UPDATES AND TRANSACTIONS IN PEER-TO-PEER SYSTEMS

Zachary Ives
Computer and Information Science Department
University of Pennsylvania
Philadelphia, PA 19104-6389
zives@cis.upenn.edu

SYNONYMS

Consistency in peer-to-peer systems; Update propagation in peer-to-peer systems

DEFINITION

In recent years, work on peer-to-peer systems has begun to consider settings in which data is updated, sometimes in the form of atomic transactions, and sometimes by parties other than the original author. This raises many of the issues related to enforcing consistency using concurrency control or other schemes. While such issues have been addressed in many ways in distributed systems and distributed databases, the challenge in the peer-to-peer context is in performing the tasks cooperatively, and potentially in tolerating some variation among the instances at different nodes.

HISTORICAL BACKGROUND

Early peer-to-peer systems focused on sharing or querying immutable data and/or files. More modern uses of peer-to-peer technology consider settings in which dynamic data and updates are being made in a distributed, autonomous context. A question of significant interest is how to define consistency in this model, while still allowing at least some autonomy among the sites.

SCIENTIFIC FUNDAMENTALS

The database community has only recently begun to consider peer-to-peer architectures for applications beyond query answering, with recent work in peer data exchange [10] and collaborative data sharing [16]. However, a popular model has been that of *distributed stream processing*, in which streams of data (usually interpreted as insertions of new data) are processed in a distributed or peer-to-peer network [1, 7]. Additionally, work in the distributed systems community, particularly on file systems, considers many issues related to distributed replication and consistency (with the granularity of updates typically being at the level of the file, or custom to an application). (See also the entry on *peer-to-peer storage.*)

The work on updates and transactions in peer-to-peer systems can be classified based on who is allowed to modify it, and how conflicting modifications are resolved. This can be divided into the following categories:

Single owner/primary copy is a setting in which each data item that originates from some source peer p can *only* be modified by (or through) p — i.e., no other peers are allowed to directly modify that data.

Owner-resolver protocols allow multiple peers to modify the data, and they typically rely on the owner to resolve any conflicts. If resolution is impossible, they "branch" the data into fully independent instances.

Consensus protocols allow multiple peers to modify the data, and some set of nodes works together to determine how to arbitrate for consistency.

Partial divergence schemes handle conflicts in a way that results in multiple divergent copies of the data, but they operate at a finer level of granularity than divergent replica protocols, and they allow some portions of the data instance to remain shared even after “branching” two instances.

The remainder of this article provides more detail on the different approaches and their common implementations.

Single-Owner/Primary Copy

In the simplest schemes, each data item is owned by a single source, which may update that data. Many other nodes may replicate the data but may not change it (except, perhaps, by going through the *primary copy* at the owner). This is sometimes referred to as the *single-writer, multiple readers* problem. In this type of scheme, the owner of the data uses a timestamp (logical or physical) to preserve the serial order of updates, or to arbitrate among different verions of the data. Since there is a single owner and a single clock, any node can look at the data and deterministically choose an ordering.

In the database world, peer-to-peer stream processing or filtering systems, e.g., ONYX [7] and PIER [11], can be considered systems in which each data item (tuple or XML tree) has a single owner: when new data arrives on the stream, this appends to or replaces the previous data items from the same source.

P2P file systems have also employed a single-owner scheme that provides very interesting properties. In particular, CFS [5], PAST [8], and similar systems build a filesystem layer over distributed hash tables (DHash and Pastry, respectively), where every version of the file is maintained in the distributed hash table. In these approaches, a user must ask for a file by name *and version*; the latest version can be obtained from the file’s owner or from a trusted third party. This model is exemplified by CFS, which retrieves the file as follows: first, it cryptographically hashes the filename and version, receiving a key from which the file’s current block map can be fetched. This map contains an ordered list of keys corresponding to the individual blocks in the requested version of the file. Each block can be fetched using its key; the key actually represents the hash of the block’s content. This content-based hashing scheme for blocks has a two key benefits: (1) if two files or file versions share a page with identical content, CFS will store only **one copy** of the page and will use it in both files; (2) CFS will employ caching as blocks are requested, and the cache lookup mechanism can be based on hashing, rather than naming, content.

Owner-Resolver

Coda [12] relaxes the single-owner scheme described above, in allowing data to be replicated throughout a network, and for changes to be made to the replicas. Coda’s focus is on allowing updates in the presence of network partition: nodes might need to make changes without having access to the primary copy. Once connectivity is restored, the newly modified replica must be reconciled with the original data and any other changed replicas; Coda does this by sharing and replaying logs of changes made to the different replicas. If Coda determines that multiple concurrent changes were made, then activates an application-specific *conflict resolver* that attempts to resolve the conflicts. In the worst case, the data may need to be branched.

Bayou [9] uses a very similar scheme, except that changes to replicas are propagated in pairwise fashion across the network (an *epidemic protocol*) instead of sent directly to each file’s owner. Nodes maintain logs of the updates they know about, including other nodes’ updates; as they communicate, they exchange logs and merge them in pairwise fashion (this is an *epidemic protocol*). If conflicts occur, an application-specific *merge procedure* is triggered. Eventually the logs reach a primary node, which determines the final order of updates.

Building even further on the notion of epidemic protocols, work by Datta et al. [6] focuses on settings in which conflicts are unlikely to arise at all. It adopts an epidemic protocol that provides eventual consistency across the network. In this model, a peer that makes an update pushes a notification to subset of its neighboring peers, who may in turn forward to additional peers. This is likely to keep many of the peers “relatively” up to date. When a peer has not received an update in a while, or if it comes back online, it tries to determine the latest state by executing a “pull” request.

Consensus

Coda and Bayou allowed for concurrent updates, but relied on the holder of the primary copy to resolve conflicts. An alternative is to reconcile conflicts through some sort of voting or consensus scheme. Like Bayou, Deno [4] uses an epidemic protocol, in which nodes share information in pairwise fashion about new updates. Here, updates are grouped into transactions that are to be atomically committed or aborted. For each transaction that is not *blocked* (i.e., waiting for another transaction to complete), a distributed vote is executed to determine whether the transaction should be committed. Nodes gossip by exchanging information about transactions and votes; ordering information is maintained using version vectors [15]. If a majority of nodes vote for the transaction, rather than any other transactions that have conflicting updates, then the transaction is committed.

A number of filesystems, including BFS [3], OceanStore [13], Farsite [2], make use of *quorums* of nodes that manage the sequencing on updates: these nodes essentially serve very similarly to the single owner schemes described previously, in that they must be contacted for each update in sequence, and they define the serialization order of updates.

Partial Divergence

Two more recent works — one focused on filesystems, and the other on database instances — enable a scheme for managing inconsistency based on peers' individual *trust policies.*

Like CFS, Ivy [14] is a filesystem built over the DHash distributed hash table. However, Ivy provides NFS-like semantics including the ability to modify files, and it does so in a novel way. Ivy has one update log per peer, and such logs are made publicly available through DHash. As a peer modifies a file, it writes these changes to its own update log (annotating them with version vectors [15] so sequencing can be tracked). As the peer *reads* a file, it may consult *all* logs; but it may also ignore some logs, depending on its local trust policy. However, Ivy assumes that it is highly undesirable for each peer to get a different version of a file (because this would prevent sharing); hence, sets of peers share a *view* of the file system — all files in the view have the same consistent version.

Finally, the Orchestra [16] *collaborative data sharing system* focuses on sharing different database instances in loose collaborations: here, multiple peers wish to share updates with one another, but each peer may selectively override the updates it receives from elsewhere. Each peer may adopt its own *trust policies* specifying (in a partial order) how much it trusts the other peers. Orchestra is specifically motivated by scientific data sharing: for instance, organizations holding proteomics and genomics data wish to import data from one another, and to refresh this imported data; however, since the data is often unreliable, each peer may wish to choose the version from the site it trusts most, and then independently curate (revise, correct, and annotate) the resulting data.

In Orchestra, as in Deno, all updates are grouped into atomic transactions. Here, each peer uses its local trust policies to choose among conflicting transactions — such a transaction will be *accepted* by the peer, and all conflicting transactions will be *rejected.* If a transaction X from a particular peer is rejected, then all subsequent transactions from that same peer that directly modified the results of X are also transitively rejected. In this trust model, the common case is that transactions come to a target peer C from a trusted peer and does not conflict with any others; hence they are immediately applied to C. Otherwise, trust composes as follows: as a transaction is propagated from peer A to peer B to peer C, it is only applied at C if it is the most trusted transaction at each step; if multiple conflicting transactions can be applied at C, then the one most trusted by C is applied. Orchestra allows instances to "partially diverge" where there are conflicts, while still maintaining sharing for portions of the data where there are no conflicts.

KEY APPLICATIONS

Recent applications of peer-to-peer technologies include distributed network monitoring, distributed stream processing, and exchange of data among collaborating organizations that host scientific databases. In all of these settings, data may be frequently updated.

FUTURE DIRECTIONS

One of the most promising directions of future study is the interaction between transactions and *mappings* or conversion routines: in many real-world applications, data is being shared between sites that have different data representations. There is some work (e.g., that related to view maintenance) that shows how to translate updates across mappings; but there is no scheme for mapping *transactions.*

CROSS REFERENCE

Distributed databases.
Distributed concurrency control.
Transactions.
Epidemic algorithms.
Peer-to-peer storage.

RECOMMENDED READING

[1] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Demonstration: Load management and high availability in the Medusa distributed stream processing system. In *SIGMOD*, 2004.

[2] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proc. Measurement and Modeling of Computer Systems, 2000*, June 2000.

[3] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4), 2002.

[4] Ugur Cetintemel, Peter J. Keleher, Bobby Bhattacharjee, and Michael J. Franklin. Deno: a decentralized, peer-to-peer object-replication system for weakly connected environments. *Transactions on Computers*, 52(7), Jul 2003.

[5] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[6] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, Washington, DC, USA, 2003.

[7] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an Internet-scale XML dissemination service. In *VLDB*, 2004.

[8] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. *HotOS*, 00, 2001.

[9] W. Keith Edwards, Elizabeth D. Mynatt, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, and Marvin M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *UIST '97*, 1997.

[10] Ariel Fuxman, Phokion G. Kolaitis, Renée J. Miller, and Wang-Chiew Tan. Peer data exchange. In *PODS*, 2005.

[11] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Quering the Internet with PIER. In *VLDB*, 2003.

[12] J. Kisler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1), 1992.

[13] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS 2000*, November 2000.

[14] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *OSDI*, 2002.

[15] D. Storr Parker, Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser, and Charles S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng.*, 9(3), 1983.

[16] Nicholas E. Taylor and Zachary G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, 2006.