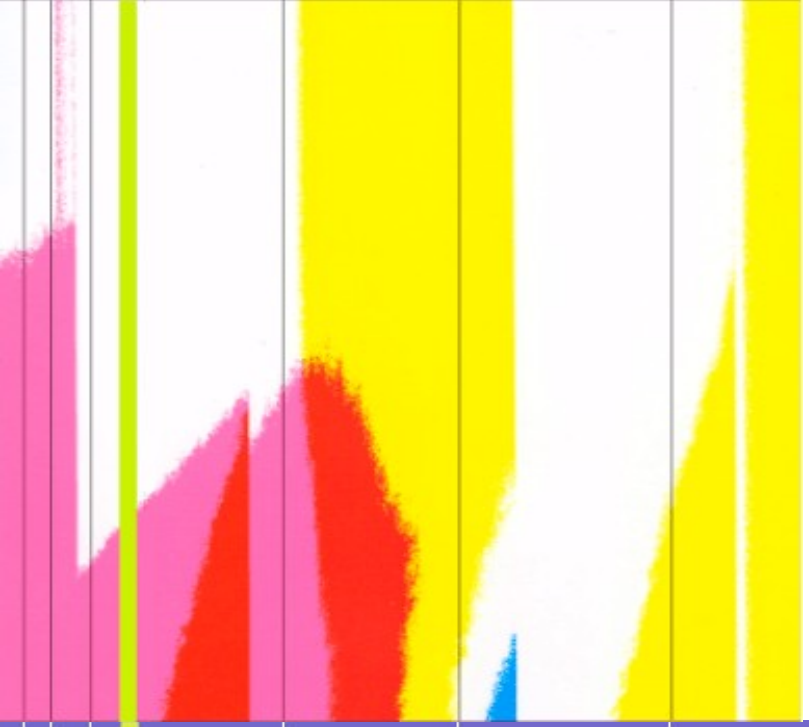


BlobSeer

How to Enable Efficient Versioning for Large Object Storage under Heavy Access Concurrency

Bogdan Nicolae, Gabriel Antoniu, Luc Bougé

Paris Project, IRISA Rennes
University of Rennes 1, INRIA, ENS Cachan

P2P approaches are becoming popular for more and more application classes

- More than 80% of data in circulation is unstructured [Grimes, 2008]
  - Free-form text: web pages, text documents, log files
  - Multimedia files: audio, video, images

- MapReduce [Google, 2004] and Hadoop [Apache, 2008]
  - Designed to deal with unstructured data
  - Transparent high-level data processing frameworks
  - Adaptation attempts for P2P environments [Marozzo, 2008]
  - Data stored in a huge blob (binary large object)

- Scientific applications

Data storage and access faces new challenges

- Mutable data
  - Poor support in traditional storage systems: GFS, HadoopFS
- Huge data size, fast generation rates
  - PB scale storage is necessary to cope with size
  - Order of TB/week not uncommon
- Heavy access concurrency
  - Thousands of clients accessing data simultaneously
- Versioning
  - Support for roll-back
  - Cheap branching

Our approach: BlobSeer

- Blob is fragmented into small equally-sized pages
  - Allows huge data amounts to be distributed all over the peers
  - Avoids contention for simultaneous accesses to disjoint parts of the data block
- Metadata is added to locate pages that make up the blob
  - Fine-grained and distributed as well to avoid contention
- Versioning
  - New pages corresponding to an update / append are generated instead of overwriting old pages
  - Metadata is enriched to incorporate the update
  - Both the old and the new version of the blob are accessible through the associated metadata

Scenario

- Online digital picture enhancing service
  - Users upload, apply filters and download back pictures
  - Pictures include metadata about camera type and settings

- Data mining: avg picture quality for camera type
  - Need to dynamically parse both metadata and image
  - MapReduce approach
    - Store all pictures in a huge blob
    - Use a past blob versions as snapshots for data mining
    - Allow users to update and add new pictures in the background by generating new blob versions

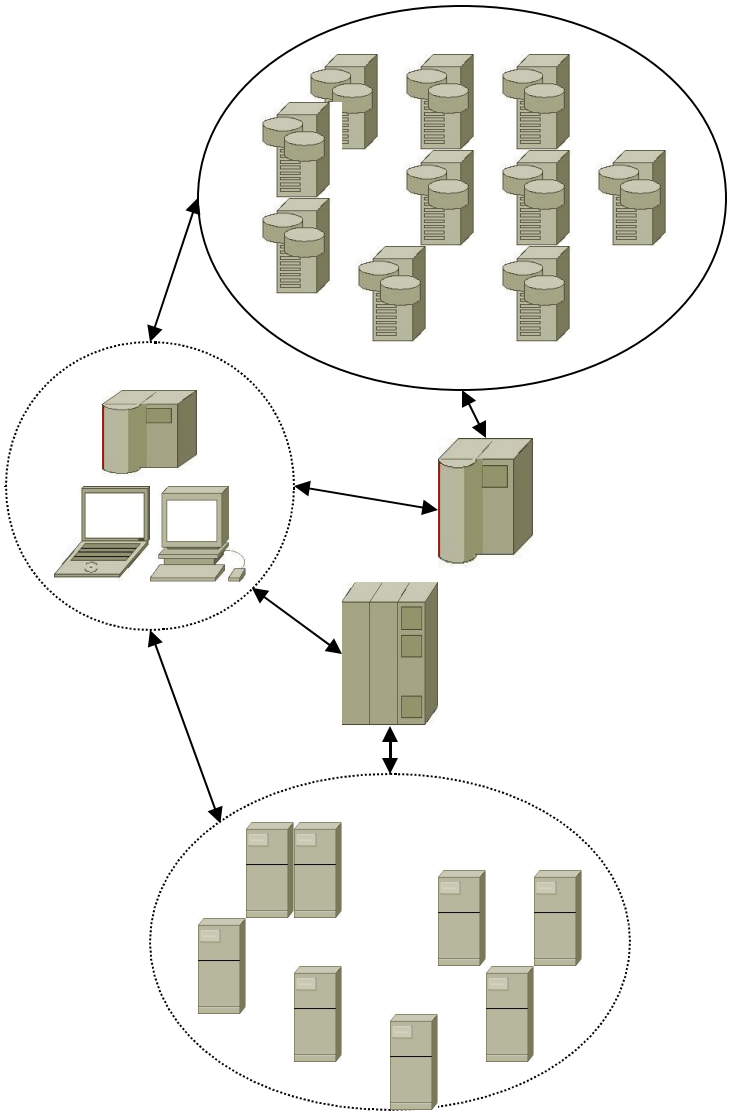
User interface

- Set of primitives:
  - id = CREATE()
  - vw = WRITE(id, buffer, offset, size)
  - va = APPEND(id, buffer, size)
  - READ(id, v, buffer, offset, size)
  - v = GET_RECENT(id)
  - size = GET_SIZE(id, v)
  - SYNC(id, v)
  - bid = BRANCH(id, v)

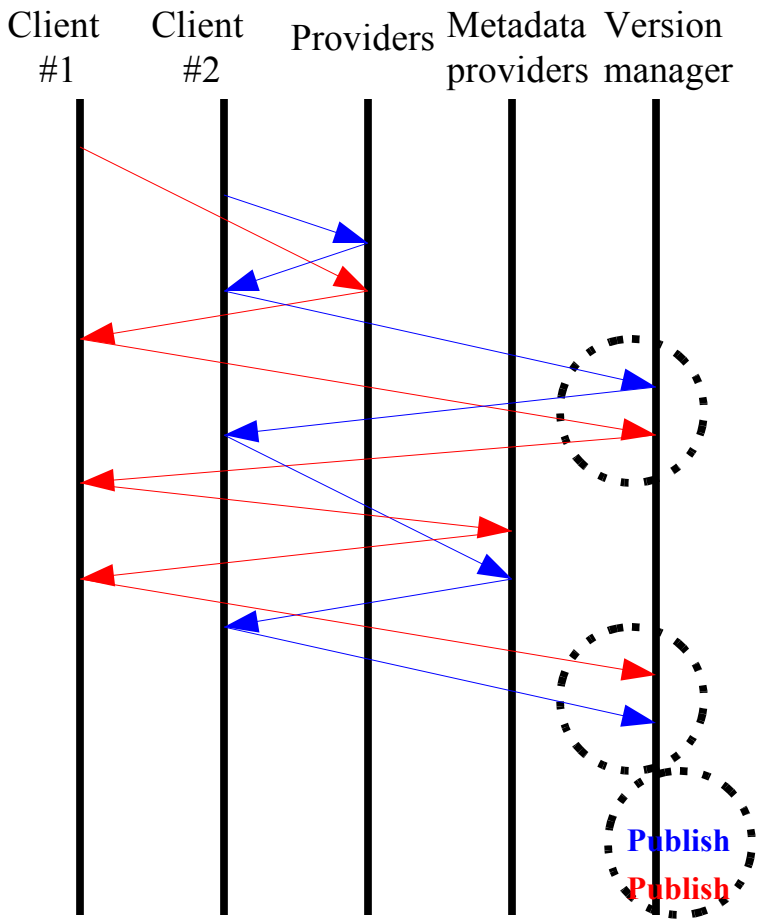
- (offset, size) may cover multiple pages

Architecture

- Clients
  - Perform fine grain blob accesses
- Providers
  - Store the pages of the blob
- Provider manager
  - Monitors the providers
  - Ensures a load balancing strategy of pages among providers
- Metadata providers
  - Store the segment tree associated to the blob
- Version manager
  - Ensures concurrency control



How to enable efficient versioning



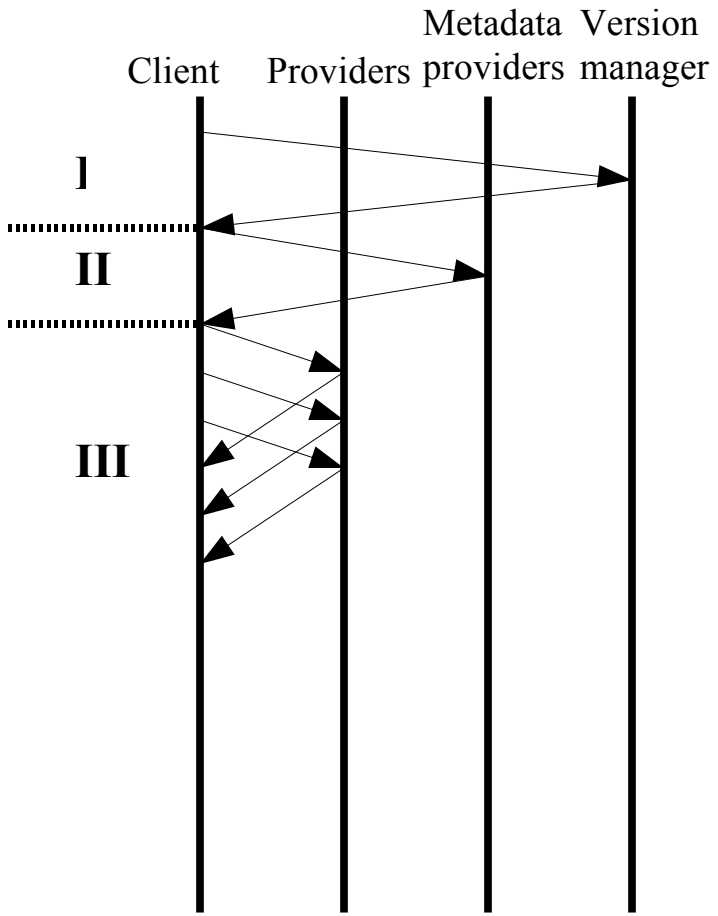
- Pages are written concurrently by the clients
- Versions are assigned in the order the clients finish writing
- Metadata is written concurrently by the clients
- Versions are published in the order they were assigned

How does a read work?

- I: optionally ask the version manager for the latest published version

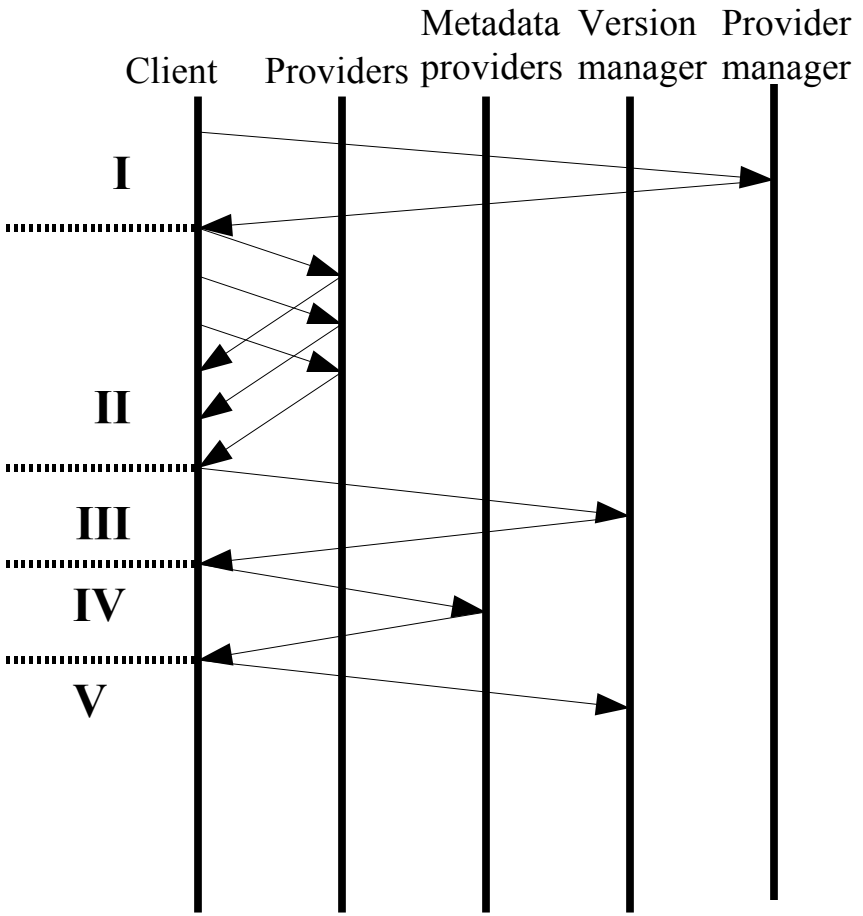
- II: fetch the corresponding metadata from the metadata providers

- III: contact providers in parallel and fetch the pages in the local buffer



How does a write / append work (1) ?

- I: get a list of providers that are able to store the pages, one for each page
- II: contact providers in parallel and write the pages to the corresponding providers
- III: get a version number for the update
- IV: add new metadata to consolidate the new version
- V: report the new version is ready for publication.



How does a write / append work? (2)

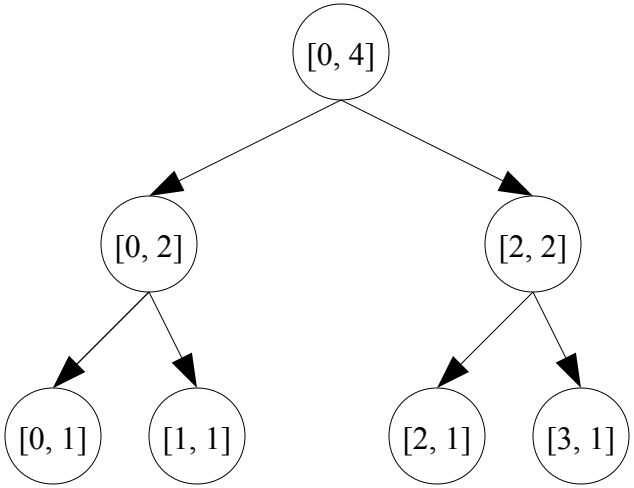
- The version manager holds a queue with all pending writes and ensures versions are published in the order they were assigned

- Priority is given to the least loaded provider when assigning a provider to store a page

- Multiple versions of the same page may be stored on potentially different providers

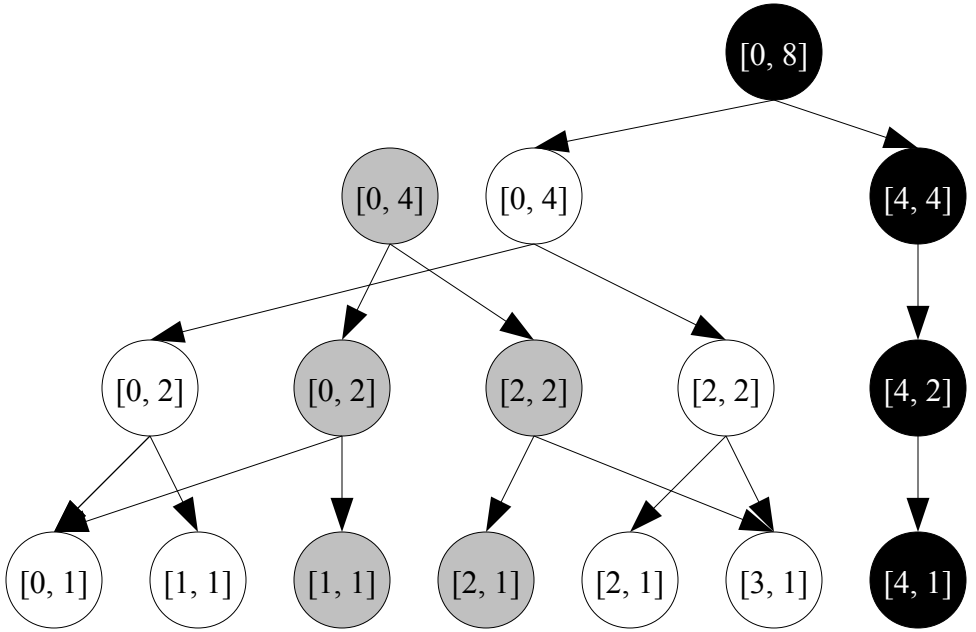
Metadata (1)

- Organized as a segment tree
- Each node covers a range of the blob identified by (offset, size)
- The first/second half of the range is covered by the left/right child
- Each leaf corresponds to a page and holds information about its location



Metadata (2)

- Each node holds versioning information
- Write/Append
  - Add leaves and build subtree up to the root
  - The tree may grow one level
- Read: descend from the root towards the leaves



Metadata (3)

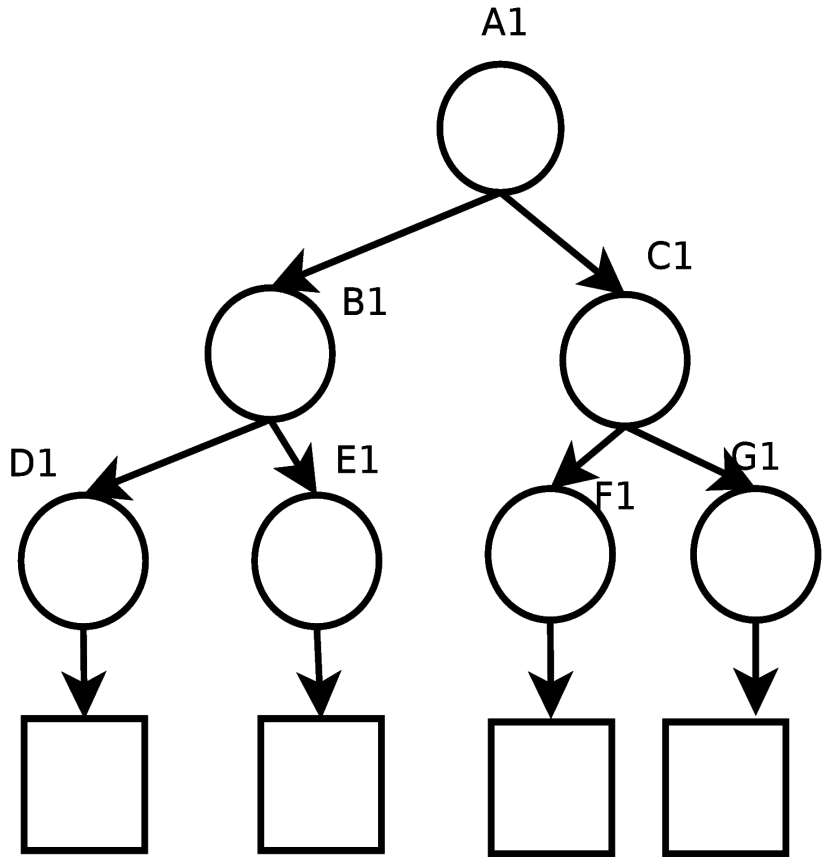
- Fully concurrent metadata consolidation is possible:
  - A node in the metadata tree may have a child labeled with a previous version
  - Such a child may not have been added yet because it belongs to a write (or append) in progress
  - The version manager holds a queue with all pending writes/appends
  - When the client asks the version manager for a new version (after writing the pages), the latter can predict all such children and pass them along

- Tree nodes are distributed among metadata providers

- Clients can fetch multiple nodes in parallel

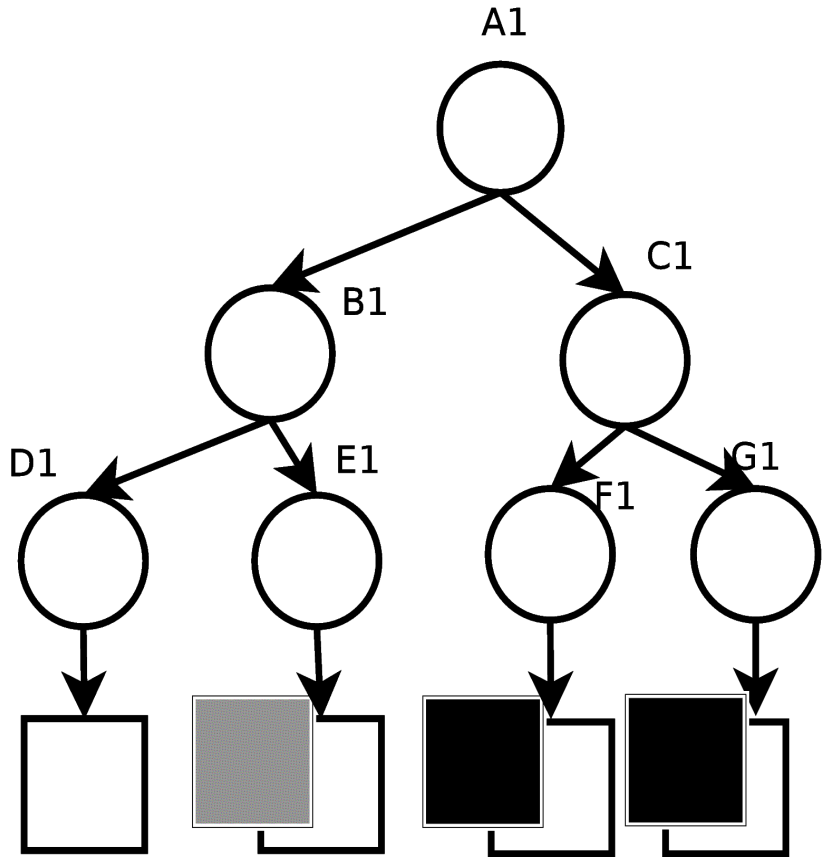
How concurrent writes work: example

- Initial version: v = 1
- 2 concurrent writers: gray and black
- Both write their pages independently
- Gray is first, it is enqueued on the versioning manager and assigned version v2, black follows and gets v3
- Both write independently the metadata tree nodes: black is faster and links to (the not yet created node) B2
- First to finish is black, it is marked ready
- Next is gray, being the first means its root gets published and it is dequeued
- Finally black gets first in the queue and and will be published



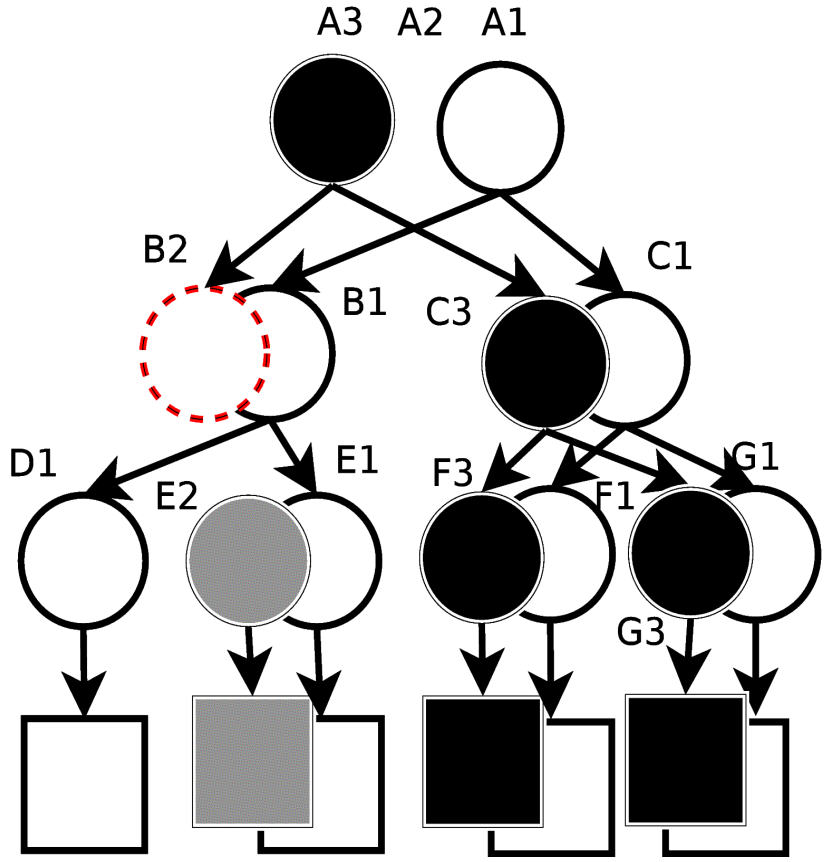
How concurrent writes work: example

- Initial version: v = 1
- 2 concurrent writers: gray and black
- Both write their pages independently
- Gray is first, it is enqueued on the versioning manager and assigned version v2, black follows and gets v3
- Both write independently the metadata tree nodes: black is faster and links to (the not yet created node) B2
- First to finish is black, it is marked ready
- Next is gray, being the first means its root gets published and it is dequeued
- Finally black gets first in the queue and and will be published



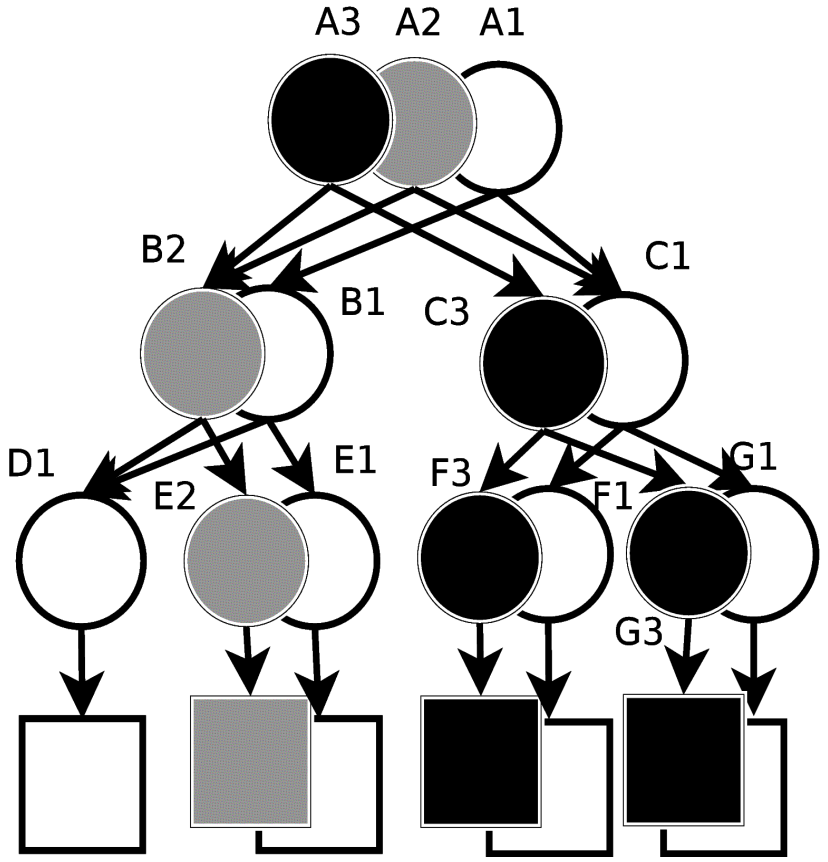
How concurrent writes work: example

- Initial version: v = 1
- 2 concurrent writers: gray and black
- Both write their pages independently
- Gray is first, it is enqueued on the versioning manager and assigned version v2, black follows and gets v3
- Both write independently the metadata tree nodes: black is faster and links to (the not yet created node) B2
- First to finish is black, it is marked ready
- Next is gray, being the first means its root gets published and it is dequeued
- Finally black gets first in the queue and and will be published



How concurrent writes work: example

- Initial version: v = 1
- 2 concurrent writers: gray and black
- Both write their pages independently
- Gray is first, it is enqueued on the versioning manager and assigned version v2, black follows and gets v3
- Both write independently the metadata tree nodes: black is faster and links to (the not yet created node) B2
- First to finish is black, it is marked ready
- Next is gray, being the first means its root gets published and it is dequeued
- Finally black gets first in the queue and and will be published



Comparsion with other systems

	Read concurrency	Write concurrency	Append concurrency	Versioning
Traditional FS	-	-	-	-
Lustre, PVFS	X	X	-	-
GFS, HadoopFS	X	-	X	-
DeepStore	-	-	X	X
BlobSeer	X	X	X	X

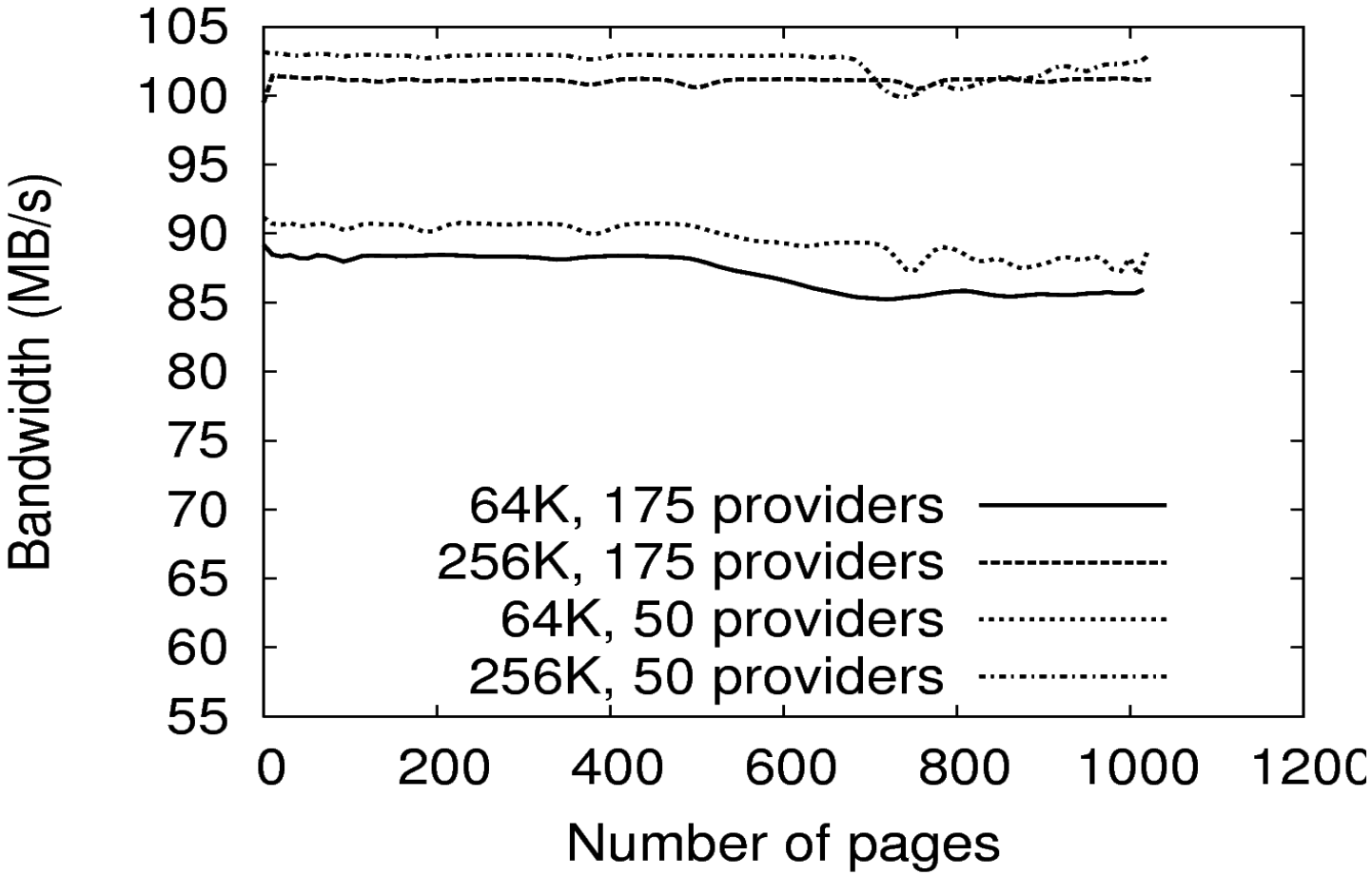
Experimental setup

- Implementation details
  - Custom DHT
  - Boost C++ collection, ASIO

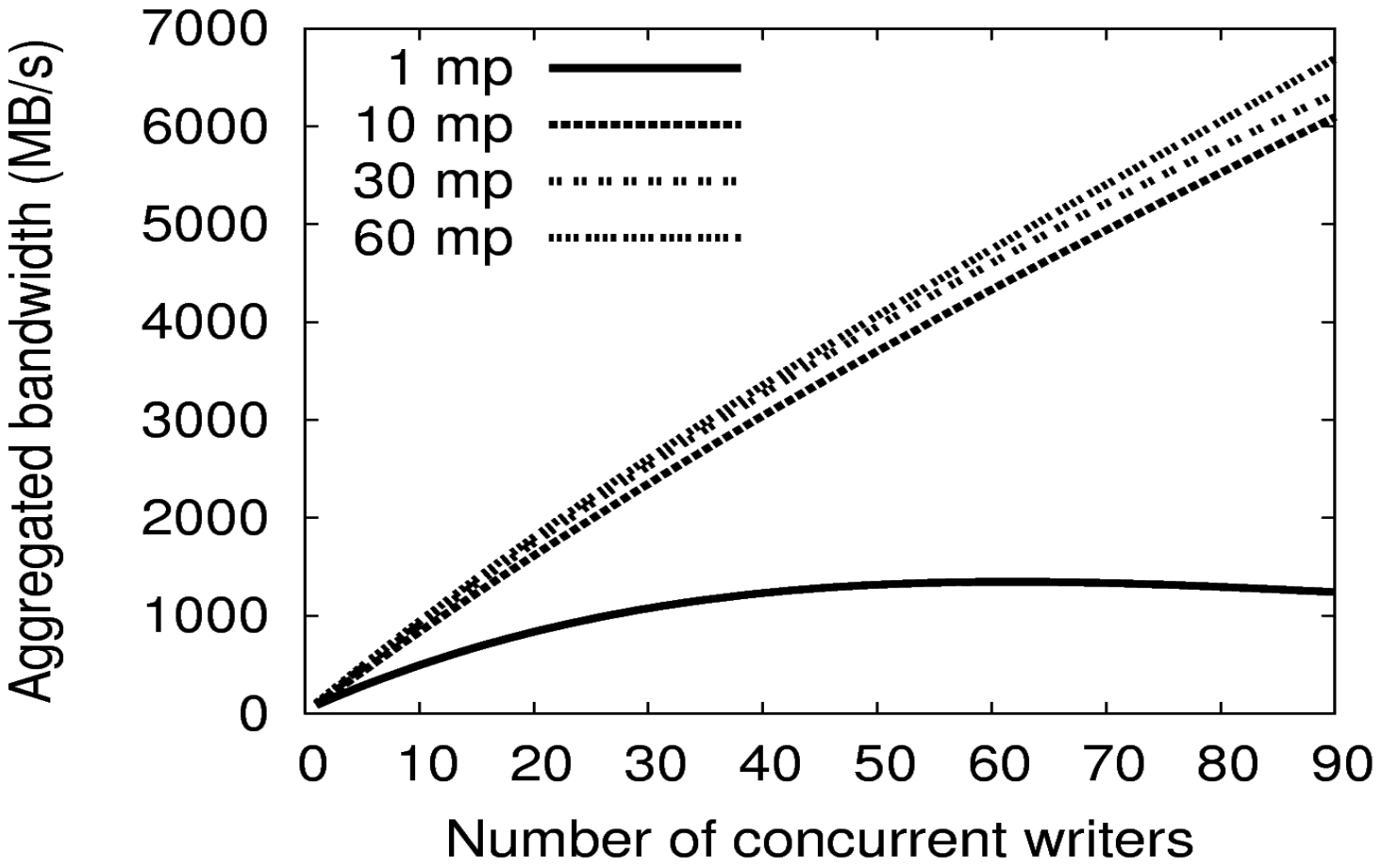
- Our machines: Reservation on Grid'5000 platform
  - 175 nodes
  - Pentium-4 CPU@2.6Ghz, 4GB RAM, Gigabit Ethernet
  - Measured bandwidth: 117.5 MB/s for MTU=1500B

- 3 sets of experiments:
  - Append bandwith for a single client
  - Aggregated write bandwidth under heavy concurrency
  - Sustained read bandwidth under heavy concurrency

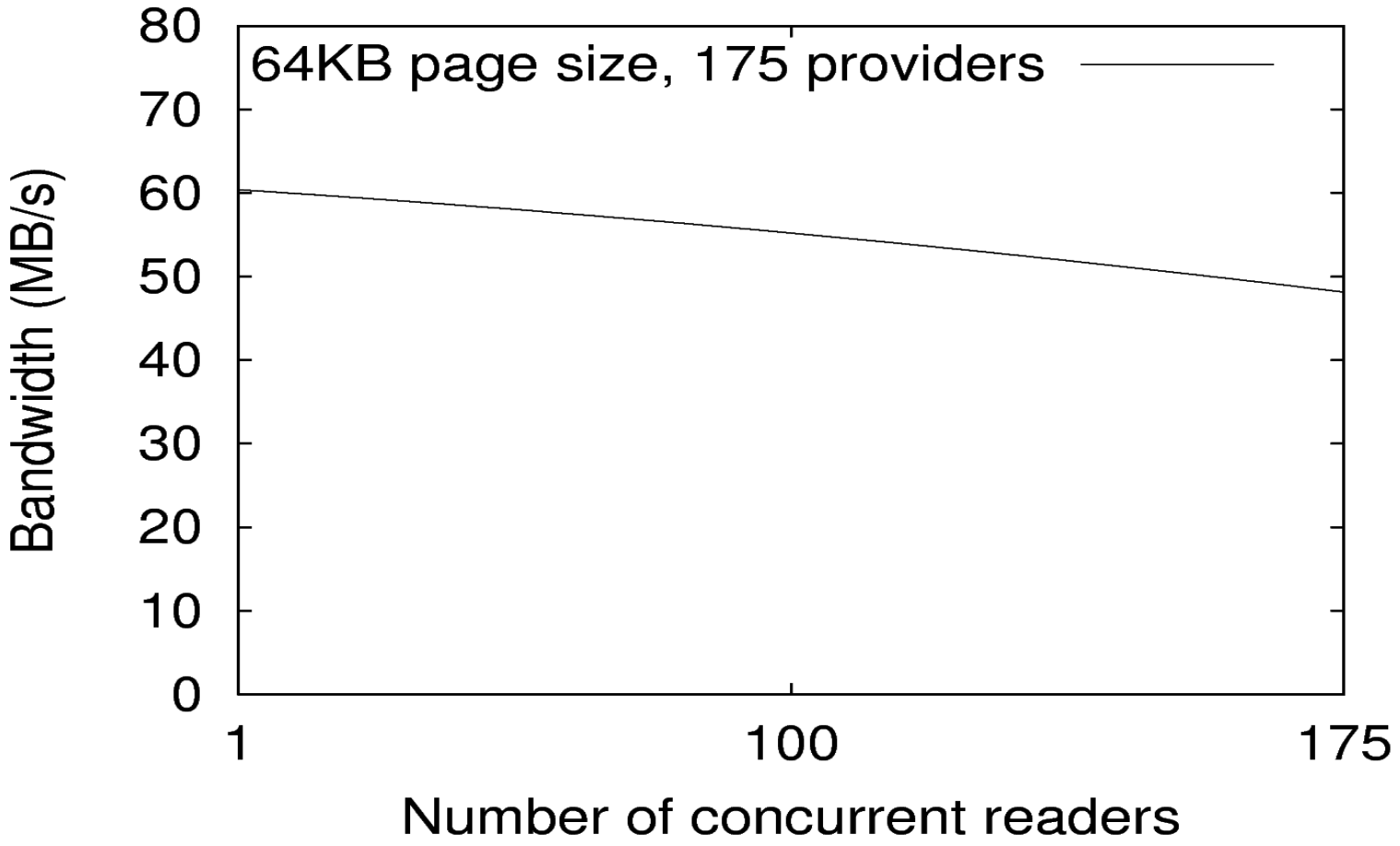
Results (1) – Append performance: single client



Results (2) – Aggregated write bandwidth



Results (3) – Sustained read bandwidth



Conclusion

- Addressed the problem of efficent versioning for blobs under heavy access concurrency in P2P environments

- Proposed a system offering support for:
  - Huge blob size: order of TB
  - Small updates: order of MB
  - Fine granularity, page size: order of KB
  - High degree of concurrency

- Experiments show promising results

Future work

- In progress
  - Fault tolerance
    - Smart replication strategies through global behavioural modeling
  - Advanced monitoring
    - Experimenting with load balancing strategies
  - BlobSeer as a storage layer in Postgres

- Planned:
  - Fault tolerance
    - Decentralized version and provider manager
  - Transactional support:
    - Export a filesystem interface with transaction management
    - Snapshot isolation by making use of multi-versioning support