Datta, A., Heuswirth, H., & Aberer, K. (2003, May). Updates in highly unreliable, replicated peer-to-peer systems. In *Proc. of ICDCS 2003, 23rd Int'l Conference on Distributed Computing Systems,* Providence, Rhode Island, (pp. 76-85).

Datta, A., Schmidt, H., & Aberer, K. (2007). Query-load balancing in structured overlays. In *Proc. of CCGrid'07, 7th Int'l Conference on Cluster Computing and the Grid,* Rio de Janeiro, Brazil (pp. 453-460).

Demers, A., Green, D., Hauser, C., Irish, W., Larson, J., Shenker, S., et al. (1987). Epidemic algorithms for replicated database maintenance. In *Proc. PODC 1987, 6th Annual ACM Symposium on Principles of Distributed Computing,* Vancouver, Canada (pp. 1-12).

Ghodsi, A., Alima, L. O., & Haridi, S. (2005). Symmetric replication for structured peer-to-peer systems. In *Proc. DBISp2p'05, 3rd Int'l VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing* (LNCS 4125, pp. 74-85). Berlin: Springer.

Gnutella. (2003). *Protocol V.0.6 RFC*. Retrieved from http://rfc-gnutella.sourceforge.net

Gopalakrishnan, V., Silaghi, B., Bhattacharjee, B., & Keleher, P. (2004). Adaptive replication in peer-to-peer systems. In *Proc. ICDCS 2004, 24th Int'l Conference on Distributed Computing Systems,* Tokyo, Japan (pp. 360-369).

Iyer, S., Rowstron, A., & Druschel, P. (2002). Squirrel: a decentralized peer-to-peer web cache. In *Proc. PODC 2002, 21st Annual Symposium on Principles of Distributed Computing,* Monterey, California, USA (pp 213-222).

Jia, Z., Pei, B., Li, M., & You, J. (2005). A comparison of spread methods in unstructured P2P networks. In *Proc. of ICCSA 2005, Int'l Conference on Computational Science and its Applications* Singapore (pp. 10-18).

Lan, J., Liu, X., Shenoy, P., & Ramamritham, K. (2003). Consistency Maintenance in Peer-to-peer File Sharing Networks. In *Proc. of WIAPP'03, 3rd IEEE Workshop On Internet Applications,* San Jose, CA, USA (pp. 76-85).

Leong, B., Liskov, B., & Demaine, E. D. (2006). EpiChord: Parallelizing the chord lookup algorithm with reactive routing state management. *Computer Communications*, *29*(9), 1243–1259. doi:10.1016/j.comcom.2005.10.002

Leontiadis, E., Dimakopoulos, V. V., & Pitoura, E. (2006). Creating and maintaining replicas in unstructured peer-to-peer systems. In *Proc. of EURO-PAR 2006, 12th Int'l Euro-Par Conference on Parallel Processing* (LNCS 4128, pp. 1015-102). Dresden, Germany: Springer.

Lv, Q., Cao, P., Cohen, E., Li, K., & Shenker, S. (2002). Search and replication in unstructured peer-to-peer networks. In *Proc. of ICS 2002, 16th ACM Int'l Conference on Supercomputing,* New York, New York, USA (pp. 84-95).

Maymounkov, P., & Mazieres, D. (2002), Kademlia: A peer to peer information system based on the XOR metric. In *Proc. of IPTPS 2002, 1st Int'l Workshop on Peer-to-Peer Systems*, Cambridge MA, USA.

Morselli, R., Bhattacharjee, B., Marsh, M. A., & Srinivasan, A. (2005). Efficient lookup on unstructured topologies. In *Proc. PODC 2005, 24th Symposium on Principles of Distributed Computing,* Las Vegas, NV, USA.

Pitoura, T., Ntarmos, N., & Triantafillou, P. (2006). Replication, load balancing and efficient range query processing in DHTs. In *Proc. EDBT 2006,* Munich, Germany (pp. 131-148).

Ramasubramanian, V., & Sirer, E. G. (2004). Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. NSDI'04, 1st Symposium on Networked Systems Design and Implementation*, San Francisco, CA.

Rao, W., Chen, L., Fu, A., & Bu, Y. Y. (2007). Optimal Proactive Caching in Peer-to-peer Network: Analysis and Application. In *Proc. CIKM 2007, 15th ACM Int'l Conference on Information and Knowledge Management,* Lisbon, Portugal (pp. 663-672).

Ratnasamy, S., Francis, P., Handley, M., Karp, R., & Shenker, S. (2001). A scalable content addressable network. In *Proc. of ACM SIGCOMM,* San Diego, CA, USA (pp. 161-172).

Rodrigues, R., & Liskow, B. (2005). High Availability in DHTs: Erasure Coding vs. Replication. In *Proc. IPTPS 2005,* Ithaca, NY, USA (pp. 226-239).

Roussopoulos, M., & Baker, M. (2003). CUP: Controlled update propagation in peer-to-peer networks. In *Proc. of the Annual USENIX Technical Conference,* San Antonio, Texas, USA (pp. 167-180).

Rowstron, A., & Druschel, P. (2001a). Pastry: Scalable, distributed, object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware 2001, IFIP/ACM Int. Conf. on Distributed System Platforms,* Heidelberg, Germany (pp. 329–350).

Rowstron, A., & Druschel, P. (2001b). Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP'01,* Banff, Canada (pp. 188-201).

Sahin, O. D., Gupta, A., Agrawal, D., & El Abbadi, A. (2004). A Peer-to-peer Framework for Caching Range Queries. In *Proc. ICDE 2004, 20th Int'l Conference on Data Engineering,* Boston, USA (pp. 165 -176).

Sozio, M., Neumann, T., & Weikum, G. (2008). Near-Optimal Dynamic Replication in Unstructured Peer-to-Peer Networks. In *Proc. PODS'08, 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems,* Vancouver, BC, Canada (pp. 281-290).

Stoica, I., Morris, R., Karger, D., Kaashoek, F., & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM,* San Diego, CA, USA (pp.149–160).

Susarla, S., & Carter, J. (2005). Flexible Consistency for Wide Area Peer Replication. In *Proc. ICDCS 2005, 25th IEEE Int'l Conference on Distributed Computing Systems,* Ohio, USA (pp. 199-208).

Tewari, S., & Kleinrock, L. (2005). Analysis of search and replication in unstructured peer-to-peer networks. In *Proc. of SIGMETRICS 2005,* Banff, Canada (pp 404-405).

Tewari, S., & Kleinrock, L. (2006). Proportional replication in peer-to-peer networks. In *Proc. of INFOCOM 2006,* Barcelona, Spain (pp 1-12).

Vecchio, D., & Son, S. H. (2005). Flexible update management in peer-to-peer database systems. In *Proc. IDEAS 2005, Int'l Database Engineering and Applications Symposium*, Montreal, Canada.

Wang, Z., Das, S. K., Kumar, M., & Shen, H. (2007). An efficient update propagation algorithm for p2p systems. *Computer Communications*, *30*, 1106–1115. doi:10.1016/j.comcom.2006.11.005

Wang, Z., Kumar, M., Das, S. K., & Shen, H. (2006). File consistency maintenance through virtual servers in P2P systems. In *Proc. ISCC 2006, 11th IEEE Symposium on Computers and Communications,* Sardinia, Italy (pp. 435-441).

Weatherspoon, H., & Kubiatowicz, J. (2002). Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS 2002,* Cambridge, MA, USA (pp 328-338).

Zhang, H., Goel, A., & Govindan, R. (2004). Using the small-world model to improve Freenet performance. *Computer Networks*, *46*, 555–574. doi:10.1016/j.comnet.2004.06.003

Zhu, X., Zhang, D., Li, W., & Huang, K. (2007). Prediction-based fair replication algorithm in structured p2p Systems. In *Proc. ATC 2007, 4th Int'l Conference on Autonomic and Trusted Computing,* Hong Kong, China (pp. 499-508).

KEY TERMS AND DEFINITIONS

Data Replication: Refers to creating and maintaining multiple copies of an item so as to improve performance and reliability.

Overlay Networks: Networks formed between nodes in large scale distributed systems which are built on top of the physical network.

Peer-to-Peer (P2P) Systems: Distributed systems where nodes act as both servers and clients. Characteristics commonly attributed to peer-to-peer systems include node autonomy, large scale and dynamicity.

Replica Placement: Refers to protocols for assigning replicas to nodes in a distributed system.

Replica Updates: Refers to protocols used to maintain consistency among replicas.

Structured Peer-to-Peer Systems: P2P systems where nodes are connected to each other to form specific overlay topologies. The most common structured p2p systems are Distributed Hash Tables (DHTs) where data items are assigned to specific peers based on hashing.

Unstructured Peer-to-Peer Systems: P2P systems where the overlay topology is not rigid and there is no explicit association between the location of data and the location of nodes. Many unstructured p2p systems have power-law degree characteristics.

ENDNOTE

[1] We note here that while both refer to creating copies, caching and replication have some subtle differences. Caching is usually initiated at the clients, in our case, the peers that made the request for an item, while replication is a server-based decision, with possibly system-wide implications. In this chapter, we will not distinguish between replication and caching and we will use the term 'replication' to refer to both of them.