Net Integration Technologies, Inc.

WvSync

Tea, Earl Grey, Hot: Replication with WvSync

Dave Coombs
v1.1 - Release: 2004 07 09

FOR PUBLIC RELEASE

Table of Contents

1 Disclaimer for Flamers....3

2 Expository Story....3

3 Replication Elaboration....4

3.1 Things You Need, Guaranteed....4

3.2 Reproachful Approaches....4

3.3 Optimize the Compromise....5

4 Solution Execution....6

4.1 Embrace the Interface....6

4.2 Precision Decisions....7

4.3 A Fix for Conflicts....8

4.4 A Screwball Protocol....8

5 Future Features....9

6 Download the Code....10

7 Acknowledgements....10

1 Disclaimer for Flamers

This paper contains many exaggerations, numerous factual errors, and several outright lies, because some of the code I will be demonstrating next month at the UKUUG conference has not yet been written. For now I hope to distract you with whimsical rhyming headings.

Readers are encouraged to check `http://open.nit.ca/wiki/?page=WvSync` for updates, corrections, complete rewrites of this paper, etc.

2 Expository Story

One day, a day like any other, Avery and I were MBWAing our R&D office in Montreal. We noticed something that hadn't really seemed clear before: something ridiculous like 75% of our developers were working on some form or another of data replication. People were writing backup code; other people were backing up these backups elsewhere; a group of people were co-operatively writing code to synchronize calendar and contact information between Outlook, Evolution, and Horde; other people were synchronizing user accounts and DNS records between servers; someone was replicating MySQL databases; someone else was synchronizing files, and one truly crazy individual was working on a distributed filesystem.

These are all largely separate projects, but these people were all doing largely the same thing, with different bugs and/or design flaws and/or tradeoffs in different places.

We decided this was crazy.

This paper describes some universal requirements for any replication project, the inherent compromises involved, and puts forth an "optimal compromise" leading to a general solution that should be useful in any project.

WvSync is the start of an open-source implementation of this solution to the general replication problem. I will describe some design elements of WvSync, hopefully without lying too much.

3 REPLICATION ELABORATION

3.1 Things You Need, Guaranteed

In any project involving replication, the requirements are bound to be pretty similar. So far as we can tell, they'll always look something like this:

- I have some data in some format all in one place.
- I have big performance requirements, and/or I need reliability, and/or I want to conserve bandwidth, and/or I want to reduce latency.
- Therefore I should have multiple copies of the data, or different parts of the data, in different places.
- Therefore I somehow need to choose which data to replicate, how many times, how often, and to which places.
- And now I need some software to actually do the replication in my particular format.

Individually these requirements may not seem that bad, but put them all together and it starts to become rather implementation-specific, dependent on what you are trying to do with what particular kind/size/shape of data.

So everyone may have these requirements, but everybody does something different with them in their own project, as follows.

3.2 Reproachful Approaches

The compromises begin. Compromises are fine, of course, but keeping it generalized is hard. Given the above requirements, the approach taken for any given project will vary, but is always the result of constraints on a combination of only the following:

- CPU power
- Storage space / memory
- Network bandwidth
- Network latency
- Reliability requirements

For example, a backup tool has constraints on CPU power and storage space, and it may choose to waste storage space to conserve CPU power. A network backup tool is restricted mainly by network bandwidth. A calendar/contact synchronizer is restricted by client-side storage *and* network bandwidth/latency, and might choose to waste space to reduce latency. A distributed filesystem is severely restricted by network bandwidth and latency. A web cache should probably waste network bandwidth (and storage, of course) and even perfect reliability to reduce latency. These are all compromises that trade away something (usually disk space or bandwidth, which are cheap) to achieve something more important to the user.

Eventually the compromise usually comes down to deciding which data you keep a copy of ("make available offline" in Windows-speak) and which data you'll go to the server for when you need it ("don't make available offline"). Some protocols implement only one or the other. Simple caching is in between, but mostly doesn't continue to work when you go "offline."

3.3 Optimize the Compromise

We believe there is, in fact, a general solution to the replication problem with an optimal set of tradeoffs, in which you don't have to choose betwen "online" and "offline".

"But there can't be an optimal tradeoff!" you say. "Otherwise, why would you use different tradeoffs with each project?"

Because we're lazy, and we didn't want to implement the whole general algorithm *each* time. Implementing the most practical tradeoff for a particular project is less work, if you're only working on one project. Now that we have a whole slew of replication projects, it makes sense to implement the right solution once. Like all optimal-tradeoff solutions, it involves cheating.

The optimal solution needs to follow these rules:

- Replicate a "set of named objects" of variable size. URLs are named objects. So are files. So are MySQL records. So is everything.
- Provide a way to convert the named objects into the things we actually want, like URLs, files, or actual MySQL records.
- Provide a special API for accessing the replicated objects whenever anybody wants them. (This is the hardest part, but we can skip it initially, so that's OK.)
- Waste all idle bandwidth. Differentiate between *critical* operations (which the user or application is actively waiting for) and *non-critical* operations (which nobody asked for, but we can do if we're bored). This allows us to use negative latency.
- Waste all available storage space. Cache *everything* and let the replication protocol tell you when your cache is invalid. Find a way to flush your cache when somebody important needs your storage space. Feel free to remember all sorts of metadata to help with decisions later.
- Waste all available CPU time. Take note of which data is used most often, and use this to decide what to flush from the cache and when, and what to replicate, how many times, and how soon. Use something like librsync to waste cheap CPU cycles instead of sending whole objects whenever a tiny change is made.

4 Solution Execution

WvSync is a general purpose replication library written in C++, using WvStreams (for easy network communication etc), UniConf (for easy metadata storage), librsync (for easy rsync-like deltas), and DaZuko (because it's neat).

Picture rsync, and add the following:

- Bidirectional synchronization. Change files on one computer, change other files on another computer, and it will copy the changes in both directions.
- Synchronization of arbitrary data types, not just files. You have to write some C++ glue to support your own data type, but it's well abstracted, and not that hard.
- Online mode. When WvSync is up and running, any changes made will be *instantly* copied to the other computer(s).

4.1 Embrace the Interface

The lowest-level class in WvSync is the `WvSyncObj`, which is a basic interface for a named generic object that can be replicated. You can derive one for a file, a database record, a hash-table dictionary, or whatever, but you'll have to provide your own methods for serializing and deserializing into and out of a `WvBuf` buffer.

`WvSyncObj` itself provides wrapper functions that use librsync to produce signatures, deltas, and patched reconstructions of the data.

As of this writing, `WvSyncFile` is the only existing useful subclass of this. It overrides/implements the following methods:

- `getdata()` reads a given number of bytes from a given offset of the object, and returns them in a `WvBuffer`. This is used when calculating the librsync signature of the object, and when reconstructing the object if a delta patch is received.
- `findlastmodtime()` finds and returns the last-modification timestamp of the object. This is easy in the case of a file; it's the mtime. This is used for making decisions about what needs to be sent in what direction.
- `findmeta()` finds and returns metadata for the object. In the case of a file, I'm using the mode, the uid, and the gid, formatted into a string. For your own data type, it can be whatever you need. `applymeta()` is related; it takes metadata received from a remote node and applies it to the local object.
- `makecopy()` makes a copy of the current object under a new name. This is used in case of a synchronization conflict. As described below, one node will have its object renamed, and making a copy lets us take advantage of librsync deltas.

4.2 Precision Decisions

In general, we need to send an object from one WvSync node to another if it has changed on the first node since the last synchronization. How do we know if it has? We have to keep a list *for each synchronization partner* of every single object, and the status of that object as of the last synchronization with that partner. Useful things to remember for each object are the last-modified time, the md5sum of the librsync signature, possibly even the librsync signature itself (it's large, but it allows for negative latency in some cases), and the user-defined metadata for the object. Clearly this uses a lot of space and CPU time, but it provides usefulness and minimizes bandwidth usage and latency.

It is the `WvSyncArbiter` class's job to determine what needs to happen. For any synchronizable object, the `arblocal()` method first determines what has happened to the object since the last synchronization. Possibilities are:

- `ok`: Nothing has happened since the last synchronization.
- `new`: The object has been added.
- `mod`: The object has changed.
- `del`: The object is no longer there.
- `meta`: The object contents are the same, but the metadata has changed.

For each object, the applicable one of these five states is sent to the other node, along with current librsync signature, metadata, and last-modification time. The other node does the same calculation based on the object *it* has by the same name, and sends back its version of the same data.

Then the `arbremote()` method is called on both sides. The goal of this function is to arrive at the same decision on both sides.

Here are the cases to consider, assuming WvSync nodes named A and B:

- A `ok`, B `ok`: Nothing to synchronize. Go on to the next object.
- A `ok`, B `new`/`mod`: B wins; copy to A.
- A `ok`, B `del`: B wins; delete on A.
- A `ok`, B `meta`: B wins; apply metadata to A.
- A `new`/`mod`, B `new`/`mod`: Classic conflict case. See below.
- A `new`/`mod`, B `del`: A wins; forget the deletion and copy to B.
- A `new`/`mod`, B `meta`: A wins; forget new metadata and copy to B.
- A `del`, B `del`: Nothing to do. Remove from both lists and continue.
- A `del`, B `meta`: Better to be safe, so copy back to A with new meta.
- A `meta`, B `meta`: Conflict case. See below.

There are numerous special cases too confusing to cover here. (What if a file turns into a directory? How should a failed transfer affect an object's stored state?) There is magic to deal with much of this, but it's more important to understand the basics.

4.3 A Fix for Conflicts

A conflict arises when the same object has been changed differently on both ends of the synchronization.

The goals for conflict resolution in WvSync are twofold:

1. Don't lose anybody's changes.
2. Make it obvious if there was a conflict.

If we don't do number 1, then we are rudely clobbering people's changes and annoying them. If we don't do number 2, then people will get confused and clobber each other's changes by accident, annoying each other. The only workable solution has to accomplish both of the above goals.

The short answer is: when two people change the same object, pick a winner, and rename the other one to a unique name. *Both* objects, the one with the real name, and the renamed one, will show up on *both* ends being synchronized. The renamed object name could (depending on the type of the object) include the server name and timestamp that it was changed, so the two users who made the changes can co-operate and merge the changes themselves. This is the only reliable way.

Which version is the "winner"? It's almost an arbitrary decision. We choose the one with the later timestamp.

4.4 A Screwball Protocol

The protocol used by WvSync aims to reduce latency by allowing multiple objects to be synchronized simultaneously. It must be stateful per object, due to the multiple phases of a librsync transfer. Multiple in-flight objects are distinguished by affixing each command with a tag identifying the object, somewhat similar to the way commands are associated with their responses in IMAP.

It is easiest to describe the protocol with an example using most of the commands that currently exist. Here, then, is a sample of one object being synchronized. It has been changed on A, and remains unchanged on B.

```
A:    xx CHDIR /some/directory
B:    xx OK
A:    o1 IHAVE mod 01234...BCDEF 1089437106 somemetadata object_name
B:    o1 OK 1 ok 12345...ABCDE 1089437100 somemetadata
      o1 SIG 4096
A:    o1 OK send 4096 bytes
B:    {sends 4096-byte librsync signature}
A:    o1 DELTA 1024
B:    o1 OK send 1024 bytes
A:    {sends 1024-byte librsync delta}
B:    o1 DONE
A:    o1 OK done
```

Other objects can be processed at the same time. For example, they could be tagged o2 and o3. In the future, in cases where the signature or (more likely) the delta are huge, they will be able to be broken up into smaller pieces and sent that way, to allow other objects to be processed simultaneously.

Before A sends the IHAVE command, it will have run the `arblocal()` function and determined the information it needs to send along, including an MD5 hash, the last-modification time, and the metadata.

When B receives the IHAVE command, it runs `arblocal()` on its copy of the same object and passes back the associated information in the OK response. It also runs `arbremote()` at the same time with both copies of the information, to determine whether it needs to request the object (ie: a delta) from A. The '1' on the OK line indicates that it does. This value is included in the OK line to assist A in detecting a conflict. When A receives this OK line, it runs `arbremote()` as well.

Now both sides know which way the transfer will be occurring. B must send its librsync signature for the object to A, so that A can calculate a librsync delta. In this example, the signature is 4096 bytes long.

A receives this, generates a delta from it, and sends it to B. In this example, the delta is 1024 bytes long.

If B is able to successfully patch its version of the object, it sends the DONE command, which A has to acknowledge with an OK, and the synchronization is complete.

If there had been a conflict, a winner would have been chosen, and one object would have been copied to a new name before the synchronization. The object with the regular name is synchronized as normal, and the protocol is instructed that it should also subsequently synchronize this new object with the new name.

If there is a failure at any stage of the synchronization, ERROR is sent instead of OK, and the other side can issue an ABORT command to skip the current object.

5 Future Features

- Step 1: Finish implementing the above for file synchronization, and make a standalone program that does just that.
- Step 2: Add a sort-of "online" mode after a full synchronization, where any changes to files (use DaZuko!) or objects are noticed and synchronized immediately.
- Step 84657: Provide the special API for accessing objects in both online/offline modes. Cache like crazy. Pull a librsync-based distributed filesystem out of my ear.
- Step 3275284654: I'm rich!

6 Download the Code

You can get WvSync from its project home page at:

```
http://open.nit.ca/wiki/?page=WvSync
```

7 Acknowledgements

Man, nothing rhymes with acknowledgements.

Thanks to Avery Pennarun for defining the General Replication Problem and Solution.

Thanks to Martin Pool for librsync, and Andrew Tridgell for doing all the math.

Thanks to Pierre Phaneuf for harassing me about renames.