Algorithms to efficiently manage files in a cloud

Mark C. Jeffrey and Weihan Wang

March 5, 2013

Presented is a set of algorithms for efficient sharing of files among devices in a peer-to-peer configuration. These algorithms are intended to build upon the system and methods claimed in Patent US 2012/0005159 A1 [1], which shall herein be referred to as the base system.

The algorithms assume such a system that identifies files and folders (concisely referred to as *objects*) via globally unique object identifiers or ids (e.g. 128-bit UUID). An *update* to an object signifies the creation, deletion, modification, or renaming of a file or folder. In the base system, a consistency algorithm exists for the propagation of these updates among devices (i.e. eventual consistency is guaranteed). The system is modeled to perform optimistic replication. The base system describes a *store* (aka a "Library") that is a special folder with which a specified group of users can share and collaborate on the folder's contents. Conceptually, through object ids, the base system maintains and synchronizes a logical representation of the physical file system tree. Updates to the logical file system are persisted to the physical file system.

The algorithms presented here extend the base system by enabling:

1. optimized network queries to download a file
2. resolution of duplication/conflict of object ids for the same file path during a network partition
3. propagation of object deletion updates, and ability to selectively synchronize files to a subset of devices for a given user
4. avoiding re-downloading a file when it exists locally but has remotely been moved from one shared folder to another
5. presentation and resolution of concurrent updates to files on multiple devices
6. one device synchronizing the stores of multiple users
7. version history for every file
8. presentation of the sync status of every file and folder

Background

Logical Object Generation

When a device physically creates a new file or folder, the base system dictates that the local device randomly generates a new object id to logically represent that physical object. This local generation

is applied to obviate centralized coordination of assigning logical object ids; centralized assignment would impede optimistic replication. Because update propagation tracks logical objects, there must be a one-to-one mapping between a physical file/directory and its logical object representation. The distributed approach to mapping logical and physical objects can result in conflicts where, in a distributed system of many devices, more than one logical object may refer to the same physical object. This problem is addressed in the "Aliasing" section of this patent.

Update Propagation

The base system achieves update propagation through push and pull of *version vectors* [1, 2]. A version vector is a data structure used in optimistic replication systems; it consists of a mapping of device ids to integer version counts per id (see the base system for further details). For each new update, a device will push the version vector corresponding to that change via a multicast, to a number of online devices. To ensure no updates are missed, each device periodically pulls all other devices for their known updates (i.e. version vectors).

Stores: Shared Folders

The system permits distinct users to collaborate on a shared folder and its contents. Such a shared folder is herein referred to as a store, which defines the set of users who may share its contents, as specified by the store's Access Control List (ACL). An ACL is a mapping from user id to permission on the contents of the store. A device is permitted to sync the files and folders of a store if its owning user id in the Access Control List (ACL) of the store. Each device has a root store with only the owning user in the ACL; this root store thus syncs only with devices owned by the user.

An object o may be moved between stores, say S_1, and S_2. In some algorithms, it is important to distinguish o under S_1 vs S_2, and we thus annotate (S_1, o) to reference object o under store S_1.

Collector: Efficient selection of devices to request file download

In a peer-to-peer configuration of devices that share a store, through the consistency algorithm of the base system [1], updates to objects are propagated among devices. In this context, receipt of an update for object with id o (from the perspective of "this" local device) signifies that the device now has knowledge that some modification, creation, renaming, or deletion of o occurred on some other device in the network. Given this update knowledge, and assuming the local device knows all those that are online, how can the local device choose from which of its peers to download that update? The Collector algorithm defines a set of steps to convert known missing updates into locally downloaded files or folders, without naively querying every device for the update until one succeeds. This latter approach would waste bandwidth and computation by potentially querying devices that do not have the update. For example, suppose a network has three devices sharing files, d_1, d_2, d_3. Device d_3 modifies object o, which propagates updates to d_1 and d_2. Upon receipt of that update for o, d_1 should only request to download the change from d_3, not wasting bandwidth and CPU resources by requesting from d_2.

The claimed algorithm records and shares, among peers, which updates each device has observed in the distributed system. The algorithm will determine, and query from, the set of devices that are known to have the update. Given the set of devices known to have the update, the algorithm

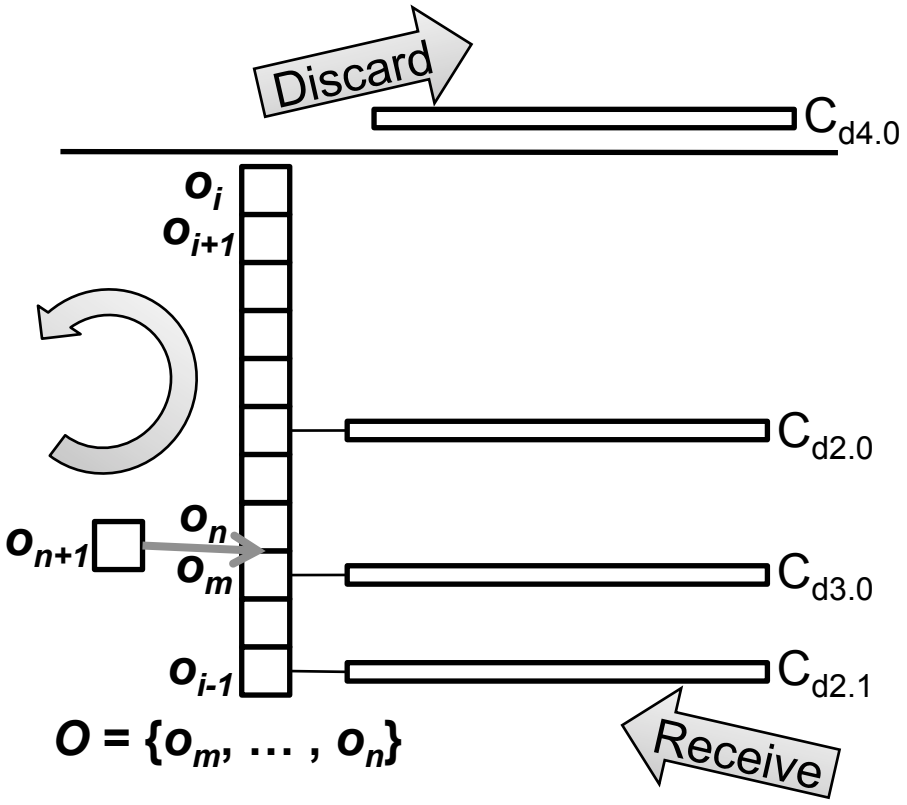

Figure 1: Collector Diagram

selects devices to query in order of preference based on a device's network bandwidth, workload, etc.

Data Structures

Sharing existence of local updates

There exists locally, for every device d known to this local device, two sets of object ids: S_{xd} is the set of objects updated since the last version exchange between the local and d. The other, S_{hd}, is the set of all other objects updated on the local device. The two sets are disjoint and their union represents all objects with updates downloaded on this device. After performing an update locally on an object, the local device adds the object to the S_{xd} set, for every known remote device.

When responding to a pull version request [1] from a remote device d, the local device additionally sends the set above, S_{xd}, of locally recorded updates. The remote device d will respond with success or failure. On receipt of a success message, the local device unions S_{xd} with S_{hd}, which records all objects ever updated at this device, then empties the S_{xd} set. Thus S_{xd} acts as a "diff" of objects updated between version exchanges of the local device and d.

Collector sets

In Figure 1, a set of objects, $O = \{o_m, \ldots, o_n\}$, is pictured as a vertical queue with o_i currently at the head, and o_{i-1} at the tail, where $1 \leq m \leq i \leq n$. These objects are known to have updates on remote devices, and are *collected* by cycling through the queue. Collecting an object involves requesting and downloading the update from a remote device (not covered in this patent). The goal of this algorithm is to collect updates for all objects in O by contacting only those devices which actually have any given update available. The queue wraps the O set, giving the objects a particular order, where their *sequence numbers* are monotonically increasing—object o_i has sequence number i and therefore precedes o_{i+1}. Object o_n is adjacent to o_m, where m is the minimum sequence number among the objects in O; m could take value 1, but objects are also removed from this queue. The current object to be collected is o_i at the head of the queue.

Also pictured in Figure 1 are the sets $C_{di.t}$ as long rectangles. Whereas sets S_x and S_h (described above) record updates the local device has observed, the Collector algorithm relies upon the set C_d, which represents the objects for which device d has updates (i.e. the S_x from d). Generally, there exists locally, for every device d known to the local device, one or more sets $C_{d.t}$, which were recorded locally as a result of exchanging versions for updates with device d. Because multiple version exchanges can occur with the local device and d, more than one of these sets may have been sent.

The set of sets is hereafter called the Collector Sets, or $CS = \{C_{d1.0}, C_{d1.1}, \ldots, C_{dp.0}, C_{dp.1}, \ldots C_{dp.tp}\}$. As seen in the figure, each set $C_{d.t}$ is assigned to some index in the object queue, O. A newly-received C is assigned to the tail at the bottom of the queue (e.g. $C_{d2.1}$) in the Figure). There is no relation between C, and the object which shares the same index; this association is intended to safely discard the sets when their use is expired. It is possible that more than one C set could be assigned to the same index, representing sets of objects from distinct devices. The reader may note that when receiving several sets from device d, the incoming set could be unioned with the locally-present C_d. This approach is not taken so that CS can be more easily pruned: if $C_{d.t}$ sets remain separated, then an older set could be discarded while keeping the recently-received collector sets. The details of receiving a set C_d from remote device d is discussed below.

Method to collect updates efficiently

Given the aforementioned data structures, the Collector algorithm is described to collect the updates for each object in O, by contacting only those devices known to have a given update. The basic idea is that the set of devices to contact for object $o \in O$ is determined by querying for the existence of o in each C set. To determine the set of devices, D, known to have updates for o:

```
function GetDevicesWithUpdates(o: object id, CS: Collector Sets)
    D ← ∅
    for C_d ∈ CS do
        if o ∈ C_d then
            D ← D ∪ {d}
        end if
    end for
    return D
end function
```

To efficiently collect all objects in O, the local device "rotates" through the objects of the queue and downloads updates from the set of devices, D, returned by the algorithm above for each object. The local device discards the object sets $\{C\}$ that are linked to the object popped from the head of O, as all elements in O that could appear in C have already been queried. If collection of some object o succeeds, the object o is removed from the queue.

```
function CollectAllUpdates(CS: Collector Sets, O: object queue)
    CS_bkup ← CS
    while O is not empty and CS is not empty do
        o ← pop the head of queue O
        D ← GetDevicesWithUpdates(o)
        download updates for o from any device in D until successful
        CS_o ← subset of CS linked to o
        for C ∈ CS_o do
            remove C from CS
        end for
        if failed in downloading all updates for o then
            push o to the tail of queue O
            for d ∈ D do
                for C_d ∈ CS_bkup where C_d was received from d do
                    CS ← CS ∪ {C_d}
                end for
            end for
        end if
    end while
end function
```

Observe that on failure of downloading all updates for o from all devices in D, the sets received from devices in D are restored from CS_{bkup}. See below for details on how a set is inserted into the Collector Sets.

Populating the Object Set and Collector Sets

Insertion to the object queue

Concurrent with the collection loop execution in CollectAllUpdates, the consistency algorithm could receive knowledge of new updates for some object o. If the object is not already in O, it must be inserted, but not at the tail as in a conventional queue. To maintain the monotonic property of sequence numbers in O, the new object o is assigned a sequence number of $n+1$, and is consequently inserted into the queue following o_n (as seen in Figure 1).

Insertion to the collector sets

Following a push or pull version exchange from device d, a Collector Set C_d is additionally received or created on the local device. The following pseudocode adds a non-empty C_d to the CS set. An empty C_d can be ignored.

```
function AddToCollectorSets(C_d: collector set from d, CS: collector sets, O: object queue)
    CS ← CS ∪ {C_d}
```

$o_t \leftarrow$ tail of O
if $\exists C_c \in CS$ where C_c shares sequence number t and $c == d$ **then**
$C_c \leftarrow C_c \cup C_d$
else
link C_d to o_t
end if
end function

Note that multiple C_d sets could be linked to the same object in the O queue, if the queue did not rotate between calls to AddToCollectorSets.

Upon receipt of a push version update from device d, after recording versions locally, the local device creates an object set C_d of all objects with new updates in the given versions. If all updates for some object o were already known locally, object o is not added to C_d. If C_d is non-empty, AddToCollectorSets is executed on the set.

Upon receipt of a pull version response from device d, the local device first records the versions locally, including stable and unstable versions (see the patent [1] for description of stable and unstable). As indicated earlier, device d sent its set S_x of updated objects. These objects have updates on d that are new since the last pull version exchange made with d from this device. If the received S_x is non-empty, the local device records it as C_d, and executes AddToCollectorSets on it. The local device subsequently responds to device d, indicating successful receipt of the set C_d.

Implementation Details

Bloom filter optimization

In one embodiment of the invention, the sets S_h and S_x (and consequently the C sets of CS) are implemented as Bloom filters [3] of length 1024 bits and four hash functions. The hash functions are disjoint bit selections from the 128-bit UUID object id. The CS Collector Sets is thus a set of Bloom filters. This implementation of the object id sets permits

- constant-space transmission of object id sets
- constant-time membership query of the collecting object in O, in each Bloom filter of CS.

The negative consequence of replacing accurate sets with Bloom filters is that the inherent membership query false positives would suggest that a device could request to download an update from a device that does not have the update locally. This results in some bandwidth waste, but the algorithm remains correct, as the requested device can respond with a NACK.

Aliasing: Name-Conflict Resolution Of Object IDs

As stated earlier, the base file syncing system generates an object identifier for every file and folder created on a device. The system thus maintains logical objects that internally represent the physical objects on the local file system. At a local device, there must be a one-to-one mapping between the logical and physical objects, however remotely received updates can point two logical objects to the same physical path; these *name conflicts* result in two specific ways:

- concurrent creation of a physical object (same pathname) on at least two peers, resulting in more than one distinct logical object (e.g. Figure 2); and

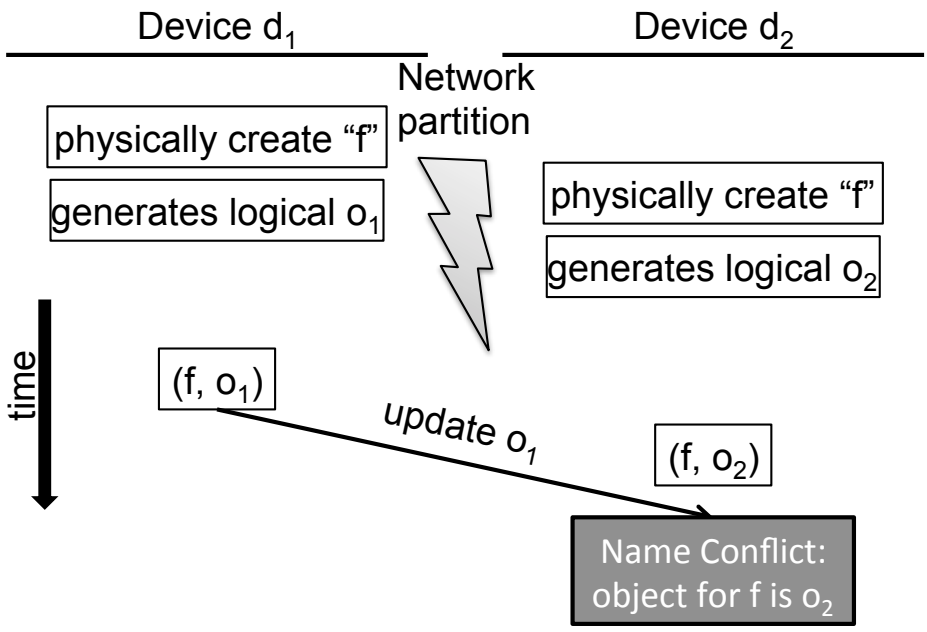

Figure 2: Concurrent creation of a file generates two logical object ids for the same path, yielding a name conflict between o_1 and o_2.

- when one peer locally renames an existing physical object to the same path as a physical object created on another peer.

Henceforth in this section, the term *object* will refer to logical object for brevity, and physical object will be stated outright.

When peers exchange information about distinct logical objects that represent the same-named physical object, as in Figure 2, this name conflict must be resolved. In the figure, devices d_1 and d_2 are initially separated by a network partition. In that partition, they both create a physical file (or directory) with equivalent path "f", which yields two distinct object ids, o_1 and o_2. Subsequently, the network partition disappears, and device d_1 sends its update that object o_1 has been created with path "f". Device d_2 discovers that it already has a logical object o_2 for path "f", violating the invariant that logical and physical objects must have a one-to-one mapping.

One approach to resolve a name conflict is to deterministically rename one of the two objects involved in the name conflict. However, it is conceivable that some users of the system could already have files replicated on multiple devices, leading to name conflicts on several files, and this rename approach will then create n copies of the same file, where n is number of peers with the replicated file. This leads to poor user experience. Striving for a better user experience, a method is presented below to avoid renaming and duplicating files/directories, instead opting to *alias* one of the name-conflicting objects as the other.

Merging alias and target objects

During a name conflict, specifically it is the differing object ids for the same physical object that conflict. The claimed resolution strategy labels one of the object ids as the *target*, and the other

as the *alias*. The device observing the name conflict merges all meta-data describing the alias object into the target, (i.e. consistency algorithm versions), then subsequently replaces all meta-data about the alias object with a pointer relationship, and shares that pointer relationship with other devices. The assignment of alias and target objects must be deterministic, so that all devices which encounter the same name conflict will label the same object as the target, and other objects as aliased. To this end, since object ids can form a total order by value, the value of object ids is used to determine the target and alias assignment. Specifically, given a set of n object ids involved in a name conflict, in one embodiment of resolving the name conflict, the object id which will become the target, o_t, has the maximum value in the set: $o_t = \max(\{o_1, \ldots, o_n\})$.

Alias Version Updates

Nearly all meta-data for the aliased object is merged into the target, including version vectors, except for the versions associated with the aliasing operation itself. When o_a is aliased to o_t at the local device, this change in state must be propagated to all other devices to achieve a consistent global state. Therefore a new version must be created for o_a, but all other versions describing the previous file updates on o_a must be merged into o_t, so that o_t covers the version history of its alias objects. In this patent, the system updates version vectors by drawing from two spaces, one for alias updates, and the other for all other object updates. When merging versions from the alias object into the target, only versions from the non-alias space are merged; the alias object keeps its alias version history.

In one embodiment, version vectors for the alias update space have odd-valued integer counts, whereas version vectors for the non-alias update space have even-valued integer counts. As with classic version vectors, the integer counts must be monotonically increasing for events with a happens-before relationship.

Object State Model

The method is first presented as a state transition model, before pseudo code implementation. The model summarizes the initial state and expected result of the algorithm. Defined first is a simplified model of the logical states which can represent a physical object at a **single** device. The model considers only the metadata of a physical object, not content. We begin with simple annotations, then definitions of state and transition. We explain how they are represented in the state transition diagrams to follow.

Let (n, o) represent physical object with path n, and logical object (id) o; the object o is called a non-aliased object. If logical object o_a is known to alias to o, then we write $o_a \rightarrow o$, knowing that o_a is an alias object for target o. Notice there is no path associated with o_a; only the alias relationship (i.e. the pointer) is stored for an aliased object.

State

In the case above, where object o_a aliases to (n, o), we write that the state of physical object n is $\{(n, o), o_a \rightarrow o\}$, which means that file/directory n is logically represented by object o, and any system references to o_a will alias (or are derefenced) to object o. Resulting from transitions, the name-conflict resolution strategy permits only "acceptable" states which abide by the following two invariants:

1. all states must include one and only one non-aliased object; and

2. the target of all aliases must be the non-aliased object—alias "chains" are not permitted

These invariants simplify the verification of correctness. The first invariant means that a device cannot download information about an aliased object o_a if its target o_t does not exist locally. This is somewhat analogous to avoiding dangling pointers in C. The second invariant avoids creating chains of aliases, e.g., $\{o_1 \rightarrow o_2, o_2 \rightarrow o_3\}$. Since o_2 is not a non-aliased or target object locally, o_1 should not refer to it. In the algorithm of the later section, when referring to a target object, one can be sure it exists locally as a non-aliased object. In subsequent state transition diagrams, acceptable states are represented as large circles or nodes, including the non-aliased object, (n, o), and any objects aliased to o. For example, at the center of Figure 3 is a node representing the state $\{(n, o_2), o_1 \rightarrow o_2\}$. The other nodes in the figure represent states $\{(n, o_1)\}$ and $\{(n, o_2)\}$.

Transitions and Messages

Since name conflicts are discovered when downloading an update about an object, the transitions among states are spurred by messages between devices. In the claimed name-conflict resolution method, the local device can receive two types of messages about any object in the system from any other device: (i) a non-alias message, or (ii) an alias message. A non-alias message is labeled and contains meta-data of the form (n, o), implying that the sender device has provided the local device with an update about file/directory n with logical object o that is not aliased on the sender. An alias message is of the form $o_a \rightarrow o_t$, implying that there is an update about o_a, and the remote device thinks its target is o_t.

In the subsequent state transition diagrams, an arrow (directed edge) shows the expected transition from a state when a particular message about an object has been received. The state transitions are agnostic about the source device. Recall that the arrows represent transitions, not messages, but they are labeled by the message that induces the transition. Sample transitions can be seen in Figure 3, such as when a device in state $\{(n, o_1)\}$ receives an alias message $o_1 \rightarrow o_2$, then subsequently transitions to state $\{(n, o_2), o_1 \rightarrow o_2\}$.

Object State Transition Diagrams

The following three diagrams show all possible states that a physical object n can occupy given a replication factor, and all expected transitions across those states.

Figure 3 considers a system where physical object n has two replicated logical objects, which are ordered such that $o_1 < o_2$. Thus o_2 is the eventual target, and o_1 should become the alias. The center node represents state $\{(n, o_2), o_1 \rightarrow o_2\}$, which implies that the name conflict on (n, o_1) and (n, o_2) has been fully resolved. The resulting transition is shown for all three possible types of messages ((n, o_1), (n, o_2), $o_1 \rightarrow o_2$) from each state. Notice that if all three messages are eventually received, the final resulting state is the fully-resolved state.

Figure 4 considers a scenario where physical object n has three replicated logical objects, which are ordered such that $o_1 < o_2 < o_3$. In this scenario, o_3 is the eventual target, with o_1 and o_2 becoming the aliases. The center node represents fully-resolved state $\{(n, o_3), o_1 \rightarrow o_3, o_2 \rightarrow o_3\}$. From this center state, receipt of messages (n, o_3), $o_1 \rightarrow o_3$, $o_1 \rightarrow o_2$, or $o_2 \rightarrow o_3$ will transition back to the same state (omitted from the diagram for brevity). The three states from Figure 3 can be seen in the left part of Figure 4. The state transitions among them that were previously shown

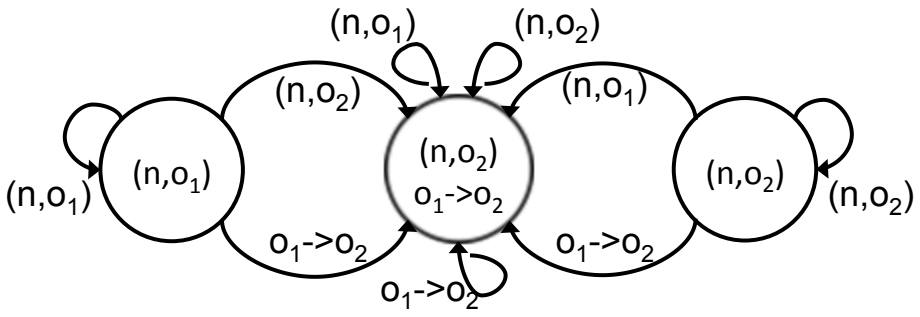

Figure 3: State transition diagram for resolving name conflicts with two distinct object ids

are hidden for brevity, but would otherwise exist in this figure for completeness. The bottom three states, $\{(n, o_2)\}$, $\{(n, o_3)\}$, and $\{(n, o_3), o_2 \to o_3\}$, model another 2-object conflict resolution, and thus share identical transitions to those in Figure 3, by simply replacing references to o_2 with o_3, and o_1 with o_2. Likewise with the three states in the right side of Figure 4. Thus in the latter figure, we opt to omit all transitions that are described in a 2-object conflict scenario to avoid redundancy, but they remain intrinsic to the state model.

Figure 5 extends the latter two scenarios to four replicated objects with ordering $o_1 < o_2 < o_3 < o_4$. In this diagram, there are four outer states with one non-alias object and two aliases, three outer states with one non-alias and one alias, and at the center is the fully-resolved state with one non-alias (n, o_4) and three aliases to o_4. As seen in the previous name-conflict resolution state diagrams, some redundant transitions and states are omitted in Figure 5 for clarity. The top state $\{(n, o_3), o_1 \to o_3, o_2 \to o_3\}$ is equivalent to that in Figure 4. All states and transitions that describe a 2- or 3-object resolution should additionally appear in this figure, but are omitted as they are redundant when considering the previous figures. Note that the three other 2-alias states of Figure 5 can be surrounded with identical state hexagons with one of the objects replaced. The figure includes states with one alias object (e.g. $\{(n, o_2), o_1 \to o_2\}$) because after receipt of a particular alias message ($o_3 \to o_4$), the local device should transition to the center, fully-resolved state. In contrast, from a state with no alias objects (e.g. $\{(n, o_2)\}$ which is not pictured), there is no single alias nor non-alias message that will transition to the fully-resolved 4-object state. From the center state, receipt of the following messages will restore to the same state: $(n, o_4), o_1 \to o_2, o_1 \to o_3, o_1 \to o_4, o_2 \to o_3, o_2 \to o_4, o_3 \to o_4$.

The presented state transition diagrams can be further extrapolated for five, six, or more replicated objects. A scenario with five replicated objects would have a center state with four aliases and one non-alias. The center of Figure 5 would be among the five surrounding states with three alias objects. One must also consider the states which can transition to the fully-resolved state through one alias message.

Aliasing Algorithm Pseudo-Code

In one embodiment of the invention, the following algorithm facilitates the transitions between acceptable states, providing two main routines for a device to locally handle receipt of the two message types. Recall that the two types of incoming messages are:

1. $(o, v(o), n(o), \ldots)$ an object, its version, its path, and other meta-data

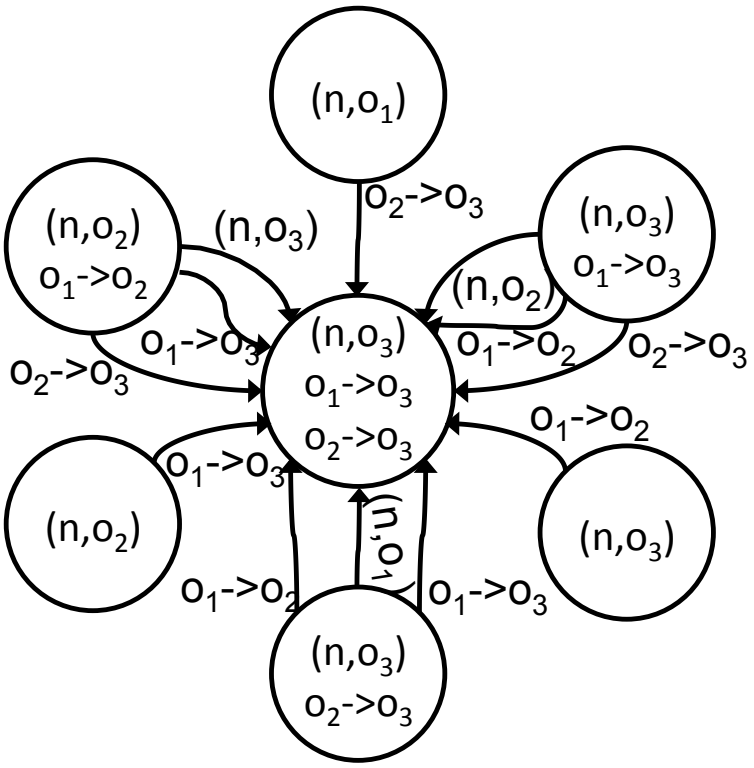

Figure 4: State transition diagram for resolving name conflicts with three distinct object ids

2. $(o_t, v(o_t), n(o_t), o_a, v(o_a), \ldots)$ an alias message with same content as above, additionally stating that o_a aliases to o_t.

Some noteworthy annotations for the following pseudocode:

- $vr(o)$ = version of object o, received remotely
- $vl(o)$ = version of object o, retrieved locally
- $nr(o)$ = name of object o, received remotely
- $nl(o)$ = name of object o, retrieved locally

Logical objects must represent the same type of physical object in order to be aliased (i.e. both the remote and local objects must be a file or both must be a directory).

Common Routines

Both message handlers share a common routine to logically receive a non-alias object that was previously locally unknown. If receipt of the new object will create an alias object, the argument $o_{nonewvers}$ is used to decide whether to publish a new version/update about the alias action. Some callers of this handler do not need to publish the creation of the alias object, such as when an alias message is being processed; the update for the aliasing operation was created at some other device, hence why the local device is now processing an alias message.

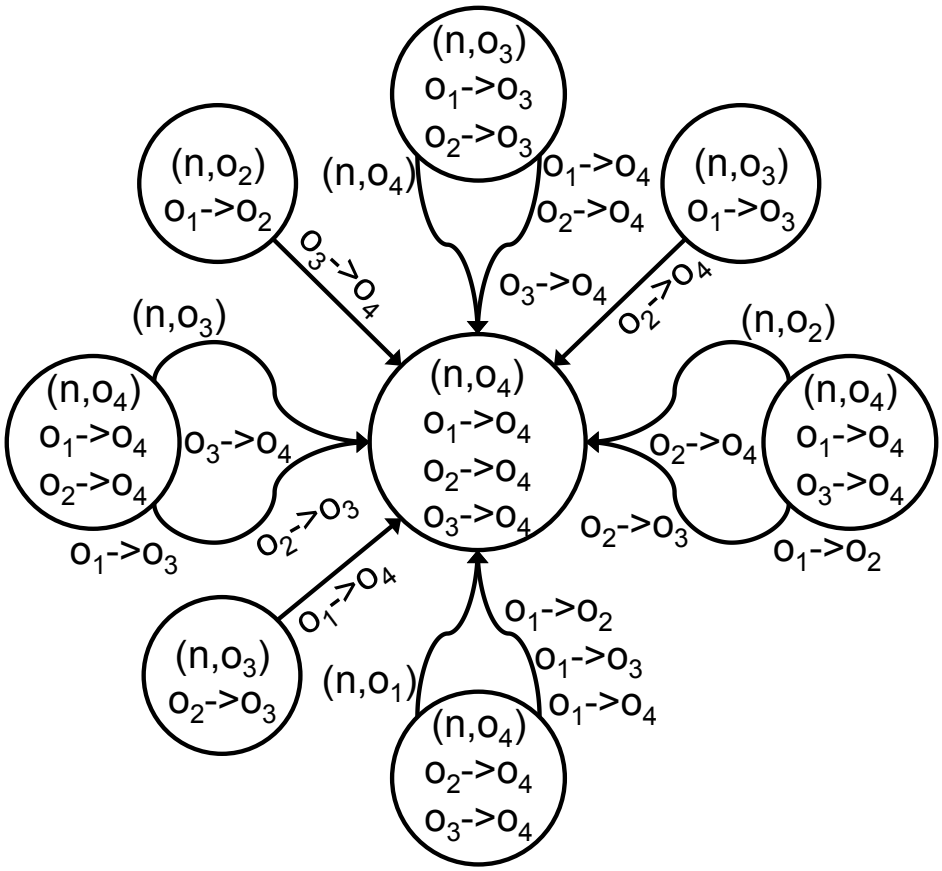

Figure 5: State transition diagram for resolving name conflicts with four distinct object ids

function ReceiveNewObject(o: object id, $vr(o)$, $nr(o)$, $o_{nonewvers}$)
  if locally $\exists o_{nr}$ such that $nr(o) == nl(o_{nr})$ and $o \neq o_{nr}$ **then** ▷ a name conflict exists
    if have not queried sender about o_{nr} **then**
      query sender for meta-data of o_{nr} and retry o later
      return
    end if
    if $o_{nr} \neq \max(o, o_{nr})$ **then**
      swap(o,o_{nr}) ▷ ensure o_{nr} is the target
    end if
    MergeAliasIntoTarget(o, o_{nr})
    if $o \neq o_{nonewvers}$ **then**
      $vl(o)$ = new version from alias version space
    else
      $vl(o) = \emptyset$ ▷ the caller has indicated that the version of o need not be incremented
    end if
  else
    perform typical non-name-conflict execution
  end if

end function

The latter routine defers to the following subroutine to merge the alias and target objects.

function MergeAliasIntoTarget(o_a, o_t)
  merge non-alias versions of o_a into o_t
  if o_a is a file **then**
    move conflicting content branches from o_a to o_t
    merge content branches with identical content in o_t
  else if o_a is a directory **then**
    move children of o_a into o_t, renaming if a name-conflict would result
    delete physical directory of o_a
  end if
  locally record $o_a \rightarrow o_t$
  for $o \in \{o | o \rightarrow o_a$ is recorded locally$\}$ **do**
    replace $o \rightarrow o_a$ with $o \rightarrow o_t$ ▷ must resolve chaining of existing alias objects
  end for
end function

Non-Alias Messages

The non-alias message handler is relatively straightforward, in some branches dispatching to the standard file syncing algorithm, not the name-conflict resolution strategy.

function HandleNonAliasMsg(o, $vr(o)$, $nr(o)$)
  if locally o is non-aliased **then**
    perform typical non-name-conflict handling
  else if locally $\exists o_t$ such that $o \rightarrow o_t$ **then**
    no-op ▷ remote updates on o will later be sent via o_t
  else
    ReceiveNewObject(o, $vr(o)$, $nr(o)$, null)
  end if
end function

Alias Messages

The alias message handler is more involved. As stated at the beginning of this section, the data of a non-alias message is actually embedded in an alias message, where this data represents the target. This protocol is required to satisfy the first invariant for our object states: a consistent state for physical object n must logically have one non-aliased object. When an alias message is received such that the target was previously unknown to the local device, the target object must be accepted into the local system before processing the aliased object. This represents the first phase of the routine. With the target object handled correctly, the alias object is then processed.

function HandleAliasMsg(o_t, $vr(o_t)$, $nr(o_t)$, o_a, $vr(o_a)$)
  if locally o_t is not known **then** ▷ as aliased or non-aliased object
    ReceiveNewObject(o_t, $vr(o_t)$, $nr(o_t)$, o_a) ▷ do not create new version for o_a
  end if
  if locally $\exists o_e$ such that $o_t \rightarrow o_e$ **then**

```
        o_t = o_e                                   ▷ avoid alias chains; reset o_t to its target
    end if
    if locally o_a is non-aliased then
        MergeAliasIntoTarget(o_a, o_t)
    else if locally ∃o_e such that o_a → o_e then
        if o_e ≠ o_t then
            if o_t ≠ max(o_e, o_t) then
                swap(o_t, o_e)
            end if
            MergeAliasIntoTarget(o_e, o_t)          ▷ resolve alias chain of o_a, o_e, and o_t
        end if
    else                                            ▷ locally o_a is not known
        locally record o_a → o_t
        vl(o_a) = ∅
    end if
    vl(o_a) = vl(o_a) ∪ vr(o_a)
end function
```

Expulsion: Propagation of Deletions and Selective Sync

The system propagates file and folder deletion updates among devices as one type of object update. Users are additionally permitted to specify those files and folders which they would not like to sync to a particular device (but which remain synchronized among all other devices). Common to both features is the method of labeling a file or folder as *expelled*. Among other data stored in the logical representation of the file system (e.g. name, object id), the system stores a boolean expelled flag for each object.

Expulsion: Selective Sync

Initially all new objects are flagged as false (*admitted*). To unsync a file at the local device, the expelled flag is set to true, and the file is consequently deleted from the physical file system, without incrementing versions in the consistency algorithm (i.e. do not create an update about this deletion). To unsync a directory, it is flagged as expelled, along with all descendent files and directories in the logical file system. The aforementioned physical files/directories are deleted. If the parent directory of an object is expelled, then that child object must necessarily be expelled as well. No versions are incremented in the consistency algorithm when a folder and its children are expelled; this is a local operation, not to be shared with other devices.

Deletion propagation via a logical trash folder

Unlike selective sync, where an object is physically deleted on only one device, but synced among all others, the system additionally supports the propagation of deletions. This feature relies on a special directory that is expelled in every store, known as the trash folder, shown in Figure 6. Pictured in the left box of the figure is a tree representation of the logical directory tree. Tree nodes are logical objects representing directories or files; nodes with children are necessarily directories on the physical file system. For example, the node labeled “Root” is the root directory of the store,

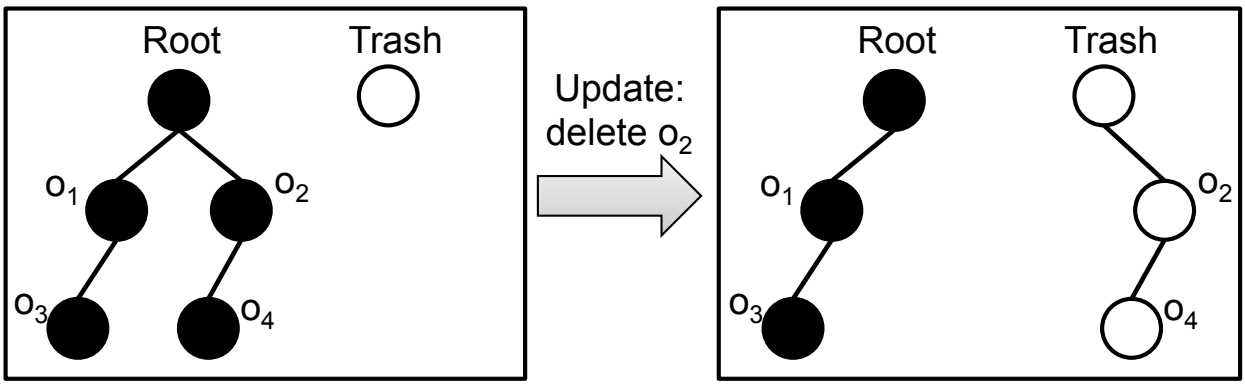

Figure 6: Updating the logical file system on deletion by moving to the Trash Folder

with two children directories o_1 and o_2, which have one child object each, o_3 and o_4, respectively. Empty (or white) nodes represent expelled objects, and full nodes are not expelled. The trash folder is always marked expelled, thus it appears only in the logical file tree, not on the physical file system. In every store, on every device, the trash folder has the same object id.

To propagate a deletion update for an object, the object is moved to the trash folder. This is demonstrated in Figure 6. In the initial state (on the left) directory o_2 was under the root directory. The system is notified that, locally, o_2 has been deleted. Thus o_2 is logically moved under the trash folder. Because children of an expelled folder are also expelled, o_2 and its children are expelled. As with all object moves, the logical movement of o_2 to the trash folder warrants a version increment for o_2 in the consistency algorithm. Via the Collector algorithm, remote devices will collect the update that o_2 has been moved under the trash folder. Thus they will set the expelled flag on o_2 after moving it under the trash folder, and delete it from the physical file system. Hence, object deletions are propagated by logically moving objects under the known, expelled trash folder.

Migration: moving files across store boundaries

Recall the concept of a store from the base system [1] and Background section of this patent. A store defines a set of users who are sharing a directory and its contents. Moving an object between stores deserves special consideration. The system supports the ability to delete files when moved out of a store, or move files among stores, depending on the context. The problem is illustrated in Figure 7. Additionally, the system maintains cross-store version history, providing a causal history for an object that crosses store boundaries, as seen in Figure 8.

Figure 7 shows the state of two stores, S_1 and S_2 on four devices, d_1, d_2, d_3, d_4, after moving an object between stores. Devices d_1 and d_2 are subscribed to both stores. Device d_3 is subscribed to S_1 only, and d_4 to S_2 only. Initially object o_1 is under the root directory of store S_1, and all devices are consistent with this state. Device d_1 moves o_1 into store S_2. The system supports the following state transitions when each of devices d_2, d_3, and d_4 receives the update of the cross-store object movement.

- on d_2, the object is physically moved, without deleting and re-downloading the content of o_1;
- on d_3 the object is physically deleted; and

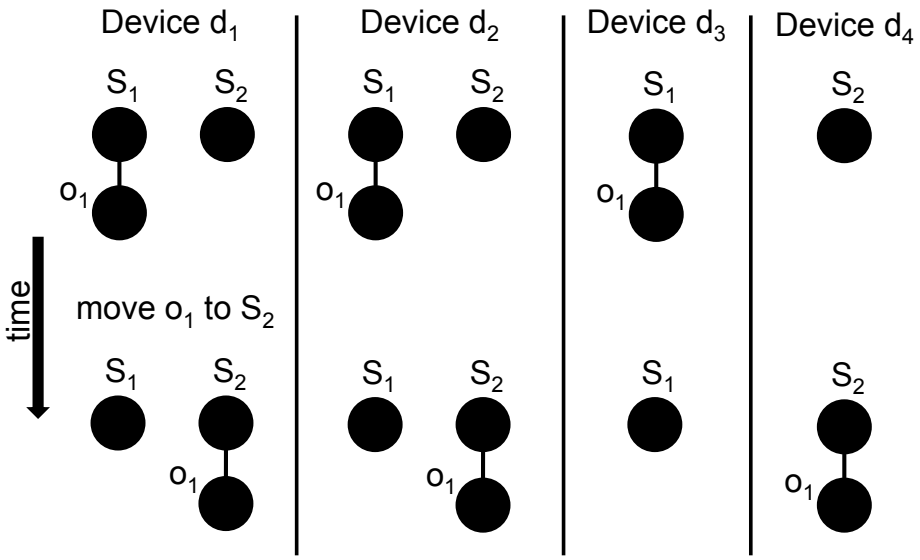

Figure 7: Expected behavior following a migration across stores for devices subscribing to a different set of stores

- on d_4 the object is downloaded and physically created.

The Collector, Expulsion, and update propagation algorithms are store-centric, thus what should be a simple move operation between stores on d_2 could be naively implemented atop these algorithms as separate deletion and creation operation. A device receiving the deletion and creation updates would thus naively re-download the file, even if the content had not changed. Through the method of *migrating* a logical object between stores, the system avoids naively deleting the object from S_1, then re-downloading the object into S_2.

Figure 8 illustrates the goal for cross-store version history. Initially, object o_1 is consistent under store S_1, on both devices d_1 and d_2. During a network partition, device d_2 modifies the content of o_1 (indicated by the modified pattern of the node), but leaves the object in store S_1. Concurrently, device d_1 moves the object to store S_2 with the original content. The network partition then disappears and the two devices propagate their updates. By maintaining the identity of o_1 across stores, and maintaining the version history, a final state can be achieved where o_1 is under store S_2 with the new content applied while it was under S_1. Without tracking identity and the cross-store version history, there would be two files, one in each store, and only one would have the updated content.

The system observes the migration of the object id o_1 and maintains the consistency algorithm version history of the file, through (i) respecting the invariant that a given object can be admitted in only one store at any time (as in the Expulsion algorithm), and (ii) an extension to the versioning system of the consistency algorithm, called *immigrant versions*.

State Change When Instigating Migration

Figure 9 shows the logical state change after an object o_1 is physically moved between stores S_1 and S_2 on the local device. Initially object o_1 with name n is under the root folder of S_1, and S_2 has no child object. Following the physical move, under store S_1 the object is effectively deleted as

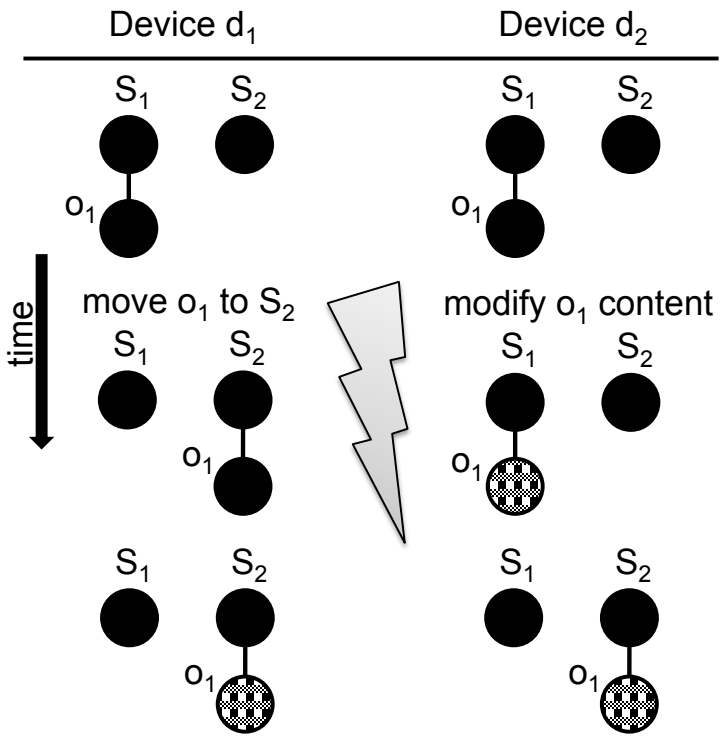

Figure 8: An update to the content of o_1 under store S_1 is successfully propagated, despite o_1 being moved to store S_2 concurrently

indicated in the Expulsion section of this patent, by logically being moved under the trash folder, and thus expelled. Notably, under the trash folder, the name of the object is the store id of S_2, the target store to which o_1 was moved. Under store S_2, object o_1 is created in the admitted state. As with the usual consistency algorithm, these two logical state changes generate two updates which are propagated to other devices:

- o_1 was moved under the S_1 trash folder with name S_2, and
- o_1 was created under the S_2 root folder with name n.

The physical object maintains its logical object id across stores in an effort to easily identify migration, and maintain its version history despite store migrations.

On Deletion Update

Consider a device which subscribes to store S_1; it will receive the first update, that o_1 was moved to the trash folder. Because the new name of o_1 under store S_1 identifies the store to which o_1 *emigrated*, the device receiving this update can infer the new store of o_1. The following pseudo-code enumerates the steps taken to migrate an object o on receipt of a deletion update under store S.

function HANDLEDELETEUPDATE(S: store id, o: object id: n: name)
  if o is locally admitted in S and has not been emigrated and n encodes a store id **then**
    $S_{target} \leftarrow$ decoded store id from n

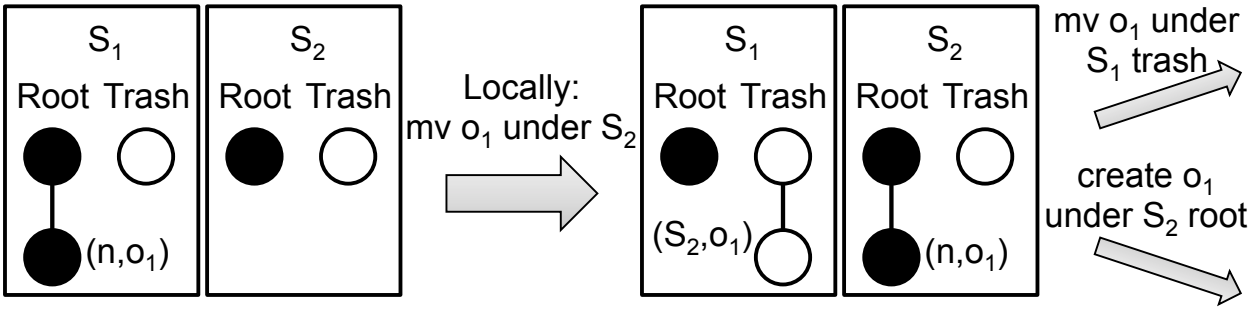

Figure 9:

```
        attempt download of S_target and its ancestor stores from remote
        if store S_target is not present locally then
            return                    ▷ do not migrate as the target store cannot be downloaded
        end if
        if o is a file then
            download o under S_target from remote
        else
            download o and its children under S_target from remote
        end if
    else
        process the deletion update for o as usual
    end if
end function
```

This method to handle migration-induced deletions determines the target store of the object to be migrated, then defers to the handler of creation updates (to follow). Because migrated objects keep the same logical identifier, once the deletion handler has determined the target store, it can simply request the object under that store. A non migration-induced deletion will be handled by the Expulsion algorithm described earlier.

On Creation Update

Now consider a device which subscribes to store S_2; it receives the second update, that o_1 was created under S_2 with name n. In a typical work flow the object is physically downloaded, but to avoid redundant transfers, the local device first determines whether o_1 is admitted in any other store. If o_1 is admitted in another store, the local device migrates o_1 under S_2 by physical file movement. The following pseudo-code details the steps taken to migrate an object o on receipt of a creation update under store S. The annotation (S, o) is used to identify object o under S. The creation of immigrant versions is explained in the following subsection.

```
function HandleCreationUpdate(S: store id, o: object id: n name)
    if ∃S_source where o is locally admitted in store S_source then
        for each content conflict branch b of (S_source, o) do
            copy modification time, file length, content hash of b to (S, o)
            physically move the branch to its corresponding location under S
```

end for
create immigrant versions from (S_{source}, o) to (S, o)
move o in S_{source} under the trash folder
logically rename o to an encoding of S
else
process the creation update for o as usual
end if
end function

Note that the creation update handler concludes by deleting o from the source store, and recording its migrated target store. This action implicitly will create a new version update for (S_{source}, O) on the local device, which will be propagated to other devices. However all devices that subscribe to the target store S will perform the same action, generating false version conflicts. The base system [1] discusses an approach to automatically resolve such conflicts.

Immigrant Versions: Cross-Store Consistency

The background section of this patent briefly summarizes update propagation in the base system [1]. This section is mainly concerned with pull requests. Naively, a device could respond to a pull request by sending its entire local set of version vectors. However, the two devices may share many of those versions, resulting in much redundant bandwidth waste. The base system further proposes the concept of stability of version vectors, by defining a *knowledge vector*, present locally on a device, one knowledge version vector for every store. All integer counts below the knowledge vector of the local device are assumed to be stable—no new version needs to be requested whose integer count is below the knowledge vector. Because of this invariant, when issuing a pull request, a device X can send its knowledge vector to device Y, and device Y can respond with only those versions which are above the given knowledge vector. Additionally, device Y responds with its knowledge vector after the version exchange so that device X can increase its own vector accordingly. See the prior patent for examples and details.

The migration of an object across stores requires special consideration with regard to the knowledge vector. The system cannot simply move versions for an object in one store to its new target store, as this naive approach could move a version from the source store to the target store that is under the knowledge vector of the target store. This in turn would break the invariant that all versions below a knowledge vector are stable, and thus we could no longer assume that pull-based update propagation will guarantee to receive all new updates from the pulled device.

To ensure that pull-based update propagation guarantees the propagation of all updates in the face of store migration, this patent presents *immigrant version vectors*. Whereas each regular version vector (*native* version vector) is associated with the update of one logical object, an immigrant version vector is associated with the migration of a native version vector. The concurrency control subsystem thus has two version management systems, one for native versions which track object updates, and one for immigrant versions, which track native version. Immigrant versions similarly have a knowledge vector, and stability of immigrant versions. When an object o is locally migrated from store S_s to S_t on device d, a new immigrant version is created for o on d, recording the version of o that was migrated from S_s. As part of the pull-based update propagation strategy, immigrant versions are requested that are above the immigrant knowledge vector. If a received immigrant version was previously unknown to the local device, then the native version tracked by the immigrant version is persisted in the local device's native version table. The immigrant version subsystem can

thus insert native versions under the native knowledge vector, but no native versions are at risk of loss because of cross-store object migration.

File Conflict Presentation and Resolution in the UI

When a file is updated concurrently on two separate devices, a file conflict results. In the base system [1], the updates made locally to this file o on a given device are present on the local file system, and any downloaded conflicting versions of the file are logically recorded as *branches* of o. That prior patent provides more details, e.g. how to apply subsequent updates to specific branches. In this patent, the extended system additionally provides a graphical user interface (GUI) to present the conflicting branches to the user. The GUI visualizes these conflict files branching from a common ancestor, including the device and user who contributed to creating each branch. Users are provided a button to open any files in a conflict branch to view its contents; these files are stored in a hidden directory. To resolve a file conflict, users can open all conflict branches, and their main local copy, and manually correct the conflict. Users are presented a button to choose branches to discard. Discarding a branch of object o creates a deletion update (see the Expulsion section) for that branch, and this update will be propagated to all other devices sharing that file.

Team Server: Multiple User Accounts on One Device

For a team of users who wish to use a shared device to back up their files, the system provides a Team Server type account which permits multiple stores from multiple users on the same device. This Team Server account would be among those in the Access Control Lists (ACLs) for all stores shared by the team members, including the root store for all team member users. The Team Server account is concerned with stores, and thus need only synchronize one copy of a store as it is shared by multiple users. Whereas the support of file migration across stores for a single-user device necessitates the invariant that an object id can be admitted in only one store on a device, on a Team Server, an object id may be admitted in multiple stores, because Team Servers are not concerned with migration.

One source of backup is often insufficient, thus the system offers a self-replicating server farm, via multiple Team Servers. In one embodiment, one Team Server account is installed on n devices, and the algorithms of this patent and the base system [1] synchronize the files and folders of those servers, providing a replication factor of n. In another embodiment, multiple devices are installed with the same account, but each device stores a partition of the total team space requirements, permitting a scalable replication factor from 1 to n. This is achieved atop the Selective Sync algorithm, by setting the "expelled" flag on some devices.

Collaborative version history

Whenever a remote file update is downloaded (including modifications and deletions), the local copy of the file is saved to a special location, creating a local version history for every file. Through a GUI display, users can restore files from this version history. The version history is truncated after some time period.

One can also consider the global version history for a file in the distributed system of devices, which includes these local saves across all devices. In another embodiment, each version history file

is tagged with its corresponding version vector, and the user id and device id which instigated the update. Users can visualize the system aggregate version history tree, and if a desired file version is not locally present, the device can request that version from the device that performed the backup. When requesting remote version history, the local device can avoid presenting duplicate version history items by detecting duplicate version vectors.

Sync Status

To inform users whether a file on their local device has been synchronized with other devices, the system provides a method to determine the *sync status* of each file and folder by comparing version vectors across multiple devices. In one embodiment, given an object and two devices sharing it, the method reports whether both devices have the same version or a different version. The sync status is recorded as a set of devices that are in or out of sync with the local device. To show meaningful status for a directory, the sync status is recursively aggregated from all descendent files and folders. Via a file-system GUI icon overlay, three possible sync status states are presented to the user for each file or folder:

- in-sync: all devices are in sync
- partial sync: at least one device is in sync and at least one is out of sync
- out of sync: all devices are out of sync

In one embodiment, the method takes a centralized structure where a single server stores the hash of the current version vector of every object for every device. On update, these version vectors are broadcasted to those client devices interested in the given object, ensuring the Sync Statuses remain up-to-date. In another embodiment, a decentralized structure is employed, where client devices record the version vector of every object and every device, or some partition of that data.

References

[1] W. Wang and Y. Sagalov, "System and method for cloud file management," US Patent Application US 2012/0 005 159 A1, Jan 5, 2012.

[2] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *IEEE Trans. Softw. Eng.*, vol. 9, no. 3, pp. 240–247, May 1983.

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, 1970.