Survey of data replication in P2P systems

Vidal Martins, Esther Pacitti, Patrick Valduriez

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Survey of data replication in P2P systems

Vidal Martins, Esther Pacitti, Patrick Valduriez

N° ????

Décembre 2006

Thème GDD

ISSN 0249-6399

Survey of data replication in P2P systems

Vidal Martins[1], Esther Pacitti[2], Patrick Valduriez[3]

Thème GDD – Gestion de Données Distribuées
Projet Atlas

Rapport de recherche n° ???? – Décembre 2006 - 45 pages

Abstract: Large-scale distributed collaborative applications are getting common as a result of rapid progress in distributed technologies (grid, peer-to-peer, and mobile computing). Peer-to-peer (P2P) systems are particularly interesting for collaborative applications as they can scale without the need for powerful servers. In P2P systems, data storage and processing are distributed across autonomous peers, which can join and leave the network at any time. To provide high data availability in spite of such dynamic behavior, P2P systems rely on data replication. Some replication approaches assume static, read-only data (e.g. music files). Other solutions deal with updates, but they simplify replica management by assuming no update conflicts or single-master replication (i.e. only one copy of the replicated data accepts write operations). P2P advanced applications, which must deal with semantically rich data (e.g. XML documents, relational tables, etc.) using a high-level SQL-like query language, are likely to need more sophisticated capabilities such as multi-master replication (i.e. all replicas accept write operations) and update conflict resolution. These issues are addressed by optimistic replication. Optimistic replication allows asynchronous updating of replicas so that applications can progress even though some nodes are disconnected or have failed. As a result, users can collaborate asynchronously. However, concurrent updates may cause replica divergence and conflicts, which should be reconciled. In this survey, we present an overview of data replication, focusing on the optimistic approach that provides good properties for dynamic environments. We also introduce P2P systems and the replication solutions they implement. In particular, we show that current P2P systems do not provide eventual consistency among replicas in the presence of updates, apart from APPA system, a P2P data management system that we are building.

Keywords: data replication, reconciliation, eventual consistency, P2P networks

[1] LINA, University of Nantes – Vidal.Martins@univ-nantes.fr
[2] LINA, University of Nantes – Esther.Pacitti@univ-nantes.fr
[3] INRIA Rennes – Patrick.Valduriez@inria.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie +33 2 99 84 71 71

Survey of data replication in P2P systems

Résumé: Des applications collaboratives en large échelle deviennent de plus en plus populaires en raison du développement rapide des technologies reparties (grid, pair-à-pair, computation mobile). Les systèmes pair-à-pair (P2P) sont particulièrement intéressants pour les applications collaboratives car ils peuvent passer à l'échelle sans avoir besoin de serveurs puissants. Dans les systèmes P2P, le stockage et le traitement de données sont repartis à travers des pairs autonomes qui peuvent joindre ou quitter le réseau à tout moment. Pour assurer la haute disponibilité des données malgré ce comportement dynamique, les systèmes P2P comptent sur la réplication de données. Certaines approches de réplication supposent des données statiques, lecture seule (par exemple fichiers de musique). D'autres solutions prennent en compte les mises-à-jour, mais elles simplifient la gestion des réplicas en supposant l'absence de conflits de mise-à-jour ou la réplication mono-maître (c.-à-d. seulement une copie de la donné répliquée accepte des opérations de mise-à-jour). Des applications P2P avancées, qui doivent traiter des données sémantiquement riches (par exemple des documents XML, tables relationnelles, etc.) en utilisant un langage de requête de haut niveau similaire à SQL, probablement ont besoin de capacités plus sophistiquées comme réplication multi-maître (c.-à-d. toutes les réplicas acceptent des opérations de mise-à-jour) et résolution de conflit de mise-à-jour. Ces sujets sont adressés par la réplication optimiste. La réplication optimiste permet la mise-à-jour asynchrone des réplicas de telle manière que les applications peuvent avancer même si quelques nœuds sont déconnectés ou sont tombés en panne. Par conséquent, les utilisateurs peuvent collaborer de manière asynchrone. Cependant, des mises-à-jour simultanées peuvent causer des divergences et des conflits entre les réplicas, qui doivent être réconciliés. Dans cet étude, nous présentons une synthèse à propos de la réplication de données en concentrant sur l'approche optimiste qui fourni de bonnes propriétés aux environnements dynamiques. Nous introduisons aussi les systèmes P2P et les solutions de réplication qu'ils implémentent. En particulier, nous montrons que les systèmes P2P existants n'assurent pas la cohérence éventuelle entre les réplicas en présence de mises-à-jour, à part le système APPA, un système de gestion de données P2P que nous sommes en train de développer.

Mots clés: réplication de données, réconciliation, cohérence éventuelle, réseaux pair-à-pair

1 Introduction

Large-scale distributed collaborative applications are getting common as a result of rapid progress in distributed technologies (grid, peer-to-peer, and mobile computing). As an example of such applications, consider a second generation Wiki that works over a peer-to-peer (P2P) network and supports mobile users. A lot of users, in particular managers, are frequently traveling and they need to access and update information even if they are disconnected from the network, e.g. in an aircraft or train. This requires that users hold local replicas of shared data. Thus, such a P2P Wiki has need for multi-master replication to assure data availability at anytime. In the multi-master approach, updates made offline or in parallel on different replicas of the same data may cause replica divergence and conflicts, which should be reconciled. In order to reduce conflicts, the reconciliation solution has to take advantage of application semantics (e.g. multiple parallel updates on the same XML document are not in conflict if they handle distinct independent elements). Obviously, mutual consistency among replicas cannot be assured in the presence of disconnected updates. However, a collaborative application as the P2P Wiki must count on eventual consistency, i.e. replicas' states must converge in such a way that if users stop to submit updates (e.g. the collaborative edition of an XML document terminates) all replicas eventually achieve the same final state. To manage information, users take advantage of different devices such as notebooks, PDAs and portable phones, which can be supported by networks of variable quality. As a result, it is not acceptable that the replication solution make strong network assumptions. Finally, a collaborative application like the P2P Wiki may handle different data types (e.g. XML documents, relational tables, etc.), and therefore the replication solution needs to be independent of data types. Hence, we can summarize the replication requirements of large-scale collaborative applications as follows: high-level of autonomy, multi-master replication, semantic conflict detection, eventual consistency among replicas, weak network assumptions, and data type independency.

Optimistic replication addresses most of these requirements by allowing asynchronous updating of replicas so that applications can progress even though some nodes are disconnected or have failed. As a result, users can collaborate asynchronously. However, existing optimistic solutions are unsuitable for P2P networks since they are either centralized or do not take into account the network limitations. Centralized approaches are inappropriate due to their limited availability and vulnerability to failures and partitions from the network. On the other hand, variable latencies and bandwidths, typically found in P2P networks, may strongly impact the reconciliation performance once data access times may vary significantly from node to node. Therefore, in order to build a suitable P2P reconciliation solution, optimistic replication techniques must be revisited.

In this survey, we first present an overview of data replication focusing on the optimistic approach that provides good properties for dynamic environments such as P2P networks. Then, we introduce P2P systems and the replication solutions they implement. In particular, we show that current P2P systems do not provide eventual consistency among replicas in the presence of updates, apart from APPA system, a P2P data management system that we are building. The rest of this survey is organized as follows. Section 2 introduces basic concepts wrt. data replication. Section 3 discusses in details optimistic replication. Section 4 presents P2P systems and the associated replication solutions. Finally, Section 5 concludes this survey.

2 Basic concepts

Data replication consists of maintaining multiple copies of data objects, called replicas, on separate sites [SS05]. An *object* is the minimal unit of replication in a system. For instance, in a replicated relational database, if tables are entirely replicated then tables correspond to objects; however, if it is possible to replicate individual tuples, then tuples correspond to objects. Other examples of objects include XML documents, typed files, multimedia files, etc. A *replica* is a copy of an object stored in a site. We call *state* the set of values associated with an object or a

replica at a given time. In addition, we use *computer* and *node* as synonyms of *site* throughout this survey.

Data replication is very important in the context of distributed systems for several reasons. First, replication improves the system availability by removing single points of failures (objects are accessible from multiple sites). Second, it enhances the system performance by reducing the communication overhead (objects can be located closer to their access points) and increasing the system throughput (multiple sites serve the same object simultaneously). Finally, replication improves the system scalability as it supports the growth of the system with acceptable response times.

A relevant issue concerning data replication is how to manage updates. Gray et al. [GHOS96] classify the replica control mechanisms according to two parameters: *where* updates take place (i.e. which replicas can be updated), and *when* updates are propagated to all replicas. According to the first parameter (i.e. where), replication protocols can be classified as *single-master* or *multi-master* solutions, as described in subsection 2.1. According to the second parameter (i.e. when), update propagation strategies are divided into *synchronous* (eager) and *asynchronous* (lazy) approaches, as described in subsection 2.3. The replica control mechanisms are also affected by the way in which replicas are distributed over the network (replica placement). Subsection 2.2 discusses the *full* and *partial* replication alternatives.

Update an object with multiple replicas and preserve equal replica states after the update is a challenging problem. Indeed, several replication solutions allow that different replicas of a single object hold different states for a while. This difference can be caused by the delay associated with the update propagation or by the presence of conflicting updates on distinct replicas, which must be reconciled. Thus, we say that two replicas are *mutually consistent* if they hold equal states at a given time. In contrast, we say that two replicas are *divergent* if they hold different states due to the parallel execution of conflicting updates. Finally, a replica is not *fresh* if its state does not reflect all validated updates due to the propagation delay (in this case, conflicting updates are prevented).

2.1 Single-master *vs.* multi-master

A replica of an object can be classified as primary copy or secondary copy according to its updating capabilities. A *primary copy* accepts read and write operations and is held by a *master site*. A *secondary copy* accepts only read operations and is held by a *slave site*.

In the *single-master* approach, there is only a single primary copy for each replicated object. In this case, every update is first applied to the primary copy at the master site, and then it is propagated towards the secondary copies held by the slave sites. Due to the interaction between master and slave sites, this approach is also known as *master/slave* replication. Centralizing updates at a single copy avoids concurrent updates on different sites, thereby simplifying the concurrency control. In addition, it assures that one site has the up-to-date values for an object. However, this centralization introduces a potential bottleneck and a single point of failure. Therefore, a failure in a master site blocks update operations, and thus limits data availability. Figure 1 shows an example of single-master replication with one primary copy and two secondary copies.

In the *multi-master* approach, multiple sites hold primary copies of the same object. All these copies can be concurrently updated, wherefrom the multi-master technique is also known as *update anywhere*. Distributing updates avoids bottlenecks and single points of failures, thereby improving data availability. However, in order to assure data consistency, the concurrent updates to different copies must be coordinated or a reconciliation algorithm must be applied to solve replica divergences. On the one hand, coordinating distributed updates can lead to expensive communication, and on the other hand reconciliation solutions can be complex. Figure 2 shows an example of multi-master replication.

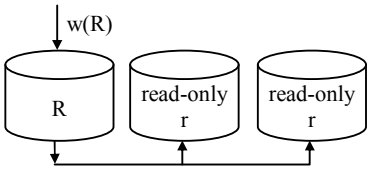


Figure 1. Single-master replication; *R* is a *primary copy* and *r* a *secondary copy*

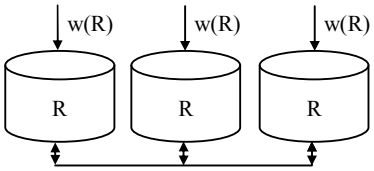


Figure 2. Multi-master replication; *R* is a *primary copy*

Table 1 summarizes the concepts introduced in this subsection.

Compared aspect	**Single-master**	**Multi-master**
Distinguishing feature	One primary copy	Multiple primary copies
Synonymous	Master/slave	Update anywhere
Distributed concurrency control	Not applied	Coordination Reconciliation
Up-to-date values at	Primary copy	Unknown copy
Update approach	Centralized	Distributed
Update blocking	Master site down	All master sites down (if using reconciliation)
Possible bottleneck	Yes	No

Table 1. Single-master *vs*. multi-master

2.2 Full replication *vs.* partial replication

Replica placement over the network directly affects the replica control mechanisms. In this subsection, we discuss the basic alternative approaches for replica placement: full replication and partial replication.

Full replication consists of storing a copy of every shared object at all participating sites. This approach provides simple load balancing since all sites have the same capacities, and maximal availability as any site can replace any other site in case of failure. Figure 3 presents the full replication of two objects named *R* and *S* respectively over three sites.

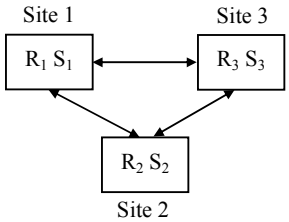


Figure 3. Example of full replication with two objects *R* and *S*

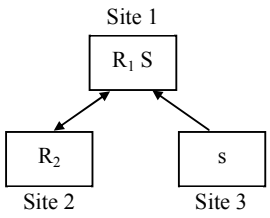


Figure 4. Example of partial replication with two objects *R* and *S*

With *partial replication*, each site holds a copy of a subset of shared objects, so that the objects replicated at one site may be different of the objects replicated at another site, as shown in Figure 4. This approach expends less storage space and reduces the number of messages needed to update replicas since updates are only propagated towards the affected sites (i.e. sites holding primary or secondary copies of the updated objects). Thus, updates produce reduced load for the network and sites. However, if related objects are stored at different sites, the propagation protocol becomes more complex as the replica placement must be taken into account. In addition,

this approach limits load balance possibilities since certain sites are not able to execute a particular set of transactions.

Table 2 summarizes the concepts introduced in this subsection.

Compared aspect	**Full replication**	**Partial replication**
Distinguishing feature	All sites hold copies of all shared objects	Each site holds a copy of a subset of shared objects
Load balancing	Simple	Complex
Availability	Maximal	Less
Storage space	May be expensive	Reduced
Communication costs	May be expensive	Reduced

Table 2. Full replication *vs*. partial replication

2.3 Synchronous *vs.* asynchronous

In distributed database systems, data access is done via transactions. A *transaction* is a sequence of read, write operations followed by a *commit*. If the transaction does not complete successfully, we say that it *aborts*. A transaction that updates a replicated object must be propagated to all sites that hold replicas of this object in order to keep its replicas consistent. Such update propagation can be done within the transaction boundaries or after the transaction commit. The former is called synchronous, and the latter, asynchronous propagation. In this subsection, we discuss these propagation approaches.

2.3.1 Synchronous propagation

The *synchronous* update propagation approaches (a.k.a. *eager*) apply the changes to all replicas within the context of the transaction that initiates the updates, as shown in Figure 5. As a result, when the transaction commits, all replicas have the same state. This is achieved by using concurrency control mechanisms like two-phase-locking (2PL) [OV99] or timestamp based algorithms. In addition, a commitment protocol like two-phase-commit (2PC) [OV99] can be run to provide atomicity (either all transaction's operations are completed or none of them are). Thus, synchronous propagation enforces mutual consistency among replicas. Bernstein et al. [BHG87] define this consistency criteria as *one-copy-serializability*, i.e. despite the existence of multiple copies, an object appears as one logical copy (*one-copy-equivalence*), and a set of accesses to the object on multiple sites is equivalent to serially execute these accesses on a single site.

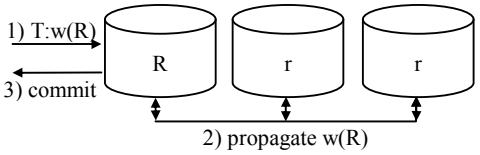


Figure 5. Principle of synchronous propagation

Early solutions [AD76, Sto79] use synchronous single-master approaches to assure one-copy-serializability. However, most of the algorithms avoid this centralized solution and follow the multi-master approach by accessing a sufficient number of copies. For instance, in the ROWA (*read-one/write-all*) approach [BHG87], read operations are done locally while write operations access all copies. ROWA is not fault-tolerant since the update processing stops whenever a copy is not accessible. ROWAA (*read-one/write-all-available*) [BG84, GSC+83] overcomes this limitation by updating only the available copies. Another alternative are quorum protocols [Gif79, JM87, PL88, Tho79], which can succeed as long as a quorum of copies agrees on executing the operation. Other solutions combine ROWA/ROWAA with quorum protocols [ES83, ET89].

More recently, Kemme and Alonso [KA00] proposed new protocols for eager replication that take advantage of group communication systems to avoid some performance limitations of the existing protocols. Group communication systems [CKV01] provide group maintenance, reliable message exchange, and message ordering primitives between groups of nodes. The basic mechanism behind the new protocols is to first perform a transaction locally, deferring and batching writes to remote replicas until transaction commit time. At commit time all updates (the write set) are sent to all replicas using a total order multicast which guarantees that all nodes receive all write sets in exactly the same order. As a result, no two-phase commit protocol is needed and no deadlock can occur. Following this approach, Jiménez-Peris et al. [JPAK03] show that the ROWAA approach, instead of quorums, is the best choice for a large range of applications requiring data replication in cluster environments. Next, in [LKPJ05] the most crucial bottlenecks of the existing protocols are identified, and optimizations are proposed to alleviate these problems, making *one-copy-serializability* feasible in WAN environments of medium size.

The main advantage of the synchronous propagation is to avoid divergences among replicas. This enables local reads since transactions surely take up-to-date values. The drawback is that the transaction has to update all replicas before committing. If one replica is unavailable, this can block the transaction, making synchronous propagation unsuitable for dynamic networks. In addition, the transaction response times and the communication costs increase with the number of replicas and, for these reasons, this approach does not scale beyond a few tens of sites.

2.3.2 Asynchronous propagation

The *asynchronous* update propagation approaches (a.k.a. *lazy*) do not change all replicas within the context of the transaction that initiates the updates. Indeed, the initial transaction commits as soon as possible, and afterwards the updates are propagated to all replicas, as shown in Figure 6. Asynchronous replication solutions can be classified as *optimistic* or *non-optimistic* according to their assumptions concerning conflicting updates. In general, *optimistic asynchronous replication* relies on the optimistic assumption that conflicting updates will occur only rarely, if at all. Updates are therefore propagated in the background, and occasional conflicts are fixed after they happen. In contrast, *non-optimistic asynchronous replication* assumes that update conflicts are likely to occur and implements propagation mechanisms that prevent conflicting updates.

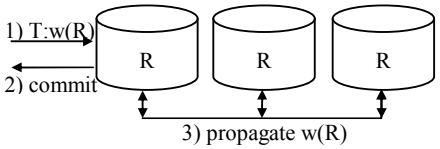


Figure 6. Principle of asynchronous propagation

An advantage of the asynchronous propagation is that the update does not block due to unavailable replicas, which improves data availability. In addition, communication is not needed to coordinate concurrent updates, thereby reducing the transaction response times and improving the system scalability. In particular, the *optimistic asynchronous* replication is more flexible than other approaches as the system can choose the appropriate time to propagate updates and the application can progress over a dynamic network in which nodes can connect and disconnect at any time. Its main drawback is that replicas may diverge, and then local reads are not guaranteed to return up-to-date values. The *non-optimistic asynchronous* replication is not as flexible as the optimistic approach, but it provides up-to-date values for local reads with high probability.

2.3.2.1 Non-optimistic approaches

The goal of non-optimistic asynchronous solutions is to use lazy replication while still providing one-copy-serializability. Chundi et al. [CRR96] have shown that serializability cannot be guaranteed in every case. To circumvent this problem, it is necessary to restrict the placement of primary and secondary copies across the system. The main idea is to define the set of allowed configurations using graphs, so that nodes represent sites and there is a non-directed edge between two sites if one holds the primary copy and the other a secondary copy of a given object. If this graph is acyclic, serializability can be guaranteed by simply propagating updates sometime after transaction commits [CRR96].

Pacitti et al. [PSM98, PMS99, PS00] have enhanced these initial results by allowing certain cyclic configurations. The replication algorithm assumes that the network provides FIFO reliable multicast, there is an upper bound on the time needed to multicast a message from a node to any other node (noted *Max*), and local clocks are ε-synchronized (i.e. the difference between any two correct clocks is not higher than ε). As a result, a transaction is propagated in at most $Max + \varepsilon$ units of time and chronological and total orderings can be assured with no coordination among sites. Experimental results show that such approach assures a consistency level equivalent to one-copy-serializability for normal workloads, and for bursty workloads the consistency level is still quite close to one-copy-serializability. Coulon et al. [CPV05] have extended this solution to work properly in the context of partial replication.

Breitbart et al. [BKRS+99] propose alternative solutions. The first one requires acyclic directed configuration graphs (edges are directed from primary copy to secondary copy). The second solution, in contrast, allows cyclic graphs, and applies lazy propagation along acyclic paths while eager replication is used whenever there are cycles.

Since these approaches use lazy update propagation, the state of a replica can be somewhat stale with respect to committed (validated) transactions. Thus, the associated consistency criterion is *freshness*, which is defined as the distance between two replicas wrt. validated transactions.

2.3.2.2 Optimistic approaches

Contrasting with non-optimistic approaches, optimistic replication does not aim to provide one-copy-serializability. Indeed, it assumes that conflicts are rare or do not happen. Thus, update propagation is made in background and replica divergences may arise. Conflicting updates are reconciled later, which means that the application must tolerate some level of divergence among replicas. This is acceptable for a large range of applications (e.g. DNS Internet name service, mobile database systems, collaborative software development, etc.). We now introduce some optimistic solutions that will be discussed in the following.

- **DNS:** *Domain Name System* is the standard hierarchical name service for the Internet [AL01]. Names for a particular zone (a subtree in the name space) are managed by a single master site that maintains the authoritative database for that zone and optional slave sites that copy the database from the master. The master and slaves can answer queries from remote sites.
- **LOCUS:** it is a distributed operating system [PPRS+83, WPEK+83] composed of a replicated file system. The file system uses version vectors to order updates on distinct replicas of the same object. A version vector [PPRS+83, Fid88, Mat89] is an array of timestamps that allows detecting update conflicts. For LOCUS, any two concurrent updates to the same object are in conflict. It automatically resolves conflicts by taking two versions of the object and creating a new one.
- **TSAE:** *Time-Stamped Anti Entropy* uses real-time clocks to order operations [Gol92]. Basically, sites exchange vector clocks (i.e. arrays of timestamps) and acknowledge vectors in order to learn about the progress of others, so that a site *i* is able to determine

which operations have surely been received by all sites at a given time. As a result, site *i* can safely execute these operations in the timestamp order and delete them. TSAE does not perform any conflict detection or resolution. It only needs to agree on the set of operations and their order.

- **Ramsey and Csirmaz's file system:** Ramsey and Csirmaz formally study the semantic of a simple file system that supports few operation types, including create, remove, and edit [RC01]. For every possible pair of concurrent operations, they define a rule that specifies how the operations interact and may be ordered. Non concurrent operations are executed in the submission order.
- **Unison:** it is a file synchronizer that reconciles two replicas of a file or directory [PV04, Uni06] based only on the current states of the replicas (i.e. it does not use operation logs). Unison takes into account the semantics of the file system when trying to merge two replicas. Non-conflicting updates are automatically propagated, but nothing is done with conflicting updates. Thus, after reconciliation replicas may hold different states.
- **CVS:** *Concurrent Versions System* is a version control system that lets users edit a group of files collaboratively and retrieve old versions on demand [CP+01, Ves03]. A central site stores the repository that contains authoritative copies of the files and the associated changes. Users create private copies (replicas) of the files and modify them concurrently. After that, users commit private copies to the repository. CVS automatically merges changes of distinct users on the same file if there is no overlap. Otherwise, user must resolve conflicts manually.
- **OT:** *Operational Transformation* was developed for collaborative editors [EG89, SYZC96, SE98, SJZY+98, VCFS00]. OT assumes that a user applies commands immediately at the local site, and then propagates these commands to other sites. As a result, all sites perform the same set of operations but possibly in different orders. The goal of OT is to preserve the intention of operations and assure replica convergence. This is achieved by defining for every pair of concurrent operations a rewriting rule. In [PC98] it is proved the correctness of OT for a shared spreadsheet. Molli et al. [MOSI03] extend the OT approach to support a replicated file system. Ferrié et al. [FVC04] deal with undo operations in the context of OT by providing a general undo algorithm based on the definition of a generic undo-fitted transformation.
- **Harmony:** the Harmony system is a generic framework for reconciling disconnected updates to heterogeneous, replicated XML data [PSG04, FGMP+05, Har06]. For instance, Harmony is used to reconcile the bookmarks of multiple web browsers (Mozilla, Safari, OmniWeb, Internet Explorer, and Camino). This application allows bookmarks and bookmark folders to be added, deleted, edited, and reorganized by different users on disconnected machines. Similar to Unison, Harmony takes only replica states and it does not resolve update conflicts.
- **Bayou:** it is a research mobile database system that lets a user replicate a database on a mobile computer, modify it while disconnected, and synchronize with any other replica of the database that the user happens to find [TTPD+95, PSTT+97]. In Bayou, each operation has attached a *dependency check* and a *merge procedure*. The dependency check is run to verify if the operation conflicts with others whereas the merge procedure is executed to repair the replica state in case of conflict. In Bayou, a single primary site decides which operations should be committed or aborted and notifies other sites about the sequence in which operations must be executed. Anyway, Bayou remains different from single-master systems as it allows any site to submit operations and propagate them, letting users to quickly see the operations effects. In single-master systems, only the master can submit updates.
- **IceCube:** it is a general-purpose reconciliation system that exploits the application semantics to resolve conflicting updates [KRSD01, PSM03, SBK04]. In IceCube, update

operations are called actions and they are stored in logs. IceCube captures the application semantics by means of constraints between actions, and treats reconciliation as an optimization problem where the goal is to find the largest set of actions that do not violate the stated constraints.

- **Distributed log-based reconciliation:** Chong and Hamadi [CH06] propose distributed algorithms for log-based reconciliation also based on the action-constraint framework introduced by IceCube. Thus, actions and constraints are partitioned amongst a set of nodes that locally compute the largest set of non conflicting actions, and then combine these local solutions into a global consistent distributed solution. This approach requires an ordering between nodes that share constraints.

Most of these optimistic replication systems assure eventual consistency [SBK04, SS05] among replicas. Eventual consistency can be formally defined based on the concept of schedule equivalence. A schedule is an ordered list of operations. Two schedules are *equivalent* when, starting from the same initial state, they produce the same final state. Notice that a final state does not include tentative operations (i.e. operations not yet committed), but only committed ones. If a schedule contains commutative operations, swapping their order preserves the equivalence. Therefore, a replicated object is eventually consistent when it meets the following conditions, assuming that all replicas start from the same initial state:

- At any time, for each replica, there is a prefix of the schedule that is equivalent to a prefix of the schedule of every other replica. It is called *committed prefix*.
- The committed prefix of each replica grows monotonically over time.
- For every submitted operation α, either α or $\neg\alpha$ will eventually be included in the committed prefix, where $\neg\alpha$ denotes an aborted operation.
- All non aborted operations in the committed prefix can be successfully executed.

2.3.3 Summary

Table 3 summarizes the synchronous and asynchronous update propagation strategies.

Compared aspect	**Synchronous**	**Asynchronous**	
		Non-optimistic	**Optimistic**
Distinguishing feature	All replicas change in the same update transaction	Commit as soon as possible, then propagation	Commit, then background propagation
Synonymous	Eager propagation	Lazy propagation	
Consistency criterion	One-copy-serializability	Freshness	Eventual consistency
Local reads	Return up-to-date values	Return up-to-date values with high probability	No guarantees
Distributed Concurrency control	Yes	No	No
Scalability	A few tens of sites	A few hundreds of sites	Larger number of sites
Environment	LAN and cluster	LAN, cluster, and WAN	Anywhere

Table 3. Synchronous propagation *vs*. asynchronous propagation

3 Optimistic replication parameters

In the previous section, we introduced some optimistic solutions for managing replicated objects. In order to compare these solutions, we now abstract their main characteristics by defining five parameters: operation storage, operation relationships, propagation frequency, conflict detection and resolution, and reconciliation. We describe these parameters by providing alternative values and presenting examples of optimistic solutions that implement each alternative. At the end of the section we present a comparative table.

3.1 Operation storage

An *operation* is a prescription to update an object. Many optimistic replication systems store operations in log files, and then propagate these operations to other sites to assure replica consistency (e.g. Bayou [TTPD+95, PSTT+97] and IceCube [KRSD01, PSM03, SBK04]). Such systems are called *operation-transfer systems*. In contrast, other systems deal with the consistency problem by propagating the updated state of a replica to other sites (e.g. DNS [AL01], Unison [PV04, Uni06], and Harmony [PSG04, FGMP+05, Har06]). Such systems are called *state-transfer systems*. In this case, replica divergences can be resolved as follows: in single-master models, the state of the secondary copy is completely replaced by the updated state of the primary copy; in multi-master models, the states associated to different replicas are compared in order to identify and resolve divergences, if possible. Thus, we classify optimistic solutions according to the policy for storing operations as follows. *Persistent operations*: operations are stored somewhere (e.g. log file) to be propagated later. *Transient operations*: operations are discarded just after execution.

3.2 Operation relationships

Operation relationships represent implicit or explicit associations between operations. Based on operation relationships we can detect update conflicts and resolve them by arranging operations in a convenient sequence. Four types of relations between operations are especially meaningful for optimistic replication systems: happens-before, concurrency, explicit constraint, and implicit constraint.

- **Happens-before:** the concept of *happens-before* is an implementable partial ordering that intuitively captures the relations between distributed events [Lam78]. Let α and β be two operations executed respectively at sites i and j. Operation α *happens before* β when: (i) $i = j$ and α was submitted before β; or (ii) $i \neq j$ and β is submitted after j has received and executed α; or (iii) $i \neq j$ and β is submitted after j has received and executed α; or (iv) for some operation γ, α happens before γ and γ happens before β.
- **Concurrency:** if neither α nor β happens before the other, they are said to be *concurrent*.
- **Explicit constraint:** an explicit constraint is an invariant dynamically introduced in the system to represent the application semantic. For instance, in Bayou [TTPD+95, PSTT+97] dependency checks are dynamically associated with operations, thus playing the role of explicit constraints. IceCube [KRSD01, PSM03, SBK04] supports several types of explicit constraints, including dependence (α executes only after β does), implication (if α executes, so does β), choice (either α or β may be applied, but not both), and so forth. In IceCube, constraints can be provided by several sources: the user, the application, a data type, or the system.
- **Implicit constraint**: an implicit constraint is an invariant statically introduced in the system to represent the application semantic; this means, an implicit constraint is embedded in the reconciliation engine, such that users and applications cannot dynamically change the associated semantics. For instance, the replication system proposed by Ramsey and Csirmaz [RC01] implements the semantic of a distributed file system using implicit constraints. Harmony [PSG04, FGMP+05, Har06] implements the semantic of tree structures by means of implicit constraints in order to reconcile divergent XML documents.

3.3 Propagation frequency

Propagation is the exchange of operations or replica states among sites in order to assure replica consistency. The frequency of operation or state exchanges relies on the degree of synchrony adopted by the propagation strategy, which can be *pulling*, *hybrid* or *pushing*. Each site in a

pull-based system takes new operations or states by pulling other sites, either *on demand* (e.g. CVS [CP+01, Ves03]) or *periodically* (e.g. DNS [AL01]). In push-based systems, a site with new updates proactively sends them to others *as soon as possible* (e.g. LOCUS [PPRS+83, WPEK+83]). Hybrid systems combine pull and push behaviors (e.g. TSAE [Gol92]). In general, the quicker the propagation happens, the lower the degree of replica divergence and the rate of conflict. Therefore, push-based propagation provides the best degree of replica consistency.

3.4 Conflict detection and resolution

Without site coordination, multiple users may update replicas of the same object at the same time. Such concurrent updates may raise update conflicts. An operation α is in conflict if α cannot be successfully executed according to the order established in the schedule to which α belongs. Thus, *conflict detection* consists of recognizing conflicts in a schedule, while *conflict resolution* refers to change the schedule in order to remove conflicts. We express the conflict parameter in the following tuple format: *conflict* = <*detection*, *resolution*>.

We classify *conflict detection* policies as *none*, *concurrency-based* and *semantic-based*. In systems with *none* policy (e.g. DNS [AL01]) conflicts are ignored. Indeed, any potentially conflicting operation is simply overwritten by a newer operation causing *lost updates*. Systems with concurrency-based policy (e.g. LOCUS [PPRS+83, WPEK+83]) declare a conflict between two operations based on the timing of operation submission. Finally, systems that know operations' semantics (e.g. Bayou [TTPD+95, PSTT+97] and IceCube [KRSD01, PSM03, SBK04]) can exploit that to reduce conflicts. For instance, in a room-booking application, two concurrent reservation requests for the same room object could be granted as long as their duration does not overlap. Concurrency-based policies are simpler and generic but cause more conflicts, while semantic-based policies are more flexible but application-specific. In this survey, we focus on semantic-based conflict detection in order to reduce conflicts.

Conflict resolution can be either *manual* or *automatic*. In the manual approach, the offending operation is removed from the schedule, and two versions of the object are presented to the user, who must create a new, merged version and resubmit the operation. CVS [CP+01, Ves03] is a system that uses this strategy. In contrast, automatic approaches do not require the user intervention. There are several strategies to automatically resolve conflicts. For example, Bayou [TTPD+95, PSTT+97] executes a *merge procedure* every time a conflict happens in order to repair the replica state. In file systems, an application-specific procedure takes two versions of an object and creates a new one. For instance, concurrent updates on a mail folder file can be resolved by computing the union of the messages from two replicas.

3.5 Reconciliation

Optimistic replication allows parallel update of replicas of a single object so that applications can progress even though some nodes are disconnected or have failed. This enables asynchronous collaboration among users. However, such parallel updates may cause conflicts and replica divergence. *Reconciliation* is the activity that brings divergent replicas back to a mutual consistent state. Different reconciliation strategies can be established according to the type of input information and the criterion for ordering updates. We express the reconciliation parameter in the following tuple format: *reconciliation* = <*input*,*ordering*>.

The *input* information handled by a reconciliation engine can be the updated state of replicas or the update operations. Thus, we call *state-based reconciler* a reconciliation engine that takes the states of replicas at a given time and tries to make them as similar as possible. Harmony [PSG04, FGMP+05, Har06] and Unison [PV04, Uni06] are representatives of this class. On the other hand, we call *operation-based reconciler* a reconciliation engine that accesses all operations performed on each replica and builds a common sequence of operations. Bayou [TTPD+95, PSTT+97] and IceCube [KRSD01, PSM03, SBK04] belong to this category.

The criterion used for ordering reconciled updates can be based on semantic properties or some ordinal information associated with updates. Therefore, we call *ordinal reconciler* a rec-

onciliation engine that tries to preserve at least the submission order of updates based on information about when, where, and by whom updates were performed. Timestamp-based ordering, as implemented by TSAE [Gol92], is the most popular example of this strategy. Version vectors also provide total order among object states in the absence of concurrent updates, as used in LOCUS [PPRS+83, WPEK+83]. On the other hand, we call *semantic reconciler* a reconciliation engine that exploits semantic properties associated with updates to reduce conflicts. For instance, Ramsey and Csirmaz [RC01] order file system operations according to the file system semantic. Collaborative editors [EG89, SYZC96, PC98, SE98, SJZY+98, VCFS00] adapt operations performed on replicas of the same object to allow different orderings per replica while preserving the operations' intentions. IceCube [KRSD01, PSM03, SBK04] captures the application semantics by means of constraints between actions (operations), and orders such actions avoiding constraint violation.

In the next subsections we describe IceCube and Harmony, respectively the major representatives of operation-based and state-based reconciliation engines. In addition, we compare these solutions according to our optimistic replication parameters.

3.5.1 IceCube

IceCube [KRSD01, PSM03, SBK04] describes the application semantics by means of constraints between actions. An *action* is defined by the application programmer and represents an application-specific operation (*e.g.* a write operation on a file or document, or a database transaction). A *constraint* is the formal representation of an application invariant (*e.g.* an update cannot follow a delete). Constraints are classified as follows:

- **User-defined constraint**[4]: user and application can create user-defined constraints to make their intents explicit. The *predSucc*(a_1, a_2) constraint establishes causal ordering between actions (*i.e.* action a_2 executes only after a_1 has succeeded); the *parcel*(a_1, a_2) constraint is an atomic (all-or-nothing) grouping (*i.e.* either a_1 and a_2 execute successfully or none does); the *alternative*(a_1, a_2) constraint provides choice of at most one action (*i.e.* either a_1 or a_2 is executed, but not both).
- **System-defined constraint**[5]: it describes a semantic relation between classes of concurrent actions. The *bestOrder*(a_1, a_2) constraint indicates the preference to schedule a_1 before a_2 (*e.g.* an application for account management usually prefers to schedule credits before debits); the *mutuallyExclusive*(a_1, a_2) constraint states that either a_1 or a_2 can be executed, but not both.

a_1^1: update T_1 set A=a1 where K=k1
a_2^1: update T_2 set A=a2 where K=k1
a_3^1: update T_3 set B=b1 where K=k1
a_3^2: update T_3 set A=a3 where K=k2
Parcel(a_3^1, a_3^2)

Example 1. Conflicting actions on T

Let us illustrate user- and system-defined constraints with Example 1. In this example, an action is noted a_n^i, where n indicates the node that has executed the action and i is the action identifier. T is a replicated object, in this case, a relational table; K is the key attribute for T; A and B are any two attributes of T. T_1, T_2, and T_3 are replicas of T. Consider that the actions in Example

[4] In IceCube, it is called *log* constraint. We prefer *user-defined* to emphasize the user intent.
[5] In IceCube, it is called *system-defined* constraint. We use *system-defined* to contrast with user intents.

1 (with the associated constraints) are concurrently produced by nodes n_1, n_2 and n_3, and should be reconciled.

In Example 1, actions a_1^1 and a_2^1 try to update the same data item (*i.e. T*'s tuple identified by *k1*) over different replicas. The IceCube reconciliation engine realizes this conflict and asks the application for the semantic relationship involving a_1^1 and a_2^1. As a result, the application analyzes the intents of both actions, and, as they are really in conflict (*i.e.* n_1 and n_2 try to set the same attribute with distinct values), the application produces a *mutuallyExclusive*(a_1^1, a_2^1) *system-defined constraint* to properly represent this semantic dependency. Notice that from the point of view of the reconciliation engine a_3^1 also conflicts with a_1^1 and a_2^1 (*i.e.* all these actions try to update the same data item). However, by analyzing actions' intents, the application realizes that a_3^1 is semantically independent of a_1^1 and a_2^1 as a_3^1 tries to update another attribute (*i.e. B*). Therefore, in this case no system-defined constraints are produced. Actions a_3^1 and a_3^2 are involved in a *parcel user-defined constraint*, so they are semantically related.

The aim of reconciliation is to take a set of actions with the associated constraints and produce a *schedule*, *i.e.* a list of ordered actions that do not violate constraints. In order to reduce the schedule production complexity, the set of actions to be ordered is divided into subsets called *clusters*. A cluster is a subset of actions related by constraints that can be ordered independently of other clusters. Therefore, the *global schedule* is composed by the concatenation of clusters' ordered actions. To order a cluster, IceCube performs iteratively these operations:

- Select the action with the highest merit from the cluster and put it into the schedule. The merit of an action is a value that represents the estimated benefit of putting it into the schedule (the larger the number of actions that can take part in a schedule containing a_n^i is, the larger the merit of a_n^i will be). If more than one action has the highest merit (different actions may have equal merits), the reconciliation engine selects randomly one of them.
- Remove the selected action from the cluster.
- Remove from the cluster the remaining actions that conflict with the selected action.

This iteration ends when the cluster becomes empty. As a result, cluster's actions are ordered. Indeed, several alternative orderings may be produced until finding the best one.

3.5.2 Harmony

The Harmony system [PSG04, FGMP+05, Har06] is a generic framework for reconciling disconnected updates to heterogeneous, replicated XML data. For example, an instance of Harmony that reconciles calendars on multiple formats (Palm Datebook, Unix ical, and iCalendar) is in daily use within the group responsible for the Harmony project. Another application that has been built on top of Harmony is a bookmark reconciler that handles multiple web browser formats (Mozilla, Safari, OmniWeb, Internet Explorer, and Camino). This reconciler allows bookmarks and bookmark folders to be added, deleted, edited, and reorganized by different users on disconnected machines.

The objects handled by Harmony are edge-labeled trees in which all children of a given node are labeled with distinct names. Thus, for Harmony, an object is a tree and a replica is a copy of a tree. The reconciliation of divergent replicas relies on two basic concepts: alignment and lens. *Alignment* consists of determining which parts of the involved replicas are intended to represent the same information. A *lens* allows transforming a concrete tree into an abstract tree (called *view*) and putting back the abstraction contents into the concrete representation. For instance, when reconciling the bookmarks b_1 and b_2 of two distinct web browsers (b_1 and b_2 have *incompatible* concrete formats) a lens allows to extract two *compatible* abstract views v_1 and v_2 from b_1 and b_2 respectively, and to put back an updated (reconciled) version of v_1 and v_2 into b_1 and b_2. Formally, let T be a set of trees; a *lens* l comprises a partial function $l\nearrow$ from T to T, called the *get function* of l, and a partial function $l\searrow$ from $T \times T$ to T, called the *putback function*.

Figure 7 shows the Harmony's architecture [PSG04] which consists of two major components: (1) a single reconciliation engine (*Reconciliation*) that takes two current replicas (R_1 and R_2) and a common ancestor (R) (all three represented as trees) as input and yields new replicas (R_1' and R_2') in which all non-conflicting changes have been merged; and (2) a bi-directional programming language [FGMP+05], composed of a collection of *lens combinators*, which allows extracting views of complex data structures and putting back updated views into the original structures. Lens combinators are assembled to describe transformations on trees. These combinators include familiar constructs from functional programming (composition, mapping, projection, conditionals, and recursion) together with some novel primitives for manipulating trees (splitting, pruning, copying, merging, etc.).

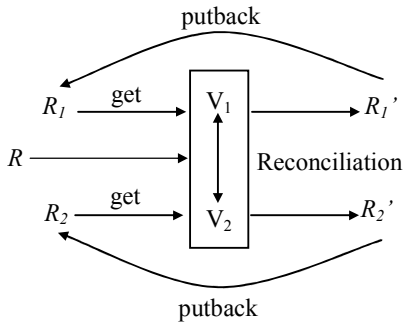


Figure 7. Harmony architecture

When reconciling replicas, updates that violate constraints associated with the tree structure are not performed. In Harmony, constraints are predefined and coupled with the reconciliation engine, so we call them *implicit constraints*. The violation of a constraint while reconciling two replicas raises a *conflict*. Table 4 summarizes Harmony's implicit constraints and the associated conflicts.

Implicit Constraints	**Conflicts**
A tree node cannot be deleted in one replica and updated in the other (update means adding a new child to the tree node or to one of its descendants)	Delete/Create
A subtree cannot be entirely delete in one replica and partially deleted in the other	Delete/Delete
Different subtrees cannot hold the same place in a tree	Create/Create
Corresponding subtrees reached by edges labeled @ must be identical	Atomicity

Table 4. Harmony's implicit constraints and the associated conflicts

The Harmony's reconciler algorithm works as follows. Let R_1 and R_2 be two replicas under reconciliation. Pairs of tree nodes (n_{R1}, n_{R2}) that correspond to each other in both replicas are recursively visited and checked with respect to their current state. If n_{R1} is equal to n_{R2} (i.e. n_{R1} and n_{R2} are already synchronized) or they are different, but a conflict between n_{R1} and n_{R2} is detected, the reconciler algorithm keeps n_{R1} and n_{R2} unchanged in the respective replicas. Otherwise, i.e. n_{R1} and n_{R2} are different and free of conflict, updates are applied to one or both replicas in order to yield $n_{R1} = n_{R2}$. In addition, the output replicas are checked against an intended schema in order to avoid the return of ill-formed structures. With this approach, the Harmony's semantic reconciler satisfies the following specification requirements:

- Never back out changes.
- Never make up contents.
- Stop at conflicting paths leaving replicas in their current states.
- Always leave the replicas in a well-typed form (safety condition).
- Propagate as many changes as possible without violating above rules (maximality condition).

3.5.3 IceCube vs. Harmony

Both IceCube and Harmony aim at reconciling divergent replicas based on semantics. However, they achieve this common goal in quite different manners. Table 5 shows the distinguishing features of these solutions according to our optimistic replication parameters.

The first striking difference between IceCube and Harmony is that the former is generic (it can handle any kind of object) and flexible (the user and application can dynamically specify constraints), while the latter is specific for tree structures and inflexible (it only deals with implicit constraints).

	IceCube	**Harmony**
Object	Application-specific (e.g. XML document, relational table, etc.)	Tree (e.g. XML document, file system, web browser bookmarks, etc.)
Operation Storage	Persistent operations (i.e. actions and constraints are stored in logs)	Transient operations (i.e. update operations are not available)
Operation Relationship	Explicit constraints between actions Constraints dynamically created by the users and reconciliation engine	Implicit constraints on tree structures Constraints are embedded in the reconciliation engine
Propagation	Periodical pull operation transfer	On demand state transfer
Conflict Detection and Resolution	Semantic-based conflict detection Automatic conflict resolution (optimization)	Semantic-based conflict detection Manual conflict resolution (conflicting tree nodes are not reconciled)
Reconciliation	Operation-based semantic approach Takes actions and constraints from several local logs and builds a global schedule that is applied to all replicas	State-based semantic approach Reconciles the corresponding tree nodes of two replicas whose divergent contents do not violate implicit constraints
Consistency	Eventual consistency	No guarantees

Table 5. IceCube *vs.* Harmony

Since Harmony is a state-based reconciler, it detects conflicts between replicas only by comparing their current states, i.e. the operations that have yielded replicas divergent are not available for the reconciliation engine (user intents are unknown). As a result, Harmony does not resolve conflicts; it only reconciles non-conflicting divergences. IceCube is an *operation-based* reconciler; thus, it can access all operations performed on each replica, understand user intents, and try to construct a common sequence of operations. In order to resolve conflicts, IceCube may undo some operations. Therefore, Harmony never undoes user changes, but it does not assure replica convergence. In contrast, IceCube assures that replicas always achieve a common final state, but it may undo some user changes to resolve conflicts.

It is important to note that, in its current version, Harmony is only a framework for reconciling two divergent trees, which offers a programming language and a reconciliation engine. It is not a complete replication protocol (or service), since it does not address the following issues:

- How to manage multiple (more than two) replicas of a tree

- Who should reconcile divergent replicas? A single site (centralized approach) or each involved site (distributed approach)
- When and who should start the reconciliation
- Is Harmony suitable for WANs? Notice that the reconciler must access the entire state of divergent replicas; state transfer of large objects in WANs may raise performance problems.
- How would Harmony behave on failure-prone dynamic environments in which sites can connect and disconnect at any time

Despite these current limitations, the Harmony's framework offers the fundamental components necessary to build a complete replication protocol. Therefore, we consider a generic and flexible solution that assures eventual consistency, as IceCube, more suitable for P2P collaborative applications. However, appropriate adaptations on the specific and inflexible approach of Harmony can rend it equally useful in this context.

3.6 Summary

In this survey we are especially interested in optimistic replication approaches as they provide good properties for dynamic environments in which nodes can connect and disconnect at any time. In order to easily compare different proposals, we have abstracted the main characteristics of optimistic replication solutions by defining 5 parameters. Table 6 summarizes such parameters and presents the solutions we have discussed throughout Section 3.

SYSTEM	OBJ	OPERATION	RELATION	PROPAGATION	CONF	RECONCILIATION	CONSISTENCY
DNS	DB	Transient		Push/pull	None		Temporal
LOCUS	File	Transient	Hb & conc.	Push	C-A	St-b; ordinal	Eventual
TSAE	DB	Persistent	Hb & conc.	Push/pull	None	Op-b; ordinal	Eventual
R&C	File	Persistent	Impl. const.		S-A	Op-b; semantic	Eventual
Unison	File	Transient	Impl. const.	On demand	S-M	St-b; semantic	No guarantees
CVS	File	Persistent	Impl. const.	On demand	C-M	Op-b	Eventual
Harmony	Tree	Transient	Impl. const.	On demand	S-M	St-b; semantic	No guarantees
Bayou	DB	Persistent	Expl. const.	On demand	S-A	Op-b; semantic	Eventual
IceCube	Any	Persistent	Expl. const.	On demand	S-A	Op-b; semantic	Eventual
DLR	Any	Persistent	Expl. const.	On demand	S-A	Op-b; semantic	Eventual

Table 6. Comparing optimistic replication solutions. In column "System", *R&C* stands for Ramsey & Csirmaz's file system and *DLR* stands for Distributed log-based reconciliation. In column "Obj" (object type), *DB* means database. In column "Relation", *Hb* stands for happens-before, *conc.* stands for concurrency, *Impl. const.* stands for implicit constraint, and *Expl. const.* stands for explicit constraint. In column "Conf" (conflict), *C* denotes conflict detection based on concurrency and *S*, conflict detection based on semantic while *A* and *M* denote respectively automatic and manual conflict resolution. Finally, in column "Reconciliation", *St-b* and *Op-b* denotes respectively standard-based and operation-based.

4 P2P Systems

Data management in distributed systems has been traditionally achieved by distributed database systems [OV99] which enable users to transparently access and update several databases in a network using a high-level query language (*e.g.* SQL). Transparency is achieved through a global schema which hides the local databases' heterogeneity. In its simplest form, a *distributed database system* is a centralized server that supports a global schema and implements distributed database techniques (query processing, transaction management, consistency management, etc.). This approach has proved effective for applications that can benefit from centralized control and full-fledge database capabilities, *e.g.* information systems. However, it cannot scale up to more than tens of databases. Data integration systems [TV00, TRV98] extend the distributed

database approach to access data sources on the Internet with a simpler query language in read-only mode. Parallel database systems [Val93] also extend the distributed database approach to improve performance (transaction throughput or query response time) by exploiting database partitioning using a multiprocessor or cluster system. Although data integration systems and parallel database systems can scale up to hundreds of data sources or database partitions, they still rely on a centralized global schema and strong assumptions about the network.

In contrast, P2P systems adopt a completely decentralized approach to resource management. By distributing data storage, processing, and bandwidth across autonomous peers in the network, they can scale without the need for powerful servers. P2P systems have been successfully used for sharing computation, e.g. Seti@home [SWBC+97, Set06] and Genome@home [LSP03, Gen06], communication, e.g. ICQ [Icq06] and Jabber [Jab03], internet service support, e.g. P2P multicast systems [RHKS01, CDKR02, LRSS02, CJKR+03, BKRS+04] and security applications [KR02, JWZ03, VAS04], or data, e.g. Gnutella [Gnu06, JAB01, Jov00], Kazaa [Kaz06] and PeerDB [OST03, SOTZ03]. We focus in this survey on P2P data management. Popular examples of P2P systems such as Gnutella and Kazaa have millions of users sharing petabytes of data over the Internet. Although very useful, these systems are quite simple (*e.g.* file sharing), support limited functions (*e.g.* keyword search) and use simple techniques (*e.g.* resource location by flooding) which have performance problems. In order to overcome these limitations, recent works have concentrated on supporting advanced applications which must deal with semantically rich data (*e.g.* XML documents, relational tables, etc.) using a high-level SQL-like query language, *e.g.* ActiveXML [ABCM+03], Edutella [NWQD+02, NSS03], Piazza [HIMT03, TIMH+03], PIER [HHLT+03]. To deal with the dynamic behavior of peers that can join and leave the system at any time, the P2P systems rely on the fact that popular data get massively duplicated.

In this section we present P2P systems in details. We first introduce and compare the P2P networks that support P2P systems (subsection 4.1). Then, we discuss the main existing solutions for data replication in P2P systems (subsection 4.2).

4.1 P2P Networks

All P2P systems rely on a P2P network to operate. This network is built on top of the physical network (typically the Internet), and therefore is referred to as an *overlay network*. The degree of centralization and the topology of the overlay network tightly affect the nonfunctional properties of the P2P system, such as fault-tolerance, self-maintainability, performance, scalability, and security. For simplicity, we consider three main classes: unstructured, structured, and super-peer networks.

4.1.1 Unstructured

In unstructured P2P networks, the overlay network is created in a nondeterministic (ad hoc) manner and the data placement is completely unrelated to the overlay topology. Each peer knows its neighbors, but does not know the resources they have.

Searching mechanisms can be simple and expensive, such as flooding the network with queries until the desired data is located, or more sophisticated and efficient including the following approaches: (1) Lv et al. [LCCL+02] suggested multiple parallel random walks, where each node chooses a neighbor at random and propagates the request only to it; (2) Yang and Garcia-Molina [YG02] proposed selecting the neighbors to which forward queries based on their past history, as well as the use of local indices for pointing data stored at nodes located within a radius from itself; (3) in [KGZ02], each peer selects a subset of its neighbors to which propagate requests according to their performance in recent queries; (4) the Gia System [CRBL+03] addresses efficiency by dynamically adapting the network topology, so that most nodes are ensured to be at a short distance from high capacity nodes, which are able to provide answers to a very large number of queries; and (5) in [CG02], Crespo and Garcia-Molina use routing indices

to provide a list of neighbors that are most likely to be "in the direction" of the content corresponding to the query.

There is no restriction on the manner to describe the desired data (query expressiveness), i.e. key look-up, SQL-like query, and other approaches can be used. Fault-tolerance is very high since all peers provide equal functionality and are able to replicate data. In addition, each peer is autonomous to decide which data it stores. However, the main problems of unstructured networks are scalability and incompleteness of query results. Searching mechanisms based on flooding are general but do not scale up to a large number of peers. Also, the incompleteness of the results can be high since some peers containing relevant data or their neighbors may not be reached because they are either off-line.

Examples of P2P systems supported by unstructured networks include Gnutella [Jov00, JAB01, Gnu06], Kazaa [Kaz06], and FreeHaven [DFM00]. Since Gnutella is the major representative of this category, it will be described later on.

4.1.2 Structured

Structured networks have emerged to solve the unscalability problem faced by unstructured networks. They achieve this goal by tightly controlling the overlay topology and data placement. Data (or pointers to them) are placed at precisely specified locations and mappings between data and their locations (e.g. a file identifier is mapped to a peer address) are provided in the form of a distributed routing table.

Distributed hash table (DHT) is the main representative of this P2P network class. A DHT provides a hash table interface with primitives put(key,value) and get(key), where key is an object identifier, and each peer is responsible for storing the values (object contents) corresponding to a certain range of keys. Each peer also knows a certain number of other peers, called neighbors, and holds a routing table that associates its neighbors' identifiers to the corresponding addresses. Most DHT data access operations consist of a lookup, for finding the address of the peer *p* that holds the requested object, followed by direct communication with *p*. In the lookup step, several hops may be performed according to nodes' neighborhoods.

Queries can be efficiently routed since the routing scheme allows one to find a peer responsible for a key in *O(log N)*, where *N* is the number of peers in the network. Because a peer is responsible for storing the values corresponding to its range of keys, autonomy is limited. Furthermore, DHT queries are typically limited to exact match keyword search. Active research is on-going to extend the DHT capabilities to deal with more general queries such as range queries and join queries [HHLT+03].

Examples of P2P systems supported by structured networks include Chord [SMKK+01], CAN [RFHK+01], Tapestry [ZHSR+04], Pastry [RD01a], Freenet [CMHS+02], PIER [HHLT+03], OceanStore [KBCC+00], Past [RD01b], and P-Grid [ACDD+03, AHA03]. Freenet is often qualified as loosely structured system because the nodes of its P2P network can produce an estimate (not with certainty) of which node is most likely to store certain object [AS04]. They use a chain mode propagation approach, where each node makes a local decision about to which node to send the request message next. P-Grid is not supported by a DHT either. It is based on a virtual distributed search tree. All these P2P systems are described later on.

4.1.3 Super-peer

Unstructured and structured P2P networks are considered "pure" because all their peers provide equal functionalities. In contrast, super-peer networks are hybrid between client-server systems and pure P2P networks. Like client-server systems, some peers, the *super-peers*, act as dedicated servers for some other peers and can perform complex functions such as indexing, query processing, access control, and meta-data management. Using only one super-peer reduces to client-server with all the problems associated with a single server. Like pure P2P networks, super-peers can be organized in a P2P fashion and communicate with one another in sophisticated ways, thereby allowing the partitioning or replication of global information across all super-

peers. Super-peers can be dynamically elected (e.g. based on bandwidth and processing power) and replaced in the presence of failures.

In a super-peer network, a requesting peer simply sends the request, which can be expressed in a high-level language, to its responsible super-peer. The super-peer can then find the relevant peers either directly through its index or indirectly using its neighbor super-peers.

The main advantages of super-peer networks are efficiency and quality of service (i.e. the user-perceived efficiency, e.g. completeness of query results, query response time, etc.). The time needed to find data by directly accessing indices in a super-peer is quite smaller than flooding the network. In addition, super-peer networks exploit and take advantage of peers' different capabilities in terms of CPU power, bandwidth, or storage capacity as super-peers take on a large portion of the entire network load. In contrast, in pure P2P networks all nodes are equally loaded regardless of their capabilities. Access control can also be better enforced since directory and security information can be maintained at the super-peers. However, autonomy is restricted since peers cannot log in freely to any super-peer. Fault-tolerance is typically low since super-peers are single points of failure for their sub-peers (dynamic replacement of super-peers can alleviate this problem).

Examples of P2P systems supported by super-peer networks include Napster [Nap06], Publius [WAL00], Edutella [NSS03, NWQD+02], and JXTA [Jxt06]. A more recent version of Gnutella also relies on super-peers [AS04]. Napster and JXTA are described later on.

4.1.4 Comparing P2P networks

From the perspective of data management, the main requirements of a P2P network are [DGY03]: autonomy, query expressiveness, efficiency, quality of service, fault-tolerance, and security. We describe these requirements in the following, and then we compare the P2P networks previously discussed based on such requirements.

- **Autonomy**: an autonomous peer should be able to join or leave the system at any time without restriction. It should also be able to control the data it stores and which other peers can store its data, e.g. some other trusted peers.
- **Query expressiveness**: the query language should allow the user to describe the desired data at the appropriate level of detail. The simplest form of query is key look-up which is only appropriate for finding files. Keyword search with ranking of results is appropriate for searching documents. But for more structured data, an SQL-like query language is necessary.
- **Efficiency**: the efficient use of the P2P network resources (bandwidth, computing power, storage) should result in lower cost and thus higher throughput of queries, i.e. a higher number of queries can be processed by the P2P system in a given time.
- **Quality of service**: refers to the user-perceived efficiency of the P2P network, e.g. completeness of query results, data consistency, data availability, query response time, etc.
- **Fault-tolerance**: efficiency and quality of services should be provided despite the occurrence of peers failures. Given the dynamic nature of peers which may leave or fail at any time, the only solution is to rely on data replication.
- **Security**: the open nature of a P2P network makes security a major challenge since one cannot rely on trusted servers. Wrt. data management, the main security issue is access control which includes enforcing intellectual property rights on data contents.

Table 7 summarizes how the requirements for data management are possibly attained by the three main classes of P2P networks. This is a rough comparison to understand the respective merits of each class. For instance, "high" means it can be high. Obviously, there is room for improvement in each class of P2P networks. For instance, fault-tolerance can be made higher in super-peer by relying on replication and fail-over techniques.

Requirements	Unstructured	Structured	Super-peer
Autonomy	high	low	moderate
Query expressiveness	"high"	low	"high"
Efficiency	low	high	high
QoS	low	high	high
Fault-tolerance	high	high	low
Security	low	low	high

Table 7. Comparison of P2P networks

4.2 Replication solutions in P2P systems

P2P systems allow decentralized data sharing by distributing data storage across all peers of a P2P network. Since these peers can join and leave the system at any time, the shared data may become unavailable. To cope with this problem, P2P systems replicate data over the P2P network. In this subsection, we present the main existing P2P systems from the perspective of data management and we discuss the corresponding data replication solutions.

4.2.1 Napster

Napster [Nap06] is a P2P system supported by a super-peer network that relies on central servers to mediate node interactions, as represented in Figure 8. Every *peer* that shares files connects to a *super-peer* and publishes the files it holds. The super-peer, in turn, keeps connection information (e.g. IP address, connection bandwidth) and a list of files provided by each peer. In order to retrieve a file from the overall P2P network, a peer sends a request (noted *query* in Figure 8) to the super-peer, which searches for matches in its index and returns a list of peers that hold the desired file (noted *reply* in Figure 8). The peer that has submitted the query then opens direct connections with one or more peers belonging to the super-peer reply and downloads the desired file.

Napster relies on replication for improving files availability and enhancing performance, but it does not implement a particular replication solution. Indeed, replication occurs naturally as nodes request and copy files from one another. This is referred to as *passive replication*. Napster is simple to implement and efficient for locating files, but it has two main limitations. First, it stores only static data (e.g. music files). Second, super-peers constitute single points of failure and are vulnerable to malicious attack.

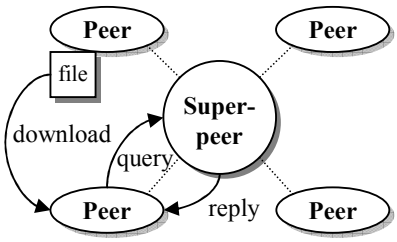


Figure 8. Super-peer network

4.2.2 JXTA

JXTA [Jxt06] is an open source application framework for P2P computing. JXTA protocols aim to establish a network overlay on top of the Internet and non-IP networks, allowing peers to directly interact and self-organize independently of their physical network. JXTA technology leverages open standards like XML, Java technology, and key operating system concepts. By using existing, proven technologies and concepts, the objective is to yield a peer-to-peer system that is familiar to developers.

JXTA's architecture is organized in three layers as shown in Figure 9: JXTA core, JXTA services, and JXTA applications. The core layer provides minimal and essential primitives that are common to P2P networking. The services layer includes network services that may not be absolutely necessary for a P2P network to operate, but are common or desirable in the P2P environment. The applications layer provides integrated applications that aggregate services, and, usually, provide user interface. There is no rigid boundary between the applications layer and the services layer.

In JXTA, all shared resources are described by *advertisements*. Advertisements are language-neutral metadata structures defined as XML documents. Peers use advertisements to publish their resources. Some special super-peers, which are called *rendezvous* peers, are responsible for indexing and locating the advertisements. JXTA does not address data replication.

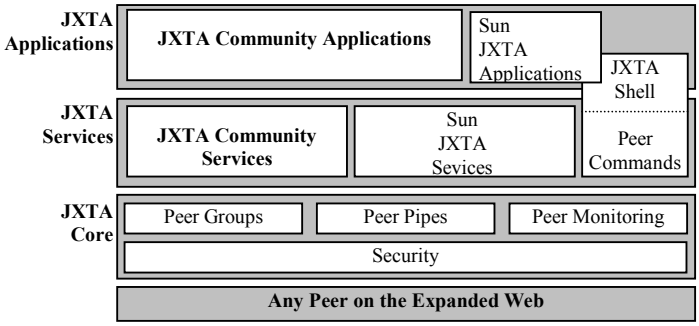


Figure 9. JXTA architecture

4.2.3 Gnutella

Gnutella [Jov00, JAB01, Gnu06] is a P2P file sharing system built on top of the IP network service. Its overlay network is unstructured. In order to obtain a shared file, the node that requests the file (henceforth *requestor*) must perform three tasks: join the Gnutella network, search the desired file, and download it. To join the Gnutella network, the requestor connects to a set of nodes already joined (a bootstrap list is available in databases such as gnutellahosts.com) and sends them a request to announce themselves. Each of these nodes then sends back a message containing its IP and port as well as the number and size of its shared files; in addition, it propagates the announcement request to its neighbors.

Once joined, the requestor can search the desired file as illustrated in Figure 10. In this figure, we use numbers before messages to indicate the time in which they are exchanged (e.g. all messages preceded by 1 are exchanged at the same time t_1). The searching mechanism starts with a query message q sent by the requestor to its neighbors (*1:q* in Figure 10a) and distributed throughout the network by flooding (*2:q* and *3:q* in Figure 10a). Replies to q are routed back along the opposite path through which q arrived. A reply of a host that can satisfy q is called *query hit* (noted *qh*) and contains the IP, port, and speed of the host. When the requestor receives a query hit message (*qh* in Figure 10b), it directly connects to the node that holds the desired file and performs the download. In order to improve efficiency and preserve network bandwidth, duplicated messages are detected and dropped. In addition, the message spread is limited to a maximum number of hops.

As Napster, Gnutella implements passive replication, i.e. a file is only replicated at nodes requesting the file. To improve locality of data, as well as availability and performance, active replication methods were proposed (e.g. [LCCL+02]) in which files may be proactively replicated at arbitrary nodes. However, Gnutella keeps on a major limitation, namely it only deals with static files.

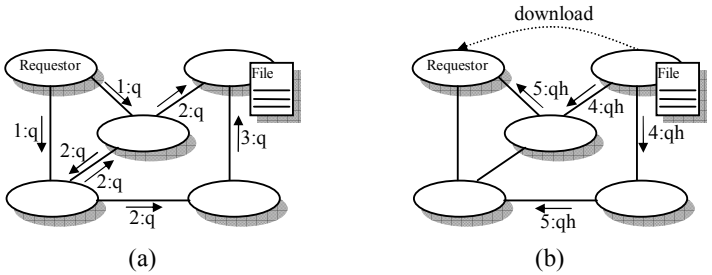


Figure 10. Gnutella: an example of the searching mechanism. (a) The requestor node submits a query *q* that is propagated by flooding. (b) When the requestor receives a query hit *qh*, it connects to the node that holds the desired file and download it.

4.2.4 Chord

Chord [SMKK+01] is a P2P routing and location system on top of a DHT overlay network. Chord uses consistent hashing [KLLL+97] for mapping data keys to nodes responsible for them. The consistent hash function assigns each node and key an *m*-bit *identifier* using a base hash function such as SHA-1 [Fip95]. The identifier length *m* must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. The term "key" is used to refer to both the original key and its image under the hash function, as the meaning is clear from the context. Similarly, the term "node" refers to both the node and its identifier under the hash function.

All node identifiers are ordered in a circle modulo 2^m. Figure 11 shows an example with m = 3 and three connected nodes (0, 1, and 3). Key k is assigned to the first node whose identifier is equal to or follows k in the identifier space. This node is called the *successor* of k and noted *successor(k)*. For instance, in Figure 11 the successor of identifier 1 is node 1, so key 1 should be located at node 1. Similarly, key 2 should be located at node 3, and key 6 at node 0. The use of consistent hashing tends to balance load as each node receives roughly the same number of keys.

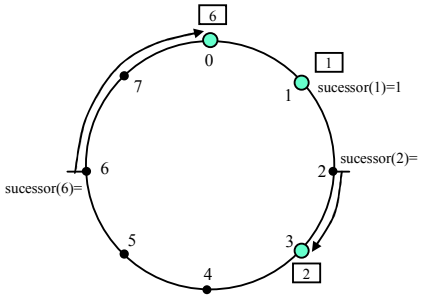


Figure 11. Chord: an example of an identifier circle

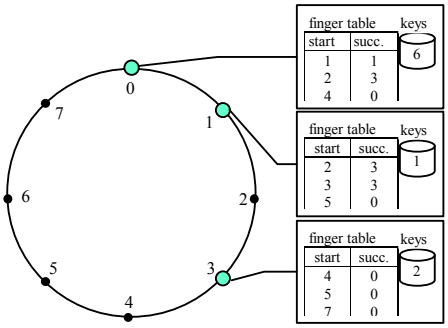


Figure 12. Chord: an example of lookup operation

In order to efficiently locate keys, each node n holds additional routing information in the form of a *finger table*. This table has at most m entries. The i^{th} entry of the n's finger table points to the successor of the identifier $[(n + 2^{i-1}) \bmod 2^m]$, where $1 \le i \le m$. For instance, consider the node 0 (n = 0) in Figure 12. The entries in its finger table are computed as follows:

- $i = 1$: successor $[(0 + 2^0)\ mod\ 2^3] \rightarrow$ successor (1) = 1
- $i = 2$: successor $[(0 + 2^1)\ mod\ 2^3] \rightarrow$ successor (2) = 3
- $i = 3$: successor $[(0 + 2^2)\ mod\ 2^3] \rightarrow$ successor (4) = 0

To illustrate the lookup operation in Chord, let *k* be a searched key. The principle is to find the node that precedes the successor(k), noted *predecessor(k)*, and request from predecessor(k) the identifier of its successor (every node knows its successor and predecessor in the circle). For instance, in Figure 12 consider that node 1 looks for the key *k = 6*, which is stored at node 0. Using the lookup principle, node 1 finds the predecessor(6), which is node 3, and then request its successor; node 3, in turn, replies that its successor is node 0 and terminates the lookup operation. This principle is implemented in practice by accessing the column *succ.* of the finger table (see Figure 12), as follows. The node that starts the query (i.e. *n = 1*) finds in its finger table the node *n'* with the highest identifier such that *n'* is located between *n* and *k* in the circle (i.e. *n'* = 3 since 3 is the highest node identifier in the column *succ.* of node 1's finger table that is located between 1 and 6 in the circle). If such a node exists, the query is forwarded to *n'*, which now becomes *n* and performs the same lookup operation. Otherwise, the node that currently holds the query returns its successor in the circle as the *successor(k)*. Using the finger table, both the amount of routing information held by each node and the time required for resolving lookups are $O(log\ N)$ for a network with *N* connected nodes.

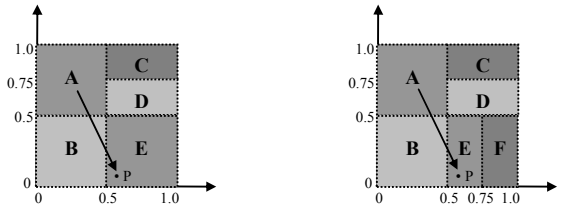
Chord does not implement data replication; it delegates this responsibility for the application. However, it proposes that the application implements replication by storing the object under several keys derived from the data's application level identifier. Knezevic et al. [KWR05] realizes this purpose assuring that in case of concurrent updates on the same replicated object only one peer completes the operation. In addition, missing replicas are proactively recreated within refreshment rounds. This approach gives probabilistic guarantees on accessing correct data at any point in time. Akbarinia et al. [AMPV06a] use multiple hash functions to produce several key identifiers from a single key. They allow updating replicas of the same object in parallel and rely on timestamps to automatically resolve conflicts. This approach provides probabilistic guarantees of consistency among replicas; however, conflicting updates might cause *lost updates*. For instance, consider the scenario where two nodes take in parallel the current version of a given object and update it thereafter. The one that gets the highest timestamp will overwrite the update performed by the other. A problem related to this approach is to determine how many hash functions should be used to replicate an object. Xia et al. [XCK06] discuss this problem and propose a solution. A major limitation of Chord is that the user cannot control data placement.

4.2.5 CAN

CAN (Content Addressable Network) [RFHK+01] relies on a structured P2P network that resembles a hash table. It uses a virtual *d*-dimensional Cartesian coordinate space to store and retrieve (*key, value*) pairs. This coordinate space is completely logical as it is not related to any physical coordinate system. At any point in time, the entire coordinate space is dynamically partitioned among all nodes in the system, so that each node owns a distinct zone that represents a segment of the entire space. Figure 13a shows a 2-dimensional [0, 1] × [0, 1] coordinate space partitioned among 5 nodes. The zone *A*, for instance, is comprised between 0 and 0.5 along the X-axis and between 0.5 and 1 along the Y-axis. To store a pair (k_1, v_1), key k_1 is deterministically mapped onto a point *P* in the coordinate space using a uniform hash function, and then (k_1, v_1) is stored at the node that owns the zone to which *P* belongs. Any node can retrieve the entry (k_1, v_1) by applying the same deterministic hash function to map k_1 onto point *P*. If this point is not owned by the requesting node or its neighbors, the request must be routed through the CAN infrastructure until it reaches the node in whose zone *P* lies. Intuitively, routing in CAN works by following the straight line path through the Cartesian space from source to destination coor-

dinates. For instance, in Figure 13, for node *A* to achieve the point *P*, the corresponding request must be routed through zones *A*, *B*, and *E*.

A CAN node maintains a coordinate routing table that holds the IP address and virtual coordinate zone of each of its neighbors (it is similar to Chord's finger table). Two nodes are neighbors in a *d*-dimensional coordinate space if their coordinate spans overlap along $d - 1$ dimensions and are adjacent along one dimension. For example, in Figure 13a, node *E* is neighbor of nodes *B* and *D*. When a new node joins the system (e.g. node F in Figure 13b), it must take on its own portion of the coordinate space. This is achieved by splitting the zone of an existing node in half and assigning one half to the joining node. In addition, the neighbors of the splitting zone must be notified in order to update their routing tables.



(a) (b)

Figure 13. CAN: (a) A 2-d coordinate space divided into 5 zones; (b) Join operation

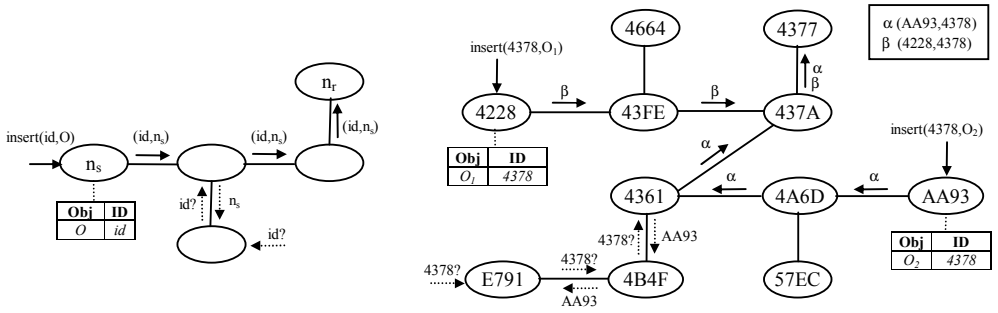
For a *d*-dimensional space partitioned into *N* equal zones, the average routing path length is $(d/4)(N^{1/d})$, and each node holds *2d* neighbors. It means that for a *d*-dimensional space, CAN can grow the number of nodes (and hence zones) without increasing per node state while the path length grows as $O(N^{1/d})$. Notice that, if the number of dimensions is set as $d = (\log_2 N)/2$, CAN could achieve the same properties of other algorithms, such as Chord, i.e. path length $O(\log N)$ and $O(\log N)$ neighbors. However, maintaining the number of neighbors independent of the network size (i.e. *d* independent of *N*) provides better scalability, and it is therefore appropriate for very large networks with frequent topology changes.

Concerning replication, CAN proposes two approaches [RFHK+01]. The first one is to use *m* hash functions to map a single key onto *m* points in the coordinate space, and, accordingly, replicate a single (*key, value*) pair at *m* distinct nodes in the network (similar to Chord's solution). The second approach represents an optimization over the basic design of CAN that consists of node *n* proactively pushing out popular keys towards its neighbors when *n* finds it is being overloaded by requests for these keys. In this approach, replicated keys should have an associated time-to-live field to automatically undo the effect of replication at the end of the overloaded period. In addition, it assumes immutable (*read-only*) contents. Similar to Chord, the main limitation of CAN is that the user cannot control data placement.

4.2.6 Tapestry

Tapestry [ZHSR+04, ZKJ01] is an extensible P2P system that provides decentralized object location and routing on top of a structured overlay network. It routs messages to logical endpoints (i.e. endpoints whose identifiers are not associated with physical location), such as nodes or object replicas. This enables message delivery to mobile or replicated endpoints in the presence of instability in the underlying infrastructure. In addition, Tapestry takes latency into account to establish nodes' neighborhoods. The location and routing mechanisms of Tapestry work as follows. Let *O* be an object identified by *id*. The insertion of *O* in the P2P network involves two nodes: the server node (noted n_s) and the root node (noted n_r). The server node holds *O* while the root node holds a mapping in the format (*id*, n_s) indicating that the object identified by *id* (i.e. *O*) is stored at node n_s. The root node is dynamically determined by a globally consis-

tent deterministic algorithm. Figure 14a shows that when *O* is inserted into n_s, n_s publishes the *O*'s identifier to its root node by routing a message from n_s to n_r containing the mapping (*id*, n_s). This mapping is stored at all nodes along the message path. During a location query (e.g. *id?* in Figure 14a), the message that looks for *id* is initially routed towards n_r, but it may be stopped before achieving n_r once a node containing the mapping (*id*, n_s) is found. For routing a message destined to *id*'s root, each node forwards this message to its neighbor whose logical identifier is the most similar to *id* [PRR97].



(a) Object publishing (b) Replica management

Figure 14. Tapestry: object publishing and replication

Tapestry does not implement object replication directly, but it offers the entire infrastructure needed to take advantage of replicas, as shown in Figure 14b. Each node in the graph represents a peer in the P2P network and contains the peer's logical identifier in the hexadecimal format. In this example, two replicas O_1 and O_2 of the object *O* (e.g. a book file) are inserted into distinct peers ($O_1 \rightarrow$ 4228 and $O_2 \rightarrow$ AA93). The identifier of O_1 is equal to O_2 (i.e. 4378 in hexadecimal) as O_1 and O_2 are replicas of the same object (i.e. *O*). When O_1 is inserted into its server node (i.e. 4228), the mapping (4378, 4228) is routed from node 4228 to node 4377 (the root node for O_1's identifier). Notice that as the message approaches the root node, the object and the node identifiers become more and more similar. In addition, the mapping (4378, 4228) is stored at all nodes along the message path. The insertion of O_2 follows the same procedure. In Figure 14b, if node E791 looks for a replica of *O*, the associated message routing stops at node 4361. Therefore, applications can replicate data across multiple server nodes and rely on Tapestry to direct requests to nearby replicas.

4.2.7 Pastry

Pastry [RD01a] is a P2P infrastructure intended for supporting a variety of P2P applications like global file sharing, file storage, group communication, and naming systems, which is built on top of a structured overlay network. Each node in the Pastry network has a 128-bit node identifier (noted *nodeId*), so that the nodeId space ranges from 0 to $2^{128} - 1$. Node identifiers are ordered in a circle like Chord identifiers. Data placement in Pastry is also similar to Chord, i.e. an object identified by *key* is stored at the node whose nodeId is closest to *key*. Contrasting with Chord, Pastry takes latency into account to establish nodes' neighborhoods. For routing a message that looks for *key*, each node forwards this message to its neighbor whose *nodeId* is the most similar to *key* [PRR97]. Thus, the routing mechanism of Pastry is comparable to the Tapestry's counterpart. In addition, the application is notified at each Pastry node along the message route, and may perform application-specific computations related to the message.

Pastry does not implement object replication directly, but it provides functionalities that enable an application on top of Pastry to easily take advantage of replicas. First, Pastry can route a message that looks for *key* to the *k* nodes whose *nodeIds* are closest to *key*. As a result, a file

storage application, for instance, can assign a *key* to a file (e.g. using a hash function on file's name and owner) and store replicas of this file on the *k* Pastry nodes with *nodeIds* closest to *key*. Second, Pastry's notification mechanisms allow keeping such replicas available despite node failures and node arrivals, using only local coordination among nodes with adjacent nodeIds.

4.2.8 Freenet

Freenet [CMHS+02] is a distributed information storage system focused on privacy and security issues. It does not explicitly try to guarantee permanent data storage. Concerning the underlying P2P network, Freenet is often qualified as loosely structured network because the policies it employs to determine the network topology and data placement are not deterministic.

To add a new file, a user sends an insert message to the system, which contains the file and its assigned location-independent globally unique identifier (GUID). The file is then stored in some set of nodes. During the file's lifetime, it might migrate to or be replicated on other nodes. To retrieve the file, a user sends out a request message containing the GUID key. When the request reaches one of the nodes where the file is stored, that node passes the data back to the request's originator.

Every node in Freenet maintains a routing table that lists the addresses of other nodes and the GUID keys it thinks they hold. When a node receives a query, if it holds the requested file, it returns this file with a tag identifying itself as the data holder. Otherwise, the node forwards the request to the node in its table with the closest key to the one requested, and so forth. If the request is successful, each node in the chain passes the file back upstream and creates a new entry in its routing table associating the data holder with the requested key. Depending on its distance from the holder, each node might also cache a copy locally. An insert message follows the same path that a request for the same key would take, sets the routing table entries in the same way, and stores the file in the same nodes. Thus, new files are placed where queries would look for them.

Data replication occurs as a side effect of search and insert operations. Searches replicate data along the query paths (upstream). In the case of an update (which can only be done by the data's owner) the update is routed downstream based on keys similarities. Since the routing is heuristic and the network may change without notifying peers that come online about the updates they have lost, consistency is not guaranteed.

4.2.9 PIER

PIER [HHLT+03] is a massively distributed query engine built on top of a CAN distributed hash table (DHT). It intends to bring database query processing facilities to widely distributed environments. PIER is a three-tier system organized as shown in Figure 15. Applications (at the higher-level) interact with PIER's Query Processor (at the middle-level) which utilizes an underlying DHT (at the lower-level) for data storage and retrieval. An instance of each DHT and PIER Query Processor component runs on every participating node. The objects stored in the DHT are tuples of relational tables. The object key used by the hash function is composed of three elements: the table name, an attribute of the tuple (usually the primary key), and a random number to uniquely identify objects whose preceding values are equals. PIER does not address replication.

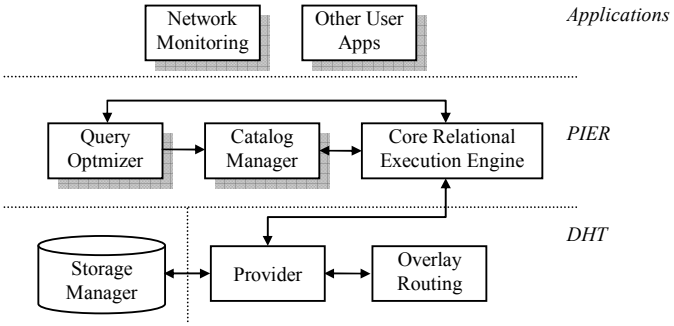


Figure 15. PIER architecture

4.2.10 OceanStore

OceanStore [KBCC+00] is a data management system designed to provide continuous access to persistent information. It relies on Tapestry [ZHSR+04] and assumes an infrastructure composed of untrusted powerful servers, which are connected by high-speed links. For security reasons, data are protected through redundancy and cryptographic techniques. To improve performance, data is allowed to be cached anywhere, anytime.

OcesanStore allows concurrent updates on replicated objects; it relies on reconciliation to assure data consistency and avoid many of the problems inherent with wide-area locking. Figure 16 illustrates the update management in OceanStore. In this example, R is a replicated object whereas R_i and r_i denote respectively a primary and a secondary copy of R. Nodes n_1 and n_2 are concurrently updating R. Such updates are managed as follows. Nodes that hold primary copies of R, henceforth the *master group of R*, are responsible for ordering updates. So, n_1 and n_2 perform tentative updates on their local secondary replicas and send these updates to the master group of R as well as to other random secondary replicas (Figure 16a). The tentative updates are ordered by the master group based on timestamps assigned by n_1 and n_2; at the same time, these updates are epidemically propagated among secondary replicas (Figure 16b). Once the master group obtains an agreement, the result of updates is multicast to secondary replicas (Figure 16c), which contains both tentative[6] and committed data.

Replica management adjusts the number and location of replicas in order to service requests more efficiently. By monitoring the system load, OceanStore detects when a replica is overwhelmed and creates additional replicas on nearby nodes to alleviate load. Conversely, these additional replicas are eliminated when they fall into disuse. Although OceanStore is a very interesting solution, it makes strong assumptions about the network and the capabilities of nodes that are not realistic for P2P environments.

[6] Tentative data is data that the primary replicas have not yet committed.

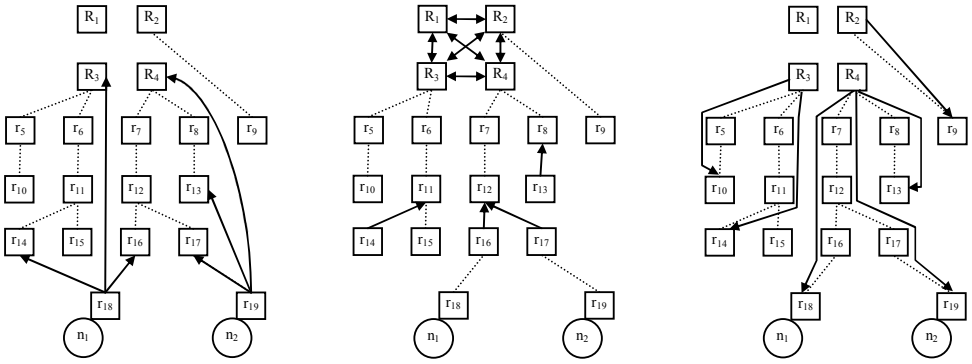


Figure 16. OceanStore: concurrent updates. (a) Nodes n_1 and n_2 send updates to the master group of R and to several random secondary replicas. (b) The master group of R orders updates while secondary replicas propagate them epidemically. (c) After the master group agreement, the result of updates is multicast to secondary replicas.

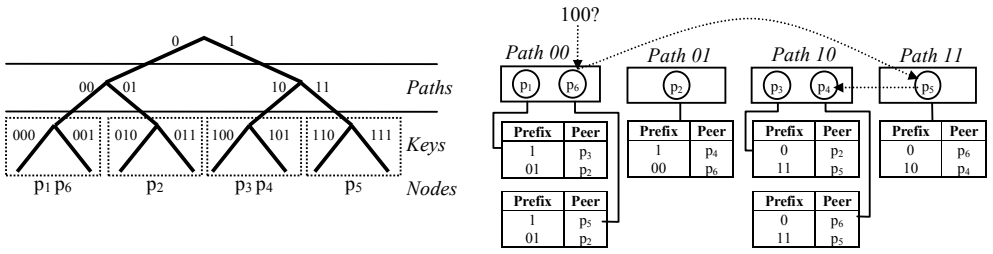
4.2.11 PAST

PAST [RD01b] is a P2P file storage system that relies on Pastry [RD01a] to provide strong persistency and high availability of immutable (*read-only*) files in the Internet. The PAST system offers the following operations: insert, lookup, and reclaim. The *insert* operation stores a file at a user-specified number k of distinct nodes within the PAST network. The *lookup* operation reliably retrieves a copy of the desired file if it exists in PAST and if at least one of the k nodes that store the file is reachable via Internet. The file is normally retrieved from a live node "near" (in terms of latency) the PAST node issuing the lookup. Finally, the *reclaim* operation reclaims the storage occupied by the k copies of a file. Once the operation completes, PAST no longer guarantees the success of lookup operations. Reclaim is different from delete because the file may remain available for a while. Replica management in PAST is based on Pastry's functionalities.

4.2.12 P-Grid

P-Grid [ACDD+03, AHA03] is a peer-to-peer data management system based on a virtual distributed search tree, similarly structured as distributed hash tables. Figure 17a shows a simple example of data placement in P-Grid. In this example, data keys are composed of 3 bits and they are grouped according to their bit prefix. For instance, all keys with prefix 00 (i.e. 000 and 001) belong to the same *path* of the tree (i.e. 00), and therefore are gathered on the same group. Each peer in P-Grid is associated with a tree *path* and is responsible for the group of keys corresponding to this path. For example, in Figure 17a, peers p_1 and p_6 are associated with path 00, and thus hold keys 000 and 001. For fault tolerance, multiple peers can be responsible for the same path (e.g. paths 00 and 10), thereby holding replicas of the same objects.

Figure 17b illustrates query routing in P-Grid. For each bit in the path of a peer p_i, p_i stores a reference to at least one peer that is responsible for the other side of the binary tree at that level. For instance, since p_6 is associated with path 00, p_6 has an entry in its routing table for the prefix 1 (the other side of the tree at first level) and another entry for the prefix 01 (the other side of the tree at second level). Thus, if a peer receives a binary query string that it cannot satisfy, it must forward the query to a peer that is closer to the result. In Figure 17b, p_6 forwards queries starting with 1 to p_5, because p_5 is associated with prefix 1 in the p_6's routing table (first entry). For example, if p_6 receives a query q looking for 100, it forwards q to p_5 that, in turn, forwards q to p_4, which replies q.



(a) Data placement (b) Query routing

Figure 17. P-Grid example

Notice that the peer's path is not associated with the peer's identifier. Indeed, peer paths are acquired and changed dynamically through negotiation with other peers as part of the network maintenance protocol. Thus, a decentralized and self-organizing process builds the P-Grid's routing infrastructure which is adapted to a given distribution of data keys stored by peers. This process also addresses uniform load distribution of data storage and uniform replication of data to support uniform availability.

To address updates of replicated objects, P-Grid employs rumor spreading and provides probabilistic guarantees for consistency. The update propagation scheme has a push phase and a pull phase as described in the following. When a peer *p* receives a new update to a replicated object *R*, *p* pushes the update to a subset of peers that hold replicas of *R* that, in turn, propagate it to other peers holding replicas of *R*, and so forth. Peers that have been disconnected and get connected again, peers that do not receive updates for a long time, or peers that receive a pull request but are not sure to have the latest update, enter the pull phase to reconcile. In this phase, multiple peers are contacted and the most up to date among them is chosen to provide the object content.

The main assumptions of the update algorithm are:

- Peers are mostly offline.
- Conflicts are rare and their resolution is not necessary in general.
- Consecutive updates are distributed sparsely.
- The typical number of replicas is substantially higher than assumed normally for distributed databases but substantially lower than the total network size.
- Replicas within a logical partition of the data space are connected among each other and each replica knows a minimal fraction of the complete set of replicas.
- The connectivity among replicas is high and the connectivity graph is random.

4.2.13 APPA System

APPA (Atlas Peer-to-Peer Architecture) [AMPV06a] is a data management system that provides scalability, availability and performance for P2P advanced applications, which must deal with semantically rich data (e.g. XML documents, relational tables, etc.) using a high-level SQL-like query language. APPA has a layered service-based architecture. Besides the traditional advantages of using services (encapsulation, reuse, portability, etc.), this enables APPA to be network-independent so it can be implemented over different structured (*e.g.* DHT) and super-peer P2P networks. The main reason for this choice is to be able to exploit rapid and continuing progress in P2P networks. Another reason is that it is unlikely that a single P2P network design will be able to address the specific requirements of many different applications. Obviously, different implementations will yield different trade-offs between performance, fault-tolerance, scalability, quality of service, etc. For instance, fault-tolerance can be higher in DHTs because no node is a

single point of failure. On the other hand, through index servers, super-peer networks enable more efficient query processing. Furthermore, different P2P networks could be combined in order to exploit their relative advantages, *e.g.* DHT for key-based search and super-peer for more complex searching.

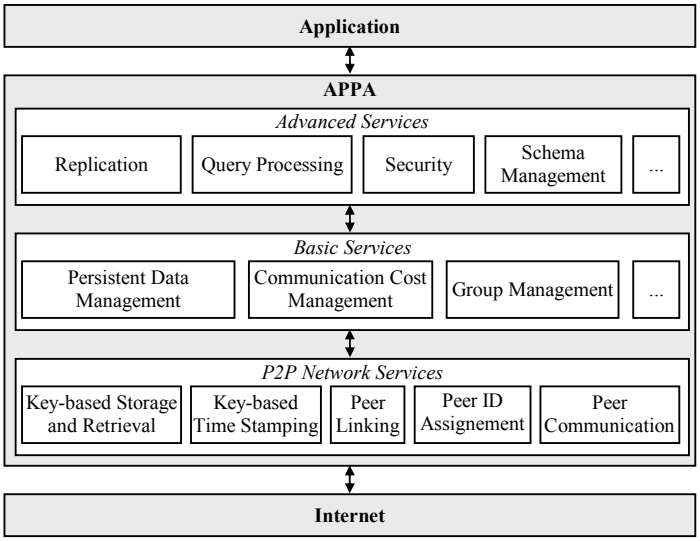


Figure 18. APPA architecture

Figure 18 shows the APPA architecture, which is composed of three layers of services: P2P network services, basic services and advanced services. The P2P network layer provides network independence with services that are common to different P2P networks. The basic services layer provides elementary functionalities for the advanced services using the P2P network layer. The advanced services layer provides advanced services for semantically rich data sharing including schema management, replication [MAPV06, MP06, MPJV06], query processing [AMPV06b, APV06], security, etc. using the basic services.

Data replication in APPA employs the action-constraint framework introduced by IceCube [KRSD01, PSM03, SBK04] to capture the application semantics and resolve update conflicts. However, APPA's approach is quite different from IceCube as it adopts distinctive assumptions and provides innovative solutions. In IceCube, a single centralized node takes update actions from all other nodes for producing a global schedule. This node may be a bottleneck. Moreover, if the reconciler node fails, the whole replication system may be blocked until recovery. In contrast, APPA provides a distributed solution that takes advantage of parallel processing to assure high availability and scalability. It assumes a failure-prone network composed of nodes that can connect and disconnect at any time and copes with this dynamic behavior. APPA also assumes nodes with variable latencies and bandwidths, which implies that data access costs may vary significantly from node to node and have a strong impact on the reconciliation performance. Its reconciliation protocol includes an algorithm for distributed semantic reconciliation as well as a distributed algorithm for selecting the best nodes in terms of communication costs to proceed as reconcilers. APPA implements multi-master replication and assures eventual consistency among replicas.

5 Conclusion

In this survey, we addressed large-scale P2P collaborative applications in which shared data are distributed across peers in the network. Since these peers can join and leave at any time, data

replication is required to provide high availability. From the perspective of the P2P collaborative applications, the replication solution must satisfy the following requirements: data type independency, multi-master replication, semantic conflict detection, eventual consistency, high level of autonomy, and weak network assumptions. These requirements are justified as follows:

- **Data type independency:** different collaborative applications may share different data types (e.g. relational tables, XML documents, files, etc.); thus, the replication solution should be generic wrt. the underlying data type.
- **High level of autonomy:** users that collaborate should be able to store local replicas of the objects they handle in order to maximize data availability. This enables asynchronous collaboration despite disconnections or system failures. A user should also be able to control which other users can store its data.
- **Multi-master replication:** each user that holds a local replica of an object should be able to update it asynchronously. Updates on replicas of the same object should be later reconciled to resolve divergences among replicas.
- **Semantic conflict detection:** asynchronous, parallel updates on different replicas of an object may raise conflicts. By exploiting the operations' semantics, the conflict rate should be reduced.
- **Eventual consistency:** replicas can diverge somewhat, but successive reconciliations should continually reduce the divergence level. In particular, if an object stops to receive updates (e.g. the collaborative edition of an XML document terminates), all its replicas should eventually achieve an equal final state.
- **Weak network assumptions:** users can take advantage of any type of computer to collaborate. In addition, the quality of the underlying network can vary considerably. Thus, the replication solution should not state strong assumptions concerning the physical network (or the infrastructure as a whole, e.g. powerful servers connected by fast and reliable links).

P2P System[7]	**P2P Network**	**Data Type**	**Autonomy**	**Replication Type**	**Conflict Detection**	**Consistency**	**Network Assump.**
Napster	Super-peer	File	Moderate	Static data	–	–	Weak
JXTA	Super-peer	Any	High	–	–	–	Weak
Gnutella	Unstructured	File	High	Static data	–	–	Weak
Chord	Structured	Any	Low	Single-master Multi-master	Concurrency None	Probabilistic Probabilistic	Weak
CAN	Structured	Any	Low	Static data Multi-master	– None	– Probabilistic	Weak
Tapestry	Structured	Any	High	–	–	–	Weak
Pastry	Structured	Any	Low	–	–	–	Weak
Freenet	Structured	File	Moderate	Single-master	None	No guarantees	Weak
PIER	Structured	Tuple	Low	–	–	–	Weak
OceanStore	Structured	Any	High	Multi-master	Concurrency	Eventual	Strong
PAST	Structured	File	Low	Static data	–	–	Weak
P-Grid	Structured	File	High	Multi-master	None	Probabilistic	Weak
APPA	Independent	Any	High	Multi-master	Semantic	Eventual	Weak

Table 8. Comparing replication solutions in P2P systems

[7] For Chord and CAN, we consider the replication approaches explained in Section 4.2.4. Although Tapestry and Pastry provide facilities for managing replicas, they do not implement replication solutions.

Table 8 compares all P2P replication solutions previously discussed based on the requirements stated above. Clearly, the only P2P system that fully satisfies such requirements is APPA. In particular, APPA is the unique system that provides eventual consistency among replicas along with weak network assumptions. The distributed log-based reconciliation algorithms proposed by Chong and Hamadi [CH06] address most of collaborative applications' requirements, but this solution is unsuitable for P2P systems as it does not take into account the dynamic behavior of peers and network limitations. Operational transformation also addresses eventual consistency among replicas, but this approach is specific for collaborative edition and it assumes synchronous collaboration (i.e. concurrent updates of replicas) whereas APPA is application-independent and allows asynchronous collaboration among users.

6 Bibliography

[ABCM+03] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 527-538, San Diego, California, June 2003.

[ACDD+03] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: a self-organizing structured P2P system. *ACM SIGMOD Record*, 32(3):29-33, September 2003.

[AD76] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proc. of the Int. Conf. on Software Engineering*, pages 562-570, San Francisco, California, October 1976.

[AHA03] D. Anwitaman, M. Hauswirth, and K. Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, page 76-85, Washington, District of Columbia, May 2003.

[AL01] P. Albitz and C. Liu. *DNS and BIND*. 4th Ed., O'Reilly, January 2001.

[AMPV06a] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. *Global Data Management* (Chapter Design and implementation of Atlas P2P architecture). 1st Ed., IOS Press, July 2006.

[AMPV06b] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. Top-k query processing in the APPA P2P system. In *Proc. of the Int. Conf. on High Performance Computing for Computational Science (VecPar)*, Rio de Janeiro, Brazil, July 2006.

[APV06] R. Akbarinia, E. Pacitti, and P. Valduriez. Reducing network traffic in unstructured P2P systems using top-k queries. *Distributed and Parallel Databases,* 19(2-3):67-86, May 2006.

[AS04] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335-371, December 2004.

[BG84] P.A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596-615, December 1984.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. 1st Ed., Addison-Wesley, February 1987.

[BKRS+04] A. Bhargava, K. Kothapalli, C. Riley, C. Scheideler, and M. Thober. Pagoda: a dynamic overlay network for routing, data management, and multicasting. In *Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 170-179, Barcelona, Spain, June 2004.

[BKRS+99] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 97-108, Philadelphia, Pennsylvania, June 1999.

[CDKR02] M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron. SCRIBE: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8):1489-1499, October 2002.

[CG02] A. Crespo, and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 23-33, Vienna, Austria, July 2002.

[CH06] Y.L. Chong and Y. Hamadi. Distributed log-based reconciliation. In *Proc. of the European Conference on Artificial Intelligence (ECAI)*, pages 108-112, Riva del Garda, Italy, September 2006.

[CJKR+03] M. Castro, M.B. Jones, A-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Proc. of the Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1510-1520, San Francisco, California, April 2003.

[CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427-469, December 2001.

[CMHS+02] I. Clarke, S. Miller, T.W. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1):40-49, January 2002.

[CP+01] P. Cederqvist, R. Pesch, et al. Version management with CVS. Available at http://www.cvshome.org/docs/manual.

[CPV05] C. Coulon, E. Pacitti, and P. Valduriez. Consistency management for partial replication in a high performance database cluster. In *Proc. of the IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 809-815, Fukuoka, Japan, July 2005.

[CRBL+03] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P systems scalable. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 407-418, Karlsruhe, Germany, August 2003.

[CRR96] P. Chundi, D.J. Rosenkrantz, and S.S. Ravi. Deferred updates and data placement in distributed databases. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 469-476, New Orleans, Louisiana, February 1996.

[DFM00] R. Dingledine, M. Freedman, and D. Molnar. The FreeHaven project: distributed anonymous storage service. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, pages 67-95, Berkeley, California, July 2000.

[DGY03] N. Daswani, H. Garcia-Molina, and B. Yang. Open problems in data-sharing peer-to-peer systems. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, pages 1-15, Siena, Italy, January 2003.

[EG89] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 399-407, Portland, Oregon, May 1989.

[ES83] D.L. Eager and K.C. Sevcik. Achieving robustness in distributed database systems. *ACM Transactions on Database Systems*, 8(3):354-381, September 1983.

[ET89] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264-290, June 1989.

[FGMP+05] J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proc. of the ACM Symp. on Principles of Programming Languages (POPL)*, pages 233-246, Long Beach, California, January 2005.

[Fid88] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of the Australian Computer Science Conference*, pages 55-66, University of Queensland, Australia, February 1988.

[Fip95] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, Virginia, April 1995.

[FVC04] J. Ferrié, N. Vidot, M. Cart. Concurrent undo operations in collaborative environments using operational transformation. In *Proc. of the Int. Conf. on Cooperative Information Systems (CoopIS)*, pages 155-173, Agia Napa, Cyprus, October 2004.

[Gen06] Genome@Home. http://genomeathome.stanford.edu/.

[GHOS96] J. Gray, P. Helland, P.E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 173-182, Montreal, Canada, June 1996.

[Gif79] D.K. Gifford. Weighted voting for replicated data. In *Proc. of the ACM SIGOPS Symp. on Operating Systems Principles (SOSP)*, pages 150-162, Pacific Grove, California, December 1979.

[Gnu06] Gnutella. http://www.gnutelliums.com/.

[Gol92] R.A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California, Santa Cruz, California, December 1992.

[GSC+83] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, and D.R. Ries. A recovery algorithm for a distributed database system. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, pages 8-15, Atlanta, Georgia, March 1983.

[Har06] Harmony. http://www.seas.upenn.edu/~harmony/.

[HHLT+03] R. Huebsch, J. Hellerstein, N. Lanham, B. Thau Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. of Int. Conf. on Very Large Databases (VLDB)*, pages 321-332, Berlin, Germany, September 2003.

[HIMT03] A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *Proc. of the Int. World Wide Web Conference (WWW)*, pages 556-567, Budapest, Hungary, May 2003.

[Icq06] ICQ. http://www.icq.com/.

[JAB01] M. Jovanovic, F. Annexstein, and K. Berman. Scalability issues in large peer-to-peer networks: a case study of Gnutella. Technical report, ECECS Department, University of Cincinnati, Cincinnati, Ohio, January 2001.

[Jab03] Jabber. http://www.jabber.org/.

[JM87] S. Jajodia and D. Mutchler. Dynamic voting. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 227-238, San Francisco, California, May 1987.

[Jov00] M. Jovanovic. Modelling large-scale peer-to-peer networks and a case study of Gnutella. Master's thesis, Department of Electrical and Computer Engineering and Computer Science, University of Cincinnati, Cincinnati, Ohio, June 2000.

[JPAK03] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257-294, September, 2003.

[JWZ03] R. Janakiraman, M. Waldvogel, and Q. Zhang. Indra: a peer-to-peer approach to network intrusion detection and prevention. In *Proc. of the IEEE Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, page 226-231, Linz, Austria, June 2003.

[Jxt06] JXTA. http://www.jxta.org/.

[KA00] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333-379, September, 2000.

[Kaz06] Kazaa. http://www.kazaa.com/.

[KBCC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao. OceanStore: an architecture for global-scale persistent storage. In *Proc. of the ACM*

Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 190-201, Cambridge, Massachusetts, November 2000.

[KGZ02] V. Kalogeraki, D. Gunopoulos, D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proc. of the ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 300-307, McLean, Virginia, November 2002.

[KLLL+97] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of the ACM Symp. on Theory of Computing,* pages 654-663, El Paso, Texas, May 1997.

[KR02] A.V.M. Keromytis and D. Rubenstein. SOS: secure overlay services. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 61-72, Pittsburgh, Pennsylvania, August 2002.

[KRSD01] A-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of diverging replicas. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)*, pages 210-218, Newport, Rhode Island, August 2001.

[KWR05] P. Knezevic, A. Wombacher, and T. Risse. Enabling high data availability in a DHT. In *Proc. of the Int. Workshop on Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems (GLOBE'05)*, pages 363-367, Copenhagen, Denmark, August 2005.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, July 1978.

[LCCL+02] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. of the ACM Int. Conf. on Supercomputing (ICS)*, pages 84-95, New York, New York, June 2002.

[LKPJ05] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Consistent data replication: is it feasible in WANs? In *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, pages 633-643, Lisbon, Portugal, September 2005.

[LRSS02] K. Lakshminarayanan, A. Rao, I. Stoica, and S. Shenker. Flexible and robust large scale multicast using i3. Technical report CSD-02-1187, University of California, Berkeley, California, June 2002.

[LSP03] S. Larson, C. Snow, and V. Pande. *Modern Methods in Computational Biology*. (Chapter Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology). Horizon Press, 2003.

[MAPV06] V. Martins, R. Akbarinia, E. Pacitti, and P. Valduriez. Reconciliation in the APPA P2P system. In *Proc. of the IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 401-410, Minneapolis, Minnesota, July 2006.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Proc. of the Int. Workshop on Parallel and Distributed Algorithms*, pages 216-226, Elsevier Science Publishers B.V., North-Holland, October 1989.

[MOSI03] P. Molli, G. Oster, H. Skaf-Molli, A. Imine. Using the transformational approach to build a safe and generic data synchronizer. In *Proc. of the ACM SIGGROUP Int. Conf. on Supporting Group Work (GROUP)*, pages 212-220, Sanibel Island, Florida, November 2003.

[MP06] V. Martins and E. Pacitti. Dynamic and distributed reconciliation in P2P-DHT networks. In *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, pages 337-349, Dresden, Germany, September 2006.

[MPJV06] V. Martins, E. Pacitti, R. Jimenez-Peris, and P. Valduriez. Scalable and available reconciliation in P2P networks. In *Proc. of the Journées Bases de Données Avancées (BDA)*, Lille, France, October 2006.

[Nap06] Napster. http://www.napster.com/.

[NSS03] W. Nejdl, W. Siberski, and M. Sintek. Design issues and challenges for RDF- and schema-based peer-to-peer systems. *ACM SIGMOD Record*, 32(3):41-46, September 2003.

[NWQD+02] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. Edutella: a P2P networking infrastructure based on RDF. In *Proc. of the Int. World Wide Web Conference (WWW)*, pages 604-615, Honolulu, Hawaii, May 2002.

[OST03] B. Ooi, Y. Shu, and K-L. Tan. Relational data sharing in peer-based data management systems. *ACM SIGMOD Record*, 32(3):59-64, September 2003.

[OV99] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. 2nd Ed., Prentice Hall, January 1999.

[PC98] C. Palmer and G. Cormack. Operation transforms for a distributed shared spreadsheet. In Proc. of the ACM Int. Conf. on Computer Supported Cooperative Work (CSCW), pages 69-78, Seattle, Washington, November 1998.

[PL88] J.F. Pâris and D.E. Long. Efficient dynamic voting algorithms. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 268-275, Los Angeles, California, February 1988.

[PMS99] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 126-137, Edinburgh, Scotland, September 1999.

[PPRS+83] D.S. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. In *IEEE Transactions on Software Engineering*, 9(3):240-247, May 1983.

[PRR97] C.G. Plaxton, R. Rajaraman, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the ACM Symp. on Parallel Algorithms and Architectures* (*SPAA)*, pages 311-320, Newport, Road Island, June 1997.

[PS00] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3-4):305-318, 2000.

[PSG04] B.C. Pierce, A. Schmitt, and M.B. Greenwald. Bringing Harmony to optimism: an experiment in synchronizing heterogeneous tree-structured data. Technical report MS-CIS-03-42, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, February 2004.

[PSM03] N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. of the Int. Conf. on Cooperative Information Systems (CoopIS)*, pages 38-55, Catania, Italy, November 2003.

[PSM98] E. Pacitti, E. Simon, and R.N. Melo. Improving data freshness in lazy master schemes. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 164-171, Amsterdam, The Netherlands, May 1998.

[PSTT+97] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, pages 288-301, St. Malo, France, October 1997.

[PV04] B.C. Pierce and J. Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical report MS-CIS-03-36, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, February 2004.

[RC01] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. In *Proc. of the ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE)*, pages 175-185, Vienna, Austria, September 2001.

[RD01a] A. Rowstron and P. Druschel. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware)*, pages 329-350, Heidelberg, Germany, November 2001.

[RD01b] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, pages 188-201, Banff, Canada, October 2001.

[RFHK+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages: 161-172, San Diego, California, August 2001.

[RHKS01] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proc. of the Int. Workshop on Networked Group Communication (NGC)*, pages 14-29, London, United Kingdom, November 2001.

[SBK04] M. Shapiro, K. Bhargavan, N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. of the Int. Conf. on Principles of Distributed Systems (OPODIS)*, Grenoble, France, December 2004.

[SE98] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of the ACM Int. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 59-68, Seattle, Washington, November 1998.

[Set06] Seti@home. http://setiathome.ssl.berkeley.edu.

[SJZY+98] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63-108, March 1998.

[SMKK+01] I. Stoica, R. Morris, D.R. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149-160, San Diego, California, August 2001.

[SOTZ03] W. Siong Ng, B. Ooi, K-L. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, March 2003.

[SS05] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42-81, March 2005.

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed Ingres. *IEEE Transactions on Software Engineering*, 5(3):188-194, May 1979.

[SWBC+97] W. Sullivan III, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on project Serendip data and 100,000 personal computers. In *Proc. of the Int. Conf. on Bioastronomy*, Bologna, Italy, 1997.

[SYZC96] C. Sun, Y. Yang, Y. Zhang, and D. Chen. A consistency model and supporting schemes for real-time cooperative editing systems. In *Proc. of the Australian*

Computer Science Conference, pages 582-591, Melbourne, Australia, January 1996.

[Tho79] R.H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180-209, June 1979.

[TIMH+03] I. Tatarinov, Z.G. Ives, J. Madhavan, A. Halevy, D. Suciu, N. Dalvi, X. Dong, Y. Kadiyska, G. Miklau, and P. Mork. The Piazza peer data management project. *ACM SIGMOD Record*, 32(3):47-52, September 2003.

[TRV98] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808-823, September 1998.

[TTPD+95] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, pages 172-183, Cooper Mountain, Colorado, December 1995.

[TV00] A. Tanaka and P. Valduriez. The Ecobase environmental information system: applications, architecture and open issues. *ACM SIGMOD Record, 3(5-6)*, 2000.

[Uni06] Unison. http://www.cis.upenn.edu/~bcpierce/unison/.

[Val93] P. Valduriez. Parallel database systems: open problems and new issues. *Distributed and Parallel Databases*, 1(2):137-165, April 1993.

[VAS04] V. Vlachos, S. Androutsellis-Theotokis, and D. Spinellis. Security applications of peer-to-peer networks. *Computer Networks Journal*, 45(2):195-205, June 2004.

[VCFS00] N. Vidot, M. Cart, J. Ferrie, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proc. of the ACM Int. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 171-180, Philadelphia, Pennsylvania, December 2000.

[Ves03] J. Vesperman. *Essential CVS*. 1st Ed., O'Reilly, June 2003.

[WAL00] M. Waldman, R. AD, and C. LF. Publius: a robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. of the USENIX Security Symposium*, pages 59-72, Denver, Colorado, August 2000.

[WPEK+83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, pages 49-70, Breton Woods, New Hampshire, October 1983.

[XCK06] Y. Xia, S. Chen, and V. Korgaonkar. Load balancing with multiple hash functions in peer-to-peer networks. In *Proc. of the IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 411-420, Minneapolis, Minnesota, July 2006.

[YG02] B. Yang, H. Garcia-Molina. Improving search in peer-to-peer networks. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 5-14., Vienna, Austria, July 2002.

[ZHSR+04] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications (JSAC),* 22(1):41-53, January 2004.

[ZKJ01] B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: an infrastructure for fault-tolerant wide-area location and routing. Technical Report CSD-010-1141, University of California, Berkeley, California, April 2001.