Plasma

Distributed file system
Map/Reduce

Gerd Stolpmann, November 2010

Plasma Project

- Existing parts:
  - PlasmaFS: Filesystem
  - Plasma Map/Reduce
- Maybe later:
  - Plasma Tracker
- Private project started in February 2010
- Second release 0.2 (October 2010)
- GPL
- No users yet

Coding Effort

- Original plan:
  - PlasmaFS: < 10K lines
  - Plasma Map/Reduce: < 1K lines
- However, goals were not reached... Currently:
  - PlasmaFS: 26K lines
  - Plasma Map/Reduce: 6.5K lines
- Aiming at very high code quality
- Plan turned out to be quite ambitious

PlasmaFS Overview

- Distributed filesystem:
  - Bundle many disks to one filesystem
  - Improved reliability because of replication
  - Improved performance
- Medium to large files (several M to several T)
- Full set of file operations

lookup/open	creat
stat	truncate
read/write (random)	mkdir/rmdir
read/write (stream)	chown/chmod/utimes
link/unlink/rename	

- Access via:
  - PlasmaFS native API
  - NFS: PlasmaFS is mountable
  - Future: HTTP, WebDAV, FUSE

PlasmaFS Features 1

- Focus on high reliability
  - Correctness → code quality
  - Replication
    - data (blocks)
    - metadata (directories, inodes)
  - Automatic failover (*)
  - Transactional API: Sequences of operations can be bundled into transactions (like in SQL)

start → lookup → read → write → commit

  - ACID (atomicity, consistency, isolation, durability)

(*) not yet fully implemented

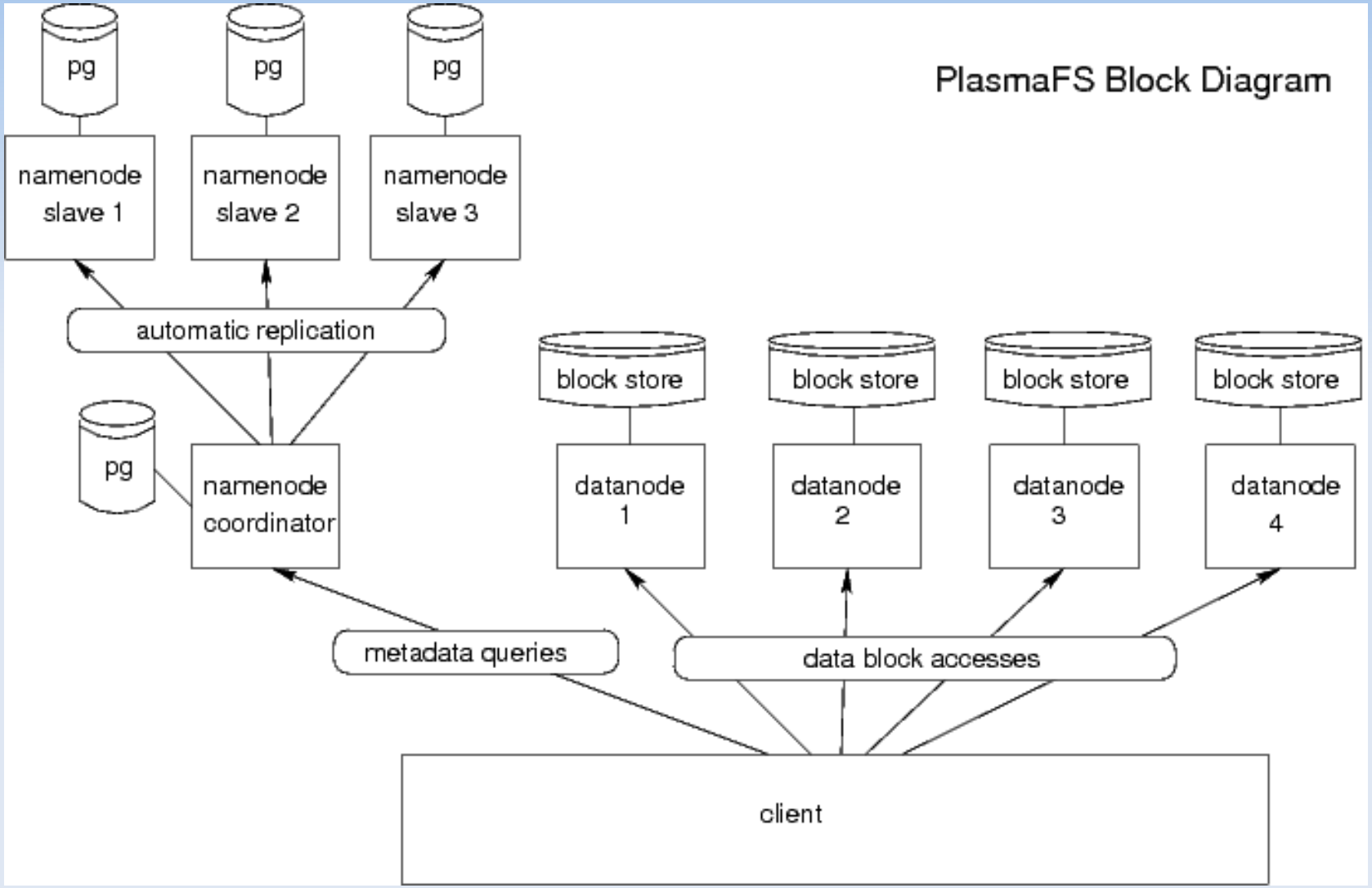
PlasmaFS Features 2

- Performance features
  - Direct client connections to datanodes
  - Shared memory for connections to local datanodes
  - Fixed block size
  - Predictable placement of blocks on disks

    Blocks are placed on disk at datanode initialization time
  - Contiguous allocation of block ranges
  - Sequential reading and writing specially supported

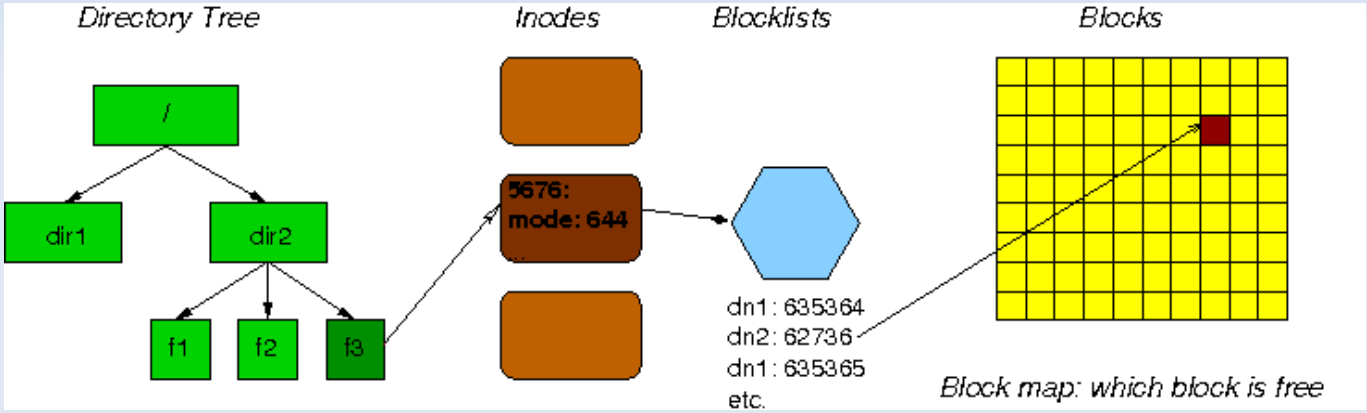
    Or better: random r/w access is supported but not fast
  - Design focuses on medium-sized blocks: 64K-1M

PlasmaFS: Architecture



PlasmaFS: Namenodes 1

- Tasks of namenodes:
  - Native API
  - Manage metadata
  - Block allocation

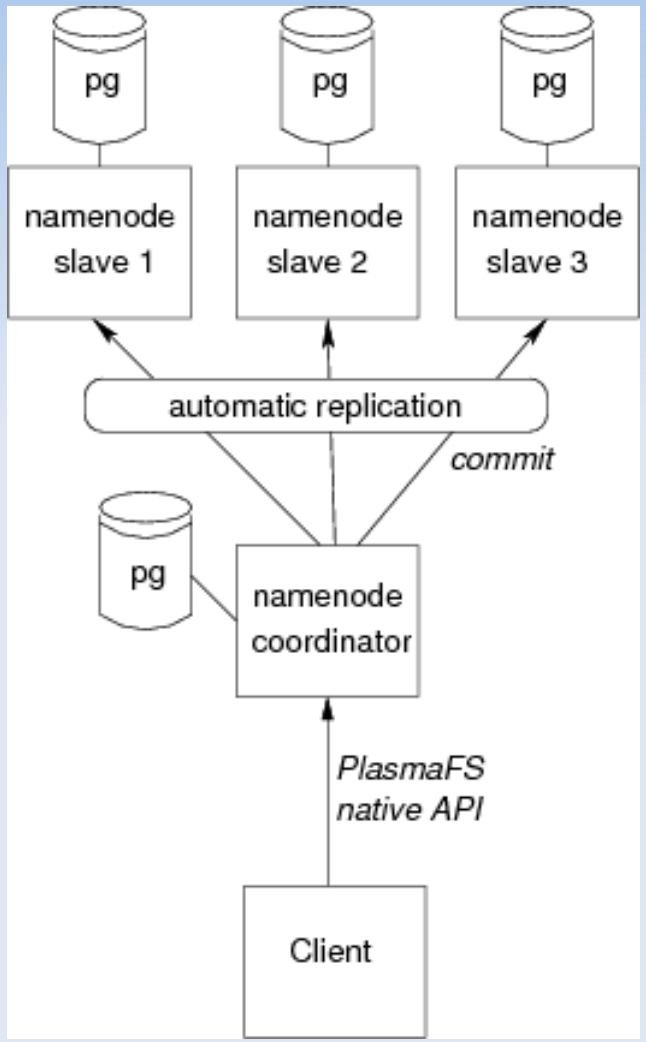


  - Manage datanodes (where, size, identity)
  - Monitoring: which nodes are up, which down (*)
- Non-task: Namenodes never see payload data

(*) not yet fully implemented

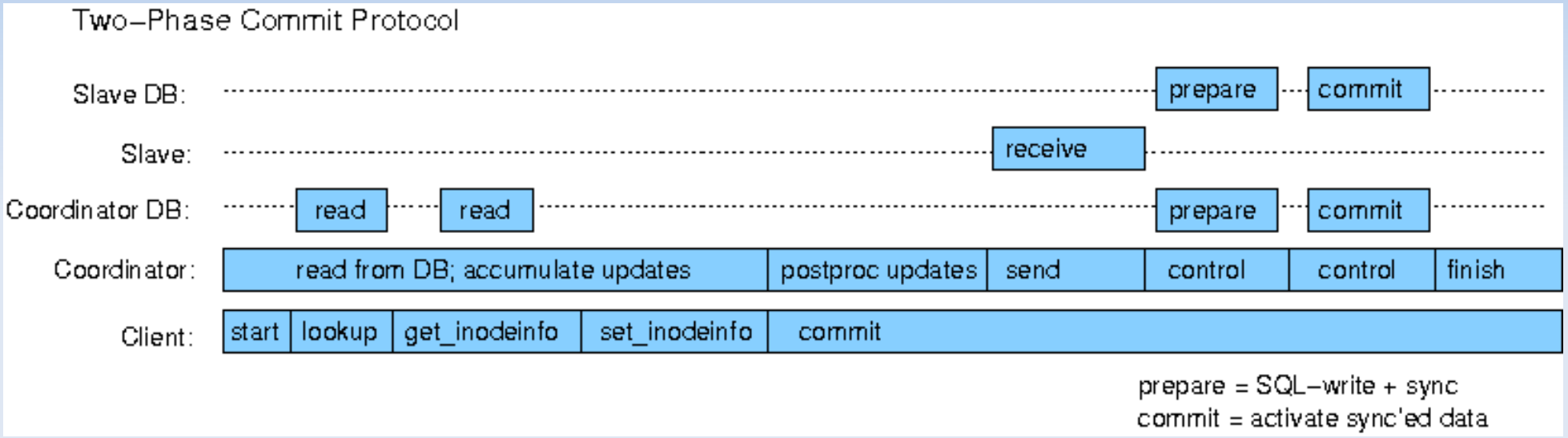
PlasmaFS: Namenodes 2

- Metadata is stored in PostgreSQL databases
  - Get ACID for free
  - Why PostgreSQL, and not another free DBMS? Has to do with replication
- Replication scheme:
  - master/slave: one namenode is picked at startup time and works as master (coordinator), the other nodes are replicas
  - Replication is ACID-compliant: committed replicated data is identical to the committed version on the coordinator. Replica updates are not delayed!
  - Two-phase commit protocol → PostgreSQL



PlasmaFS: Namenodes 3

- Two-phase commit protocol
  - Implemented in the inter-namenode protocol



  - PostgreSQL feature of prepared commits is needed
- Only partial support for getting transaction isolation
  - → additional coding, but easy
- Metadata: reads are fast. Writes are slow+safe

PlasmaFS: Namenodes 4

- DB transactions ≠ PlasmaFS transactions
  - For reading data a PlasmaFS transaction can pick any DB transaction from a set of transactions designated for this purpose → high parallelism
  - Writing to DB occurs first when the PlasmaFS transaction is committed. Writes are serialized.
  - DB accesses are lock-free (MVCC) and never conflict with each other (write serialization)

Plasma FS: Native API 1

- SunRPC protocol
- Ocaml module: Plasma_client
- Example:

```
let c = open_cluster ”clustername” [ ”m567”, 2730 ] esys
let trans = start c
let inode = lookup trans ”/a/filename” false
let () = commit trans
let s = String.create n_req
let (n_act,eof,ii) = read c inode 0L s 0 n_req
```

PlasmaFS: Native API 2

- Plasma_client metadata operations:
  - `create_inode`, `delete_inode`, `get_inodeinfo`, `set_inodeinfo`, `lookup`, `link`, `unlink`, `rename`, `list`
  - `create_file` = `create_inode` + `link`, for regular files or symlinks
  - `mkdir` = `create_inode` + `link`, for directories
- Sequential I/O: `copy_in`, `copy_out`
- Buffered I/O: `read`, `write`, `flush`, `drop`
- Low-level: `get_blocklist`
  - Important for Map/reduce

Time for demo!

PlasmaFS: Native API 3

- Bundle several metadata operations in one trans
  - Isolation guarantees: E.g. Prevent that a concurrent transaction replaces a file behind your back
  - Atomicity: E.g. Do multiple renames at once
- Conflicting accesses:
  - E.g. Two transactions want to create the same file at the same time
  - The late client gets `` `econflict `` error
  - Strategy: abort transaction, wait a bit, and start over
  - One cannot (yet) wait until the conflict is gone

Plasma FS: plasma.opt

- plasma: utility for reading and writing files using sequential I/O

  plasma put <localfile> <plasmafsfile>

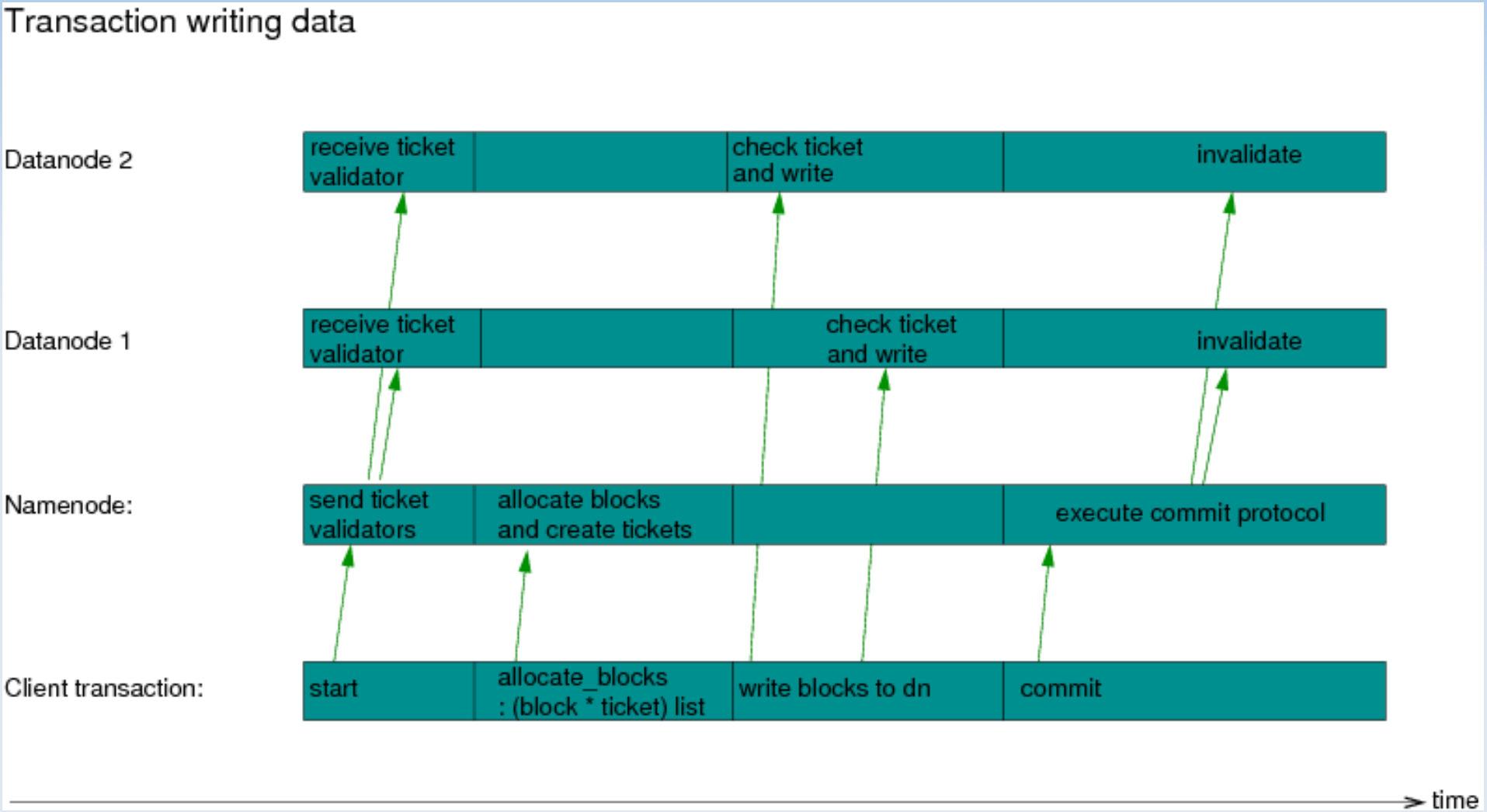
- Also many metadata ops available (ls, rm, mkdir...)

PlasmaFS: Datanode Protocol 1

- Simple protocol: `read_block, write_block`
- Transactional encapsulation:
  - `write_block` only possible when the namenode handed out a ticket permitting writes
  - `read_block`: still free access, but similar is planned
  - Tickets are bound to transactions
  - Tickets use cryptography
  - Reasons: Namenode can control which transactions can write, for access control (*), and for protecting against misbehaving clients

(*) not yet fully implemented

PlasmaFS: Datanode Protocol 2



PlasmaFS: Write Topologies

- Write topologies:
  - How to write the same block to all datanodes storing replicas
  - Star: Client writes directly to all datanodes.
    → Lower latency. This is the default.
  - Chain: Client writes to one datanode first, and requests that this node copies the block to the other datanodes
    → Good when client has bad network connectivity
  - Only `copy_in, copy_out` implement Chain

PlasmaFS: Block replacement

- Client requests that a part of a file is overwritten
- Blocks are never overwritten!
- Instead: Allocate replacement blocks
- Reason 1: Avoid that in any situation some block replicas are overwritten while others are not
- Reason 2: A concurrent transaction might have requested access to the old version. So the old blocks must be retained until all accessing transactions have terminated

PlasmaFS: Blocksize 1

- All blocks have the same size
- Strategy:
  - Disk space is allocated for the blocks at datanode init time (static allocation)
  - It is predictable which blocks are contiguous on disk
  - This allows the implementation of block allocation algorithms to allocate ranges of blocks, and these are likely to be adjacent on disk
  - Good clients try to exploit this by allocating blocks in ranges. Easy for sequential writing. Hard for buffer-backed writes that are possibly random
  - Hopefully no performance loss for medium-sized blocks (compared to large blocks, e.g. 64M)

PlasmaFS: Blocksize 2

- Advantages of avoiding large blocks:
  - Saves disk space
  - Saves RAM. Large blocks also means large buffers (RAM consumption for buffers can be substantial)
  - Better compatibilty with small block software and protocols
    → Linux kernel: page size is 4K
    → Linux NFS client: up to 1M blocksize
    → FUSE: up to 128K blocksize
- Disadvantages of avoiding large blocks:
  - Possibility of fragmentation problems
  - Bigger blockmaps (1 bit/block in DB; more in RAM)

PlasmaFS: NFS support 1

- NFS version 3 is supported by a special daemon working as bridge
- Possible deployments:
  - Central bridges for a whole network
  - Each fs-mounting node has its own bridge, avoiding network traffic between NFS client and bridge
- NFS bridge uses buffered I/O to access files
  - NFS blocksize can differ from PlasmaFS blocksize. The buffer layer is used to ”translate”
  - Buffered I/O often avoids costs for creating transactions. Many NFS read/write accesses need no help from namenodes.

PlasmaFS: NFS support 2

- Blocksize limitation: Linux NFS client restricts blocks on the wire to 1M
  - Other OS: even worse, often only 32K
- Experience so far:
  - Read accesses to metadata: medium speed
  - Write accesses to metadata: slow
  - Reading files: good speed, even when the NFS blocksize is smaller than the PlasmaFS blocksize
  - Writing files: medium speed. Can get very bad when misaligned blocks are written, and the client syncs frequently (because of memory pressure). Writing large files via NFS should be avoided.

PlasmaFS: Further plans

- Add fake access control
- Add real access control with authenticated RPC (Kerberos)
- Rebalancer/defragmenter
- Automatic failover to namenode slave
- Ability of hot-adding namenodes
- Namenode slaves could take over load for managing read-only transactions
- Distributed locks
- More bridges (HTTP, WebDAV, FUSE)

Plasma M/R: Overview

- Data storage: PlasmaFS
- Map/reduce phases
- Planning the tasks
- Execution of jobs

Plasma M/R: Files

- Files are stored in PlasmaFS

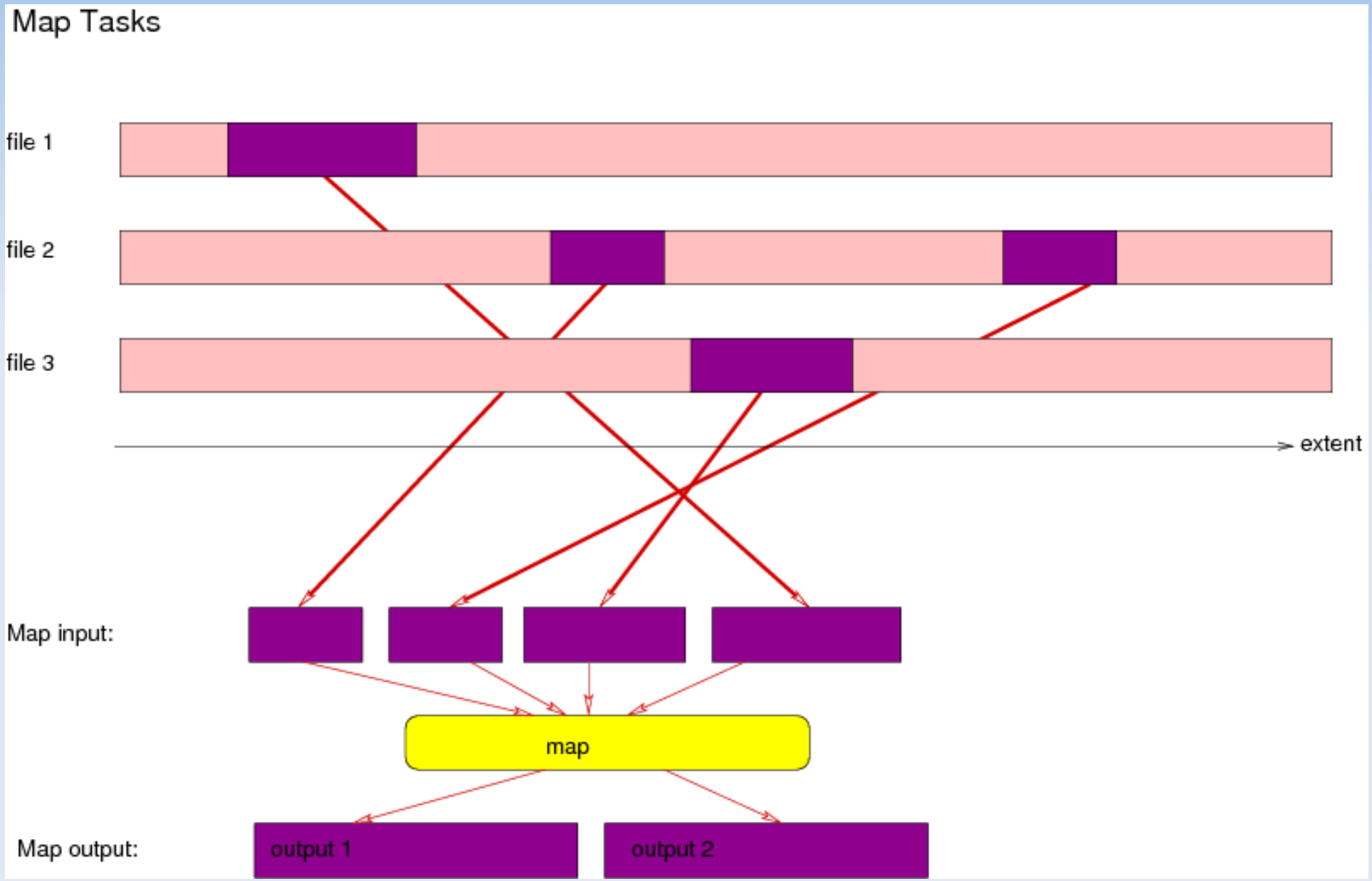
  (this is true even for intermediate files)
- Files are line-structured: Each line is a record
- Files are processed in chunks of *bigblocks*

  Bigblocks are whole multiples of PlasmaFS blocks
- Size of records is limited by size of bigblocks
- Example:
  - PlasmaFS blocksize: 256K
  - Bigblock size: 16M (= 64 blocks)

Plasma M/R: Phases 1

- Map:
  - Before starting Map, the physical locations of the file blocks are determined
  - the Map operation is split into m tasks, where each task maps a number of bigblocks
  - Optimal: the bigblocks of a task are locally available
  - A Map task usually emits several output files. Each file is not longer than the Sort phase can process
  - The output files are also stored in PlasmaFS, with the preference of allocating the data blocks locally

Plasma M/R: Phases 1 (cont)



Plasma M/R: Phases 2

- Sort:
  - A Sort task takes a single output file from Map, sorts it, and writes the result into a second file
  - Because the input files do not exceed a certain maximum in length, it is ensured that the sort can always be done in RAM
  - Sort criterion: First by hash(key), and only if two hashes are identical, compare the keys
  - The Sort output file is again written to PlasmaFS, also with preference for local storage

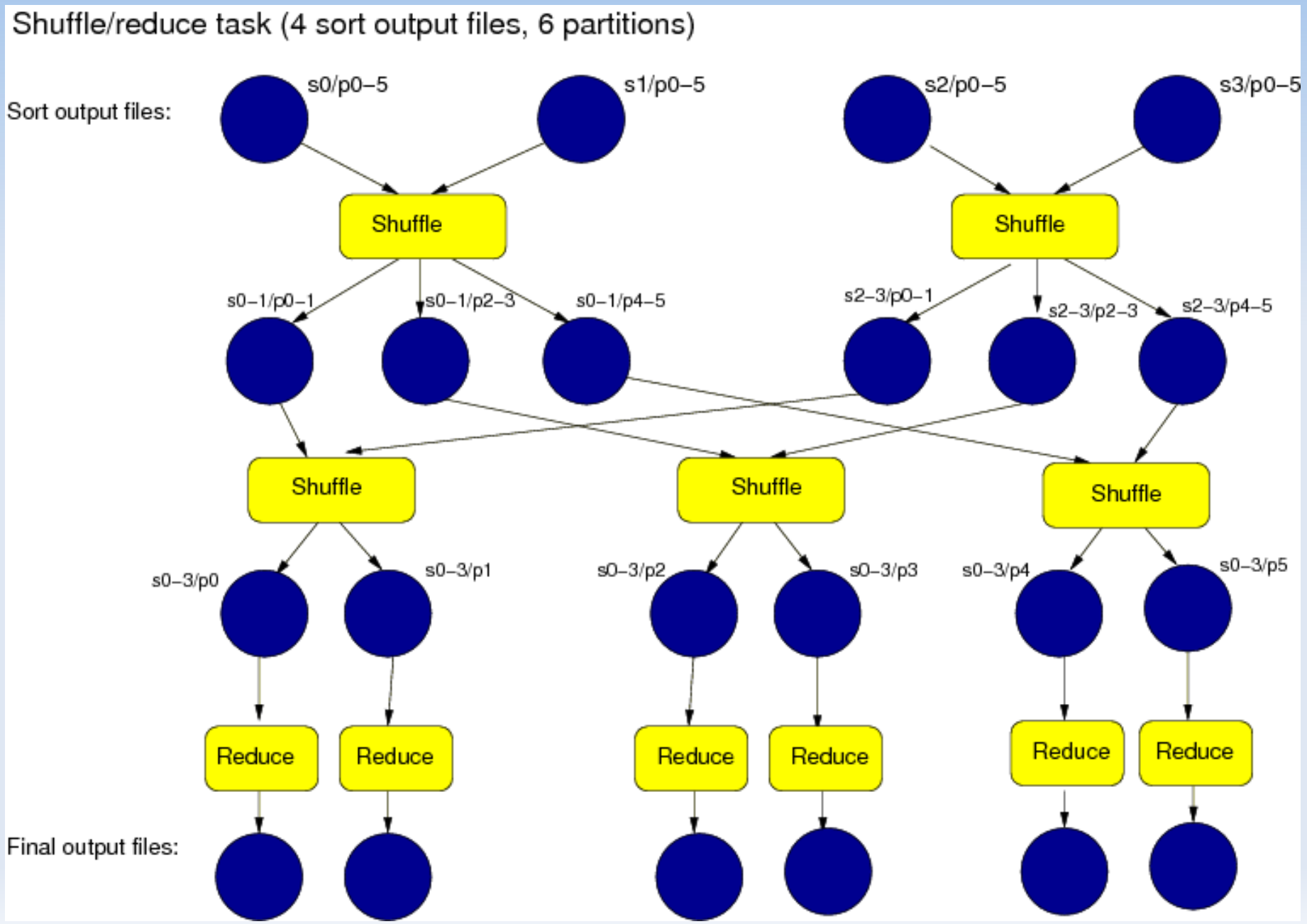
Plasma M/R: Phases 3

- Shuffle:
  - A Shuffle task takes p sorted input files, merges them, splits the records by partition ranges, and writes q sorted output files.
  - p and q can be freely configured, e.g. p=q=4
  - Before the data is completely merged, and completely split into partitions, a number of Shuffle tasks must run in sequence (quite a lot...)
  - Finally, all Shuffle tasks together produce one output file per partition, so that each output file contains the records of all Map outputs falling into the partition

Plasma M/R: Phases 4

- Reduce:
  - Reduce is not a task type in Plasma M/R, but simply a post-operation of the final Shuffle tasks that write the output files

Plasma M/R: Shuffle+Reduce



Plasma M/R: Planning

- Planning means to analyze the input files, and to produce a scheme which tasks need to be run in which order, on which nodes, and with which data
- The planning scheme is dynamic:
  - It is not a-priori known how many files are emitted by the Map task
  - Whenever a Map task finishes, the plan is extended, and more Sort and Shuffle tasks can be added

Plasma M/R: Execution 1

- User builds an mr.opt executable (by linking the Plasma libraries with custom map and reduce functions)
- Config file: mr.conf (s/.opt/.conf/)
- Start the task server on all task nodes:
  - `./mr.opt start_task_servers  # or stop_task_servers`
- Start the job:
  - `./mr.opt exec_job <parameters>`
  - Job is controlled by local process (no job tracking)
  - Tasks are invoked on the task servers

Plasma M/R: Execution 2

- Jobs can be killed by SIGINT, sent to the job-controlling process (press CTRL-C)
- By default, intermediate files are deleted asap. Prevent this with -keep-temp-files
- There is the possibility of copying helper files to all task nodes at job start time (config)
- Log files are moved to PlasmaFS

Time for demo!

Plasma M/R: Example 1

- Count word frequencies (mr_wordfreq.ml):
- Map: splits each line into words
  Reduce: Counts identical words + emits number

```
let job : Mapred_def.mapred_job =
object
  …
end

let () =
  Mapred_main.main job
```

Plasma M/R: Example 2

```
(* map: split each line of the input into words, and output
   each word on a separate line *)

method map me jc ti rd wr =
  try
    while true do
      let r = rd#input_record() in
      let words = Pcre.split r in
      List.iter (fun word -> wr # output_record word) words
    done
  with End_of_file ->
    wr # flush()

(* The whole line is the key: *)

method extract_key me jc line =
  line

(* Just create some partitions *)

method partition_of_key me jc key =
  (Hashtbl.hash key) mod jc#partitions
```

Plasma M/R: Example 3

```
(* Count adjacent words, and emit frequency: *)

method reduce me jc ti rd wr =
  let last = ref "" in
  let freq = ref 0  in
  try
    while true do
      let r = rd#input_record() in
      if r = !last then
         incr freq
      else (
        if !freq > 0 then
          wr # output_record
            (!last ^ " " ^ string_of_int !freq);
        last := r;
        freq := 1
      );
    done
  with End_of_file ->
    if !freq > 0 then
     wr # output_record (!last ^ " " ^ string_of_int !freq);
    wr # flush()
```

Plasma M/R: Configuration

- Task server config: which nodes exist, capacity limits etc.
- Job config: Interpreted at job_exec time

```
mapredjob {    (* minimal config *)
  input_dir = "/input";
  output_dir = "/output";
  work_dir = "/work";
  log_dir = "/log";
  partitions = 10;
}
```

Plasma M/R: Streaming

- Use mr_streaming
- Special job configs:
  - task_files = ”file1 file2 ...”: files to copy to task nodes
  - map_exec = ”./command ...”
  - reduce_exec = ”./command ...”
  - extract_mode = ”key” (default)
extract_mode = ”key_tab_value”
extract_mode = ”key_tab_partition_tab_value”
(requires that map outputs this)
- No counters or progress messages implemented

Plasma M/R: Further plans

- More optimizations (esp. Sort)
- Factor task server framework out into a separate library; add queueing; add distributed load control; make it operating-friendly
- This & that (your opinion is needed)