Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks

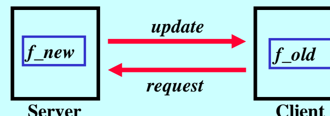
Torsten Suel

CIS Department
Polytechnic University

Joint work with *Patrick Noel* (1) *and Dimitre Trendafilov* (2)

Current Affiliations: (1) National Bank of Haiti, (2) Google Inc.
Work performed while at Polytechnic University

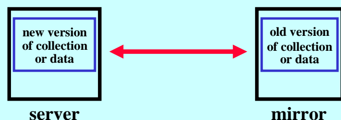
File Synchronization Problem



- clients wants to update outdated file
- server has new file but does not know old file
- update without sending entire new file *(using similarity)*
- *rsync*: file synchronization tool, part of Linux
- note: files are unstructured *(record/page oriented case is different)*

If server has copy of both files
➡ local problem at server *(delta compression)*

Main Application: Mirroring/Replication

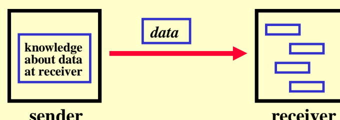


- to mirror web and ftp servers *(e.g., downloads: tucows, linux)*
- to synchronize data between PCs *(e.g., at work and at home)*
- *our motivation:* maintaining web collections for mining
- *our motivation:* efficient web page subscription
- currently done using *rsync* protocol and tool
- can we significantly improve performance?
- recall: server has no knowledge of old version

General Background and Motivation:

- network links are getting faster and faster
- but many clients still connected by fairly slow links
- 56K and cell modems, USB, cable, DSL, 802.11
- bandwidth also a problem in P2P systems (large data)
- how can we deal with such slow links in ways that are transparent to the user?
  - at the application layer (e.g., http, mail)
  - application-independent at IP or TCP layer
- applications: *web access, handheld synchronization, software updates, server mirroring, link compression*

Setup: Data Transmission over Slow Networks



- sender needs to send data set to receiver
- data set may be "similar" to data the receiver already holds
- sender may or may not know data at receiver
  - sender may know the data at receiver
  - sender may know something about data at receiver
  - sender may initially know nothing

Two Standard Techniques:

- caching: *"avoid sending same object again"*
  - widely used and important scenarios
  - done on the basis of objects
  - only works if objects completely unchanged
  - how about objects that are slightly changed?
- compression: *"remove redundancy in transmitted data"*
  - avoid repeated substrings in data (Lempel-Ziv approach)
  - often limited to within one object
  - but can be extended to history of past transmissions
  - often not feasible to extend to long histories and large data
  - overhead: sender may not be able to store total history
  - robustness: receiver data may get corrupted
  - does not apply when sender has never seen data at receiver

Types of Techniques:

- file synchronization
- delta compression
- database reconciliation *(for structured data or blocks/pages)*
- substring caching techniques *(Spring/Wetherall, LBFS, VB Cach.)*

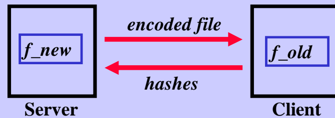
Major Applications:

- data mirroring *(web servers, accounts, collections)*
- web access acceleration *(NetZero, AOL 9.0, Sprint Vision)*
- handheld synchronization *(Minsky/Trachtenberg/Zippel)*
- link compression *(Spring/Wetherall, pivia)*
- content distribution networks
- software updates *(cellphone software upgrades: DoOnGo, others)*

File Synchronization: State of the Art

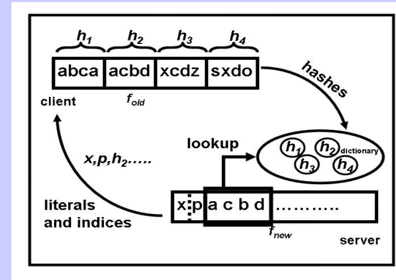
- the *rsync* algorithm
- performance of *rsync*
- theoretical results
  - file similarity measures
  - theoretical bounds *(Orlitsky 1984, Cormode et al, Orlitsky/Viswanathan)*
- improved practical algorithms
  - *Cormode et al (Soda 2000)*
  - *Orlitsky/Viswanathan (ISIT 2001)*
  - *Langford (2001)*

The *rsync* Algorithm



- clients splits *f_old* into blocks of size *b*
- compute a hash value for each block and send to server
- server stores received hashes in dictionary
- server transmits *f_new* to client, but replaces any *b*-byte window that hashes to value in dictionary by reference

The *rsync* Algorithm (cont.)



- simple, widely used, single roundtrip
- optimizations: 4-byte rolling hash + 2-byte MD5, *gzip* for literals
- choice of block size problematic (default: max{700, sqrt(n)})
- not good in theory: granularity of changes, sqrt(n lg n) lower bound

Some performance numbers for *rsync*

- *rsync* vs. *gzip, vcdiff, xdelta, zdelta*
- files: gnu and emacs versions *(also used by Hunt/Vo/Tichy)*

	gcc size	emacs size
total	27288	27326
gzip	7479	8191
xdelta	461	2131
vcdiff	289	1821
zdelta	227	1431
rsync	964	4451

Compressed size in KB (slightly outdated numbers)

factor of 3-5 gap between rsync and delta !

Theoretical Results:

Formal File Similarity Measures:

- Hamming distance
  - inserting one character gives large distance
- Levenshtein distance
  - insertion, deletion, change of single characters
- Edit distance with block moves
  - also allows blocks of data to be moved around
- Edit distance with block copies
  - allows copies, and deletions of repeated blocks

Some Known Bounds

- Suppose files of length n with distance k
- lower bound *Omega(k lg n)* bits (even for Hamming)
- general framework with asymptotically optimal upper bounds for many cases *(Orlitsky 1994)*
- one message not optimal, but three messages are
- impractical: decoding takes exponential time!
- recent practical algos with provable bounds:
  - Orlitsky/Viswanathan 2001, Cormode et al 2000, Langford 2001
  - *lg n* factor from optimal for measures without block copies
  - not fully implemented and optimized

Recursive approach to improve on *rsync*

- How to choose block size in *rsync* ?
- *rsync* has cost at least *O(sqrt(n lg n))*
- Idea: recursive splitting of blocks
  Orlitsky/Viswanathan, Langford, Cormode et al.
- Upper bound *O(k lg^2(n))* (does not work for block copies)
- This paper:
  - optimized practical multi-round protocols
  - several new techniques with significant benefits
- Ongoing work:
  - show optimal upper bounds
  - new approaches for single- and two-round protocols
  - applications

Our Contribution: *zsync*

- server sends hash values to client (unlike in *rsync*)
- client checks if it has a matching block
  if yes: "client understands the hash"
- server recursively splits blocks that do not find match

New Techniques in *zsync*

- use of optimized delta compressor: *zdelta*
- decomposable hash functions
  - after sending parent hash, need to only send one child hash
- continuation hashes to extend matches to left and right
- optimized match verification
- several other minor optimizations

Our Framework: *zsync*

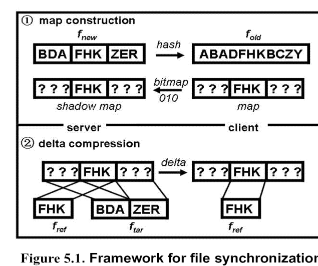
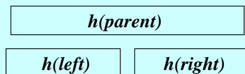


Figure 5.1. Framework for file synchronization.

- multiple roundtrips (starting with 32K block size, down to 8 bytes)
- servers sends hashes, client confirms and verifies
- both parties maintain a "map" of the new file
- at end, server sends unresolved parts (delta-compressed)

Techniques and Ideas:

- decomposable hash functions
  - after sending parent hash, need to only send one child hash



decomposable: *h(right) = f (h(parent), h(left))*

- continuation hashes
  - to extend hashes to left or right using fewer bits
  - standard hashes expensive: are compared to all shifts in *f_old*
  - continuation hashes: only compared to one position
    "is this a continuation of an adjacent larger hash?"
  - might give improved asymptotic bounds …

Techniques and Ideas:

- optimized match verification
  - server sends a fairly weak hash to find likely matches
  - client sends one verification hash for several likely matches
  - closely related to group testing techniques and binary search with errors

server: *"here's a hash. Look for something that matches"*

client: *"how about this one? Here are 2 additional bits"*

server: *"looks good, but I need a few more bits to be sure"*

client: *"OK, but to save bits, I will send a joint hash for 8 matches to check if all of them are correct"*

server: *"apparently not all 8 are correct. Let's back off"*

client: *"OK, how about this joint hash for two matches?"*

An example:

Experimental evaluation with decomposable hash

- gcc data set
- initial block size 32K, variable minimum block size

Basic results for *emacs* data set

Continuation hashes

Optimized Match Verification

Best Numbers

	gcc	emacs
number of files	1,002	1,286
uncompressed	27,288	27,326
gzip	7,563	8,577
rsync default	964	4,452
rsync best	839	3,731
zdelta	227	1,431
our results	336	2,343

- > 50 roundtrips
- but within 1-2% with about 10 roundtrips
- not per-file roundtrips!

Results for Web Repository Synchronization

- updating web pages to new version
- for search/mining or subscription

	2 days	20 days	72 days
uncompressed size	143 MB	143 MB	140 MB
number of files	10,000	10,000	10,000
number of changed files	2,818	3,747	5,127
rsync (default)	5,339	7,766	11,770
zdelta	1,479	2,170	3,879
our results	2,062	3,623	6,703

Table 6.2. Cost of updating a web collection using various methods, for various update frequencies. The cost is in KB for 10000 web pages.

Current and Future Work

- Building *zsync* tool and library
- Improving asymptotic bounds
  - continuation hashes seem to do the trick
- New approaches to file synchronization
- Applications
- Substring caching, record-based data, and other techniques