Fast Encounter-based Synchronization for Mobile Devices

Daniel Peek[1]
Microsoft Research

Venugopalan Ramasubramanian
Microsoft Research

Thomas L. Rodeheffer
Microsoft Research

Douglas B. Terry
Microsoft Research

Meg Walraed-Sullivan[2]
Microsoft Research

Ted Wobber
Microsoft Research

Abstract

The large and growing number of computing devices used by individuals has caused the challenges of distributed storage to take on increased importance. In addition to desktops and laptops, portable devices, such as cell phones, digital cameras, iPods, and PDAs are capable of storing and sharing data. These devices' mobility coupled with wireless networking capabilities allows them to opportunistically propagate data to other devices they might encounter, even if connectivity is unplanned and transient. To take advantage of such ad hoc connectivity, devices must have an efficient method for determining which files they hold in common and which versions must be propagated to achieve consistency. This paper presents new techniques for reducing the cost of this data synchronization operation by over an order of magnitude in many situations.

1. Introduction

Going to school on any given morning, an intrepid graduate student can expect to encounter not only his or her own iPod, cell phone, laptop, and desktop, but also the devices and computers of other students, staff, and professors. All of these casual encounters offer opportunities for wireless communication between devices and can provide the substrate for an extremely useful distributed storage system in which files rapidly propagate between interested parties through direct exchanges.

This style of communication enables many applications that are not well served by traditional distributed file systems. Allowing the receipt of updates from third parties permits uses such as receiving lecture notes from other students who have a more recent version simply by being in proximity to one of them, sharing project files with other project members at a group meeting, and exchanging photos taken during a conference.

Ordinarily, however, little communication takes place between nearby, unfamiliar devices. An initial barrier to participation is that the devices must have suitable communication hardware as well as software for naming and locating devices and their files. These issues are rapidly being addressed since modern portable devices routinely come with wireless networking hardware, such as Bluetooth and WiFi, and discovery and naming protocols have been devised [2][8][17].

Even after discovery and communication problems are solved, another fundamental problem arises that is the focus of this paper. How can devices that encounter each other efficiently determine whether they store common files and, if so, whether one device holds updated versions of such files? The cost of determining whether this is the case for any pair of devices can be very large in terms of bandwidth, time, and energy.

We have devised techniques that can greatly reduce the cost of determining which files must be transferred to synchronize two devices. Reducing this cost improves the ratio of useful data transferred to resources expended. Given that this cost is paid on each encounter between devices, keeping it to a minimum is critical, especially since most encounter-based synchronizations will not result in any actual file transfers. Reduced cost, especially in the case where no updates need to be transmitted, allows devices that perform periodic synchronization to check for updates more frequently. This not only bounds inconsistency between devices but also decreases the window of vulnerability to conflicts in which a user could modify a file before receiving a previous update of the same file.

This paper explores a variety of optimization techniques that can be applied to existing file synchronization protocols. We present a new protocol that combines the optimizations, reducing

[1] Current affiliation: Department of Electrical Engineering and Computer Science, University of Michigan

[2] Current affiliation: Department of Computer Science and Engineering, University of California, San Diego

1-4244-1476-8/07/$25.00 ©2007 IEEE.

synchronization overhead by more than an order of magnitude in a sample music sharing scenario.

2. Device synchronization

To synchronize two devices during an encounter, updated files must be transferred from the device that has them to the device that wants them. Methods for encounter-based synchronization must take into account the important characteristics of mobile devices and their operating environment. Although many devices can now store hundreds of thousands of files, the set of files that might be shared will be vastly larger than the capacity of any single device. Furthermore, device owners have varied interests, implying that the overlap in files between nearby devices may be small. Thus, synchronization protocols that assume full replication of a large file space or that assume the maintenance of complete update logs are impractical for mobile devices. Consequently, devices must communicate lists of stored (or at least recently updated) files along with versioning information as the first step of the synchronization process.

Commonly, per-file version vectors [13] are employed for tracking the versions of files and determining the relationship between versions of the same file. They allow each file to be updated and propagated independently while reliably detecting whether updates to a file were performed concurrently and hence conflict. Version vectors do this by associating a unique identifier and per-writer update counts with each file. A device need only store version vectors for files currently stored by that device, a critical property when the number of files in the system is large and device storage space is precious.

An example of a conventional version vector protocol similar to the one used by EnsemBlue [14] and Ficus [9] is shown in Figure 1. In this protocol, when one device, such as the iPod wishes to synchronize its data with another device, such as a laptop, the iPod simply transmits the version vectors and unique file identifiers of all the files it contains. The laptop then compares the version vectors it has received with the version vectors of the files it stores.

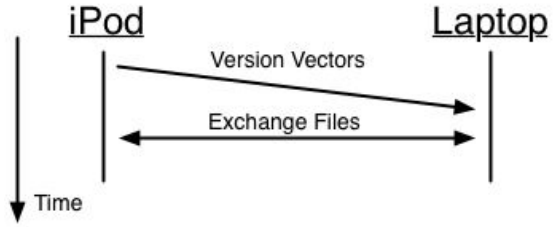

Figure 1. Conventional Version Vector Protocol

If only one of the devices has a version vector for a file identifier, then only one of the devices contains the file and no action is required. If both devices have a version vector with the same file identifier, then we must compare the per-writer update counts. If the counts are equal for all version vector entries, then the file versions are the same. If, for all entries, the update counts in one version vector are greater than or equal to the update counts for the same writer in the other version vector, then the former version vector is associated with a more recent version of the file and it must be transmitted. Otherwise, the version vectors indicate that a concurrent update has occurred and neither of the versions is more recent than the other; in this case, systems may rely on automatic conflict resolvers to produce new, merged file data but such techniques are orthogonal to the topic of this paper.

3. Optimizations

The conventional version vector protocol is often monstrously inefficient because it always requires a device to transmit all of its version vectors to complete the synchronization process. With devices that can store over 100,000 files and version vectors that can be dozens of bytes in size, this protocol can require megabytes of data to be transmitted to determine which updates are needed. So, we have developed several techniques to reduce the size and number of version vectors transmitted in practice. In combination, our techniques can reduce the total transmission required by over an order of magnitude in many cases, such as the example in Section 4.

3.1. Shrinking version vectors

In a straightforward implementation, version vectors consist of a system-wide unique file identifier followed by, for each device that has updated the file, hereafter called a “client”, the device's system-wide unique client identifier and an update count. Suppose that a 128-bit universally unique identifier (UUID), as generated by the Windows operating system, is used to identify each file and client, and a 32-bit update count is used for each client that has modified the file. This results in a per-file cost of 16 bytes with an additional 20 bytes per client that has updated the file. This is depicted at the top of Figure 2.

This scheme can be optimized by noticing that, in the transmission of many version vectors, the UUIDs of the updating clients may be frequently repeated, causing these UUIDs to represent a large fraction of the total transmission. These UUIDs can be replaced with indices into a table of UUIDs transmitted as a header. By replacing client UUIDs with single byte indexes, we can reduce the cost per client that has updated a file from 20 to 5 bytes. This optimization is shown on the second row of Figure 2.

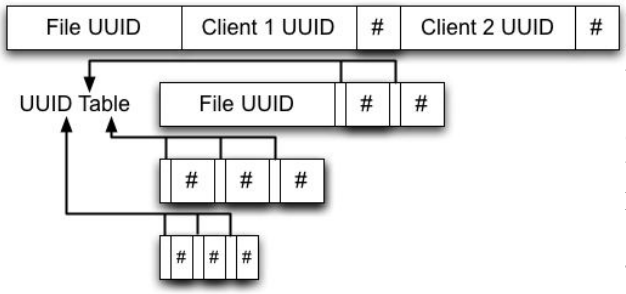

Figure 2. Progressive Optimization of Version Vectors (Drawn to Scale)

The UUIDs identifying each file represent a large fraction of the remaining transmission. This can be reduced by replacing these UUIDs with a new form of identifier, a combination of the UUID of the client that created the file and a 4-byte sequence number assigned to that file by that client upon creation. At first, this would seem to increase the size of each version vector, but reusing the trick of replacing the client UUID with an index into the table of UUIDs provides a net reduction in the size of file identifiers. This reduces the per-file cost from 16 bytes to 5 bytes. This is shown on the third row of Figure 2.

Much of the remaining size is in the representation of counters, both for the file creation sequence number and update counts for each client that has updated the file. However, because most files are updated rarely, a 4-byte integer representation for the count of updates is excessive in most cases. There are many possible smaller representations for integers. For simplicity, a scheme where numbers can be represented by a variable number of bytes will be used here. In each byte, 7 bits are part of a representation of the number and the final bit determines whether the number representation is continued in the next byte. Thus, a single-byte counter can represent values up to 2^7-1, a 2-byte counter can represent values up to 2^{14}-1, and so on. In addition to permitting compact representations of small numbers, numbers of unbounded size can be represented in this scheme, avoiding the possibility of overflow that arises when fixed-size 4-byte counters are used.

If each integer is represented with an average of 2 bytes, the per-file cost of a version vector drops to 3 bytes with a per-updater cost of 3 bytes. This optimization is shown on the final row of Figure 2.

Consider a scenario requiring the transmission of 100,000 version vectors, each with an average of two updaters, a total of 100 distinct updaters, and update counts uniformly distributed between 1 and 1000. Taken together, our optimizations reduce the cost of sending the version vectors from approximately 5,469 KB to 880 KB, a savings of 84%.

We compared this application-specific compression method to a general-purpose compression method, gzip. Running gzip on a collection of 100,000 synthesized version vectors with parameters as in the scenario above, yielded savings of only 35% compared with 84% for our optimizations. Applying gzip to a collection of version vectors optimized using our techniques provided no additional savings.

3.2. Smaller device sends

In the conventional version vector protocol, one device sends the version vectors for every file it contains to the other. A simple optimization is for the device with fewer files to send its version vectors to the device with more files. This is a substantial consideration when the storage sizes of devices are greatly mismatched, such as a desktop with hundreds of thousands of files and a cell phone with a few hundred. This can be accomplished by exchanging short messages containing the number of version vectors that would have to be sent.

Although this optimization requires an additional round-trip, the synchronization protocol will typically run in the background, protecting the user from small increases in latency. Thus, saving bytes of transmission in the protocol at the expense of an additional round-trip is a good trade-off which we will apply again in subsequent optimizations.

3.3. Type filters

It is easy to imagine situations involving devices that store disjoint sets of data such as an iPod that stores only music and a digital camera that stores only photos. Transmitting version vectors to synchronize these two devices would be useless because they have no files in common.

To fix this, devices can exchange a list of file types that they hold. Now, version vectors can be sent for only the files of types both synchronization partners store. In cases such as the iPod and digital camera example above, this optimization can prevent the transmission of all version vectors.

It is noteworthy that the type filters optimization may prevent eventual consistency from being achieved in the event that a file changes type. A device storing a version of the file with the old type, but without interest in the new type, may never receive the new version of the file. However, files changing type does not seem likely.

3.4. File bloom filters

Although type filters can help in many situations, their effectiveness is limited if the two synchronizing devices are interested in the same types of files. Essentially, type filters are a way to determine, with coarse granularity, which files exist on both devices and, hence, might need to be transferred. An

alternative method of determining this information, such as using Bloom filters, could also reduce the number of version vectors transmitted.

A Bloom filter is an efficient data structure for representing a set of objects in a way that permits probabilistic membership tests [5]. Bloom filters have been used in a variety of network applications [6], but, to our knowledge, we are the first to explore their use in the context of a file synchronization protocol.

Bloom filters can be used to reduce the list of files, and ultimately the number of version vectors, sent as follows. A device constructs a Bloom filter containing the file identifier of each file it stores that matches its synchronization partner's type filter. This "file Bloom filter" is transmitted to the synchronization partner. The synchronization partner tests the file identifier of each of its files that matched the type filter for membership in the received Bloom filter. For each file that is a member, that file is determined to exist on both devices (with high probability) and so the version vector for that file is transmitted.

Bloom filters can produce false positives, but the only harm is that useless version vectors will be transmitted to the synchronization partner, which will be able to discard them. Bloom filters have no false negatives. Hence, the result is that version vectors are transmitted for all common files along with perhaps a few extra ones.

3.5. Version bloom filters

Bloom filters can be exploited even further. Observe that most files are updated rarely. Therefore, even if a protocol transmits the version vectors only for files contained on both synchronization partners, the version vectors on both devices will often be the same. A second use of Bloom filters can avoid transmitting version vectors in that case.

After the file Bloom filter transmission allows one device to determine which files are in common, that device constructs a "version Bloom filter" containing the version vector and file identifier of each file stored by both devices. This filter is sent to the synchronization partner who responds with version vectors for all files stored by both devices and whose version vector is not a member of the version Bloom filter. The failure of this membership test indicates that the devices store a different version of the file in question.

There is a small complication that arises though false positives produced by the version Bloom filter. A false positive on the membership check of a version vector results in the version vector not being transmitted, causing incomplete synchronization.

This can be resolved by rotating the hash functions used in the construction of the version Bloom filter. Each rotation greatly decreases the chance that an update will be missed. Because synchronizations are performed periodically, the update will eventually arrive.

4. Putting it all together

To make use of these proposed optimizations, these ideas need to be composed into a synchronization protocol. We have devised a 6-step protocol, shown in Figure 3.

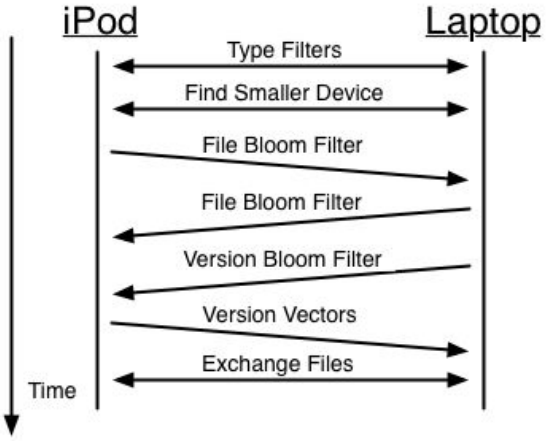

Figure 3. Efficient Synchronization Protocol

An example usage situation is synchronization between an iPod and a laptop. For illustrative purposes, the iPod contains 10,000 MP3 files and 10,000 JPG files and the laptop contains 50,000 MP3 files (including 5,000 of the ones on the iPod), as well as 100,000 other files of other types, but no JPGs. Each file has historically been modified by 2 devices and thus each version vector contains two entries. 10 of the MP3s on the iPod have recently been modified and those updates have to be transmitted to the laptop. For simplicity, all Bloom filters are sized such that their expected false positive rate is approximately 1%, consuming 1.2 bytes per element contained, assuming ideal hash functions.

In the first step, the iPod initiates the synchronization protocol by sending its type filter for MP3s and JPGs to the laptop. In return, the laptop responds with its type filter for MP3s but not JPGs.

Next, both devices count the number of files they store that match their synchronization partner's type filter. These counts are exchanged to determine which partner has fewer files to transmit information about. In this case, the iPod has 10,000 MP3s that match the laptop's type filter and the laptop has 50,000 MP3s that match the iPod's type filter. Using the "smaller device sends" optimization, this indicates that the iPod should be the one to act next.

The iPod places the file identifiers of all its files matching the laptop's interest (the 10,000 MP3s) into

a Bloom filter and sends the resulting file Bloom filter to the laptop. This Bloom filter is approximately 12 KB.

Now, the laptop uses the file Bloom filter to determine which files are contained on both devices. It creates another file Bloom filter containing the identifiers of all such files, the 5,000 MP3s on both machines, which it sends to the iPod to share this information. This Bloom filter is approximately 6 KB. It also takes those files and places their version vectors into a Bloom filter, creating a 6 KB version Bloom filter that is also sent to the iPod.

At this point, the iPod also knows which files are on both devices and can use the version Bloom filter to determine which files have different versions across the two devices. For each such file, it transmits the version vector for that file to the laptop. The expected number of version vectors to send (after false positives) is 59.4. With each version vector 9 bytes, that is approximately 535 bytes.

Using the version vectors received from the iPod, the laptop can determine which machine has the newer version (or detect a conflict) and send files to or issue requests for files from the iPod, completing the protocol. In total, this method determines which files must be transmitted to achieve synchronization with transmission of only around 25 KB, an improvement of over 40x.

Version Vector Protocol	1,094 KB or 8,203 KB
Smaller Device Sends	1,094 KB
Compressed Version Vector	176 KB
Type Filter	88 KB
File Bloom Filter	68 KB
Version Bloom Filter	25 KB

Figure 4. Cumulative Effects of Optimizations

To demonstrate the size of the improvement gained by these optimizations, Figure 4 shows the transmission costs that would have been incurred by the conventional version vector protocol and each successive optimization. The cost of the straightforward version vector protocol, presented in the first row of this table, depends on which device transmits version vectors. Subsequent rows of this table demonstrate the benefit of proposed optimizations. Each row includes the optimizations of all previous rows.

5. Related work

The work presented in this paper is the first to explore the optimization of overhead in a version vector synchronization protocol through the use of Bloom filters and protocol-specific compression techniques. Others have taken different approaches in addressing a similar goal of reducing the metadata that must be stored and communicated in a replicated system.

Coda relied on update logs and log compaction to support disconnected operation on file system clients and to efficiently reconcile such clients with the server upon reconnection [10]. However, Coda only supported a client-server model and, hence, its log-based techniques would be unsuitable for encounter-based synchronization between nearby devices.

Work on Fluid Replication focused on fast, cheap synchronization between dynamic replicas, called "WayStations," and the file server [7]. Like Coda, it assumed a client-server architecture, relied on update logs, and supported partial replication through a caching model. The key idea in this work is to record the least common ancestor for client updates and use this information to quickly determine the sets of files that may have been updated since the last synchronization. Unfortunately, this depends on having a single file system tree and performing synchronization with a single server.

Ficus is a replicated file system that used peer-to-peer synchronization to ensure full propagation of updated files and used version vectors to detect file conflicts [9]. Selective replication was added to later versions of Ficus [15]. It synchronized groups of common files at one time and used status vectors to determine the files stored at other replicas, assuming that each replica knows about all other replicas. The optimization techniques described in this paper apply directly to Ficus and similar systems without requiring the maintenance and storage overhead of per-file status vectors.

Bayou also supported a peer-to-peer synchronization model that is well-suited for chance encounters between mobile devices [18]. Its use of per-replica knowledge vectors rather than per-file version vectors allowed a replica to compactly represent and communicate the complete set of updates that it had seen. Bayou achieved synchronization overhead that is even lower than reported in this paper. However, Bayou relied on an update log, only supported full replication, and utilized application-specific conflict detection.

PRACTI extended the Bayou design to allow partial replicas that can contain arbitrary subsets of files [4]. Conceptually, partial replicas in PRACTI store metadata for all files but are able to reduce metadata overhead through "imprecise invalidations".

WinFS extended the Bayou design to provide automatic conflict detection based on version vectors while avoiding storing a version vector per file [12]. WinFS also supported partial replication through the use of "community folders" but did not allow devices to store arbitrary collections of files.

Finally, concerns about the unbounded size of version vectors have caused some researchers to

propose compact representations [1][3][10] and techniques for pruning entries that are no longer needed, such as entries that are globally known by all replicas [16]. These techniques could be applied in conjunction with the protocol optimizations presented in this paper.

6. Conclusion

Through changes to the representation of per-file version vectors and a protocol that uses checks of increasingly fine granularity to reduce the number of version vectors transmitted, we have achieved huge reductions in the cost of determining the updates that must be sent in a distributed storage system exploiting opportunistic connectivity. These optimizations could be applicable in a variety of synchronization protocols that use version vectors or similar concepts.

Importantly, these cost reductions allow us to take advantage of file synchronization opportunities that would previously have been prohibitively expensive and improve existing systems by increasing the frequency of synchronization. These gains are possible even in a system with open-ended membership and composed of devices with limited computational and storage resources. Thus, this work is an important enabler for encounter-based synchronization between mobile devices.

10. References

[1] Almeida, J. B., P. S. Almeida, and C. Bacquero. Bounded version vectors. *Proceedings International Symposium on Distributed Computing (DISC)*, 2004, pp. 102-116.

[2] Apple Corporation. Bonjour. http://developer.apple.com/networking/bonjour/.

[3] Arora, A., S.S. Kulkarni, and M. Demirbas. Resettable vector clocks. *Proceedings 19th Symposium on Principles of Distributed Computing (PODC),* Portland, Oregon. 2000.

[4] Belaramani, N., M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, J. Zheng. PRACTI replication. *Proceedings USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[5] Bloom, B. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM* 13(7), pp. 422-426, 1970.

[6] Broder, A. and Mitzenmacher, M. Network applications of Bloom filters: A survey. *Internet Mathematics* 1(4), pp. 485-509, 2004.

[7] Cox, L. P. and Noble, B. D. Fast reconciliation in fluid replication. *Proceedings International Conference on Distributed Computing Systems*, 2001.

[8] Ford, B., Strauss, J., Lesniewski-Laas, C., Rhea, S., Kaashoek, F., and Morris, R. Persistent personal names for globally connected mobile devices. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006, pp. 233-248.

[9] Guy, R. G., Heidemann, J. S., Mak, W., Page, T. W., Popek, G. J., and Rothmeier, D. Implementation of the Ficus replicated file system. *Proceedings of the Summer USENIX Conference*, June 1990, pp. 63-71.

[10] Huang, Y.-W. and P. Yu. Lightweight version vectors for pervasive computing devices. *Proceedings IEEE International Workshops on Parallel Processing*, 2000, pp. 43-48.

[11] Kistler, J. J. and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems* 10(1), pp. 3-25, February 1992.

[12] Malkhi, D., and Terry, D. Concise version vectors in WinFS. *Proceedings Symposium on Distributed Computing*, Cracow, Poland, September 2005, pp. 339-353.

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

[14] Peek, D., and Flinn, J. EnsemBlue: Integrating consumer electronics and distributed storage. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006, pp. 219-232.

[15] Ratner, D., Popek, G. J., Reiher, P., and Guy, R. Peer replication with selective control. *Proceedings MDA '99, First International Conference on Mobile Data Access*, Hong Kong, December 1999.

[16] Saito, Y. Unilateral version vector pruning using loosely synchronized clocks. Technical Report HPL-2002.

[17] Scott, D., Sharp, R., Madhavapeddy, A., and Upton, E. Using visual tags to bypass Bluetooth device discovery. *ACM SIGMOBILE Mobile Computing and Communications Review* 9(1), pp. 41-53, January 2005.

[18] Terry, D. B., M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings 15th Symposium on Operating Systems Principles (SOSP)*, Cooper Mountain, Colorado, December 1995, pp. 172-183.