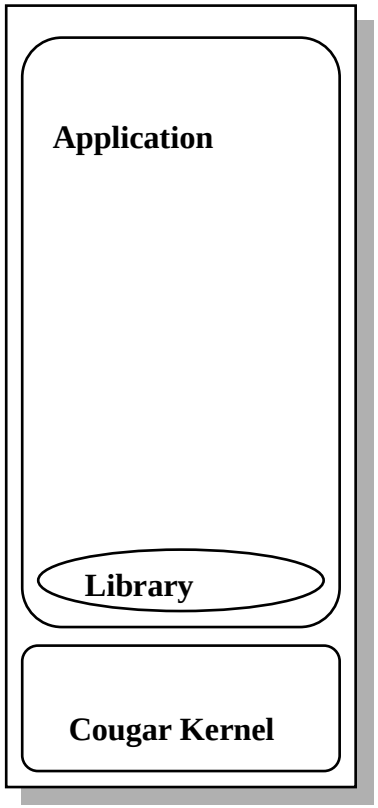
I/O Software Architecture and API Interfaces

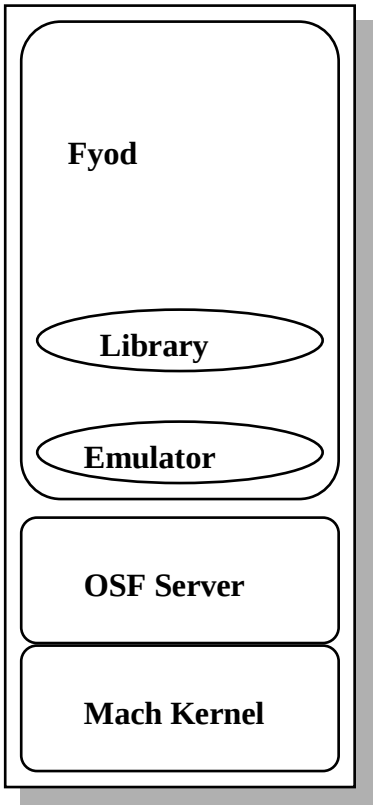
David Robboy

Part 1: I/O Software Architecture

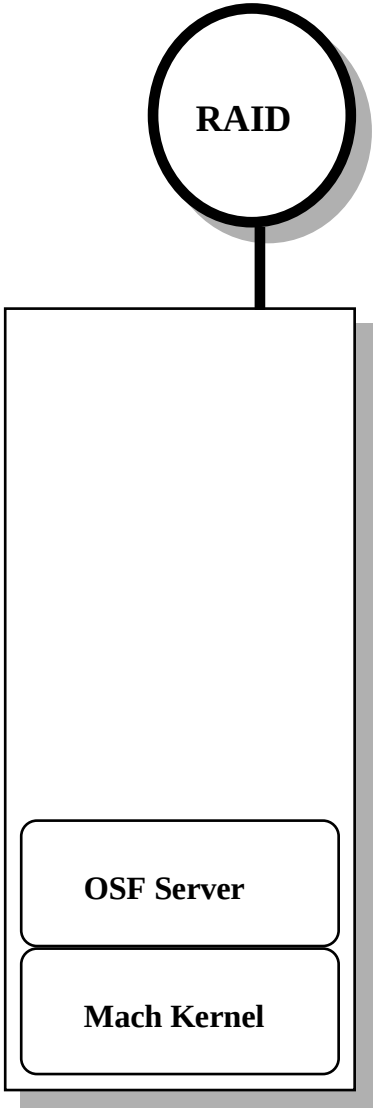
Software Architecture



Compute Node

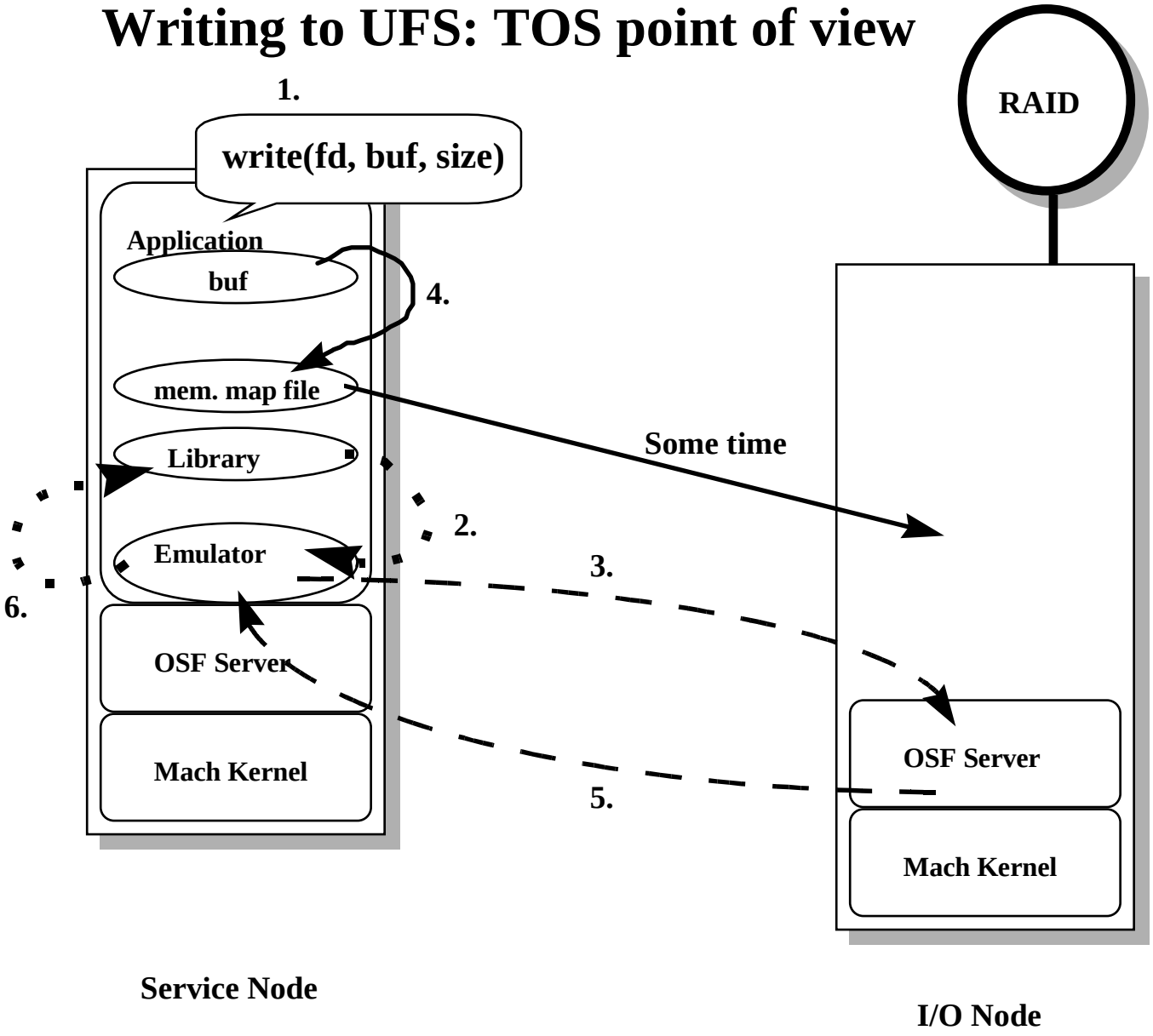


Service Node

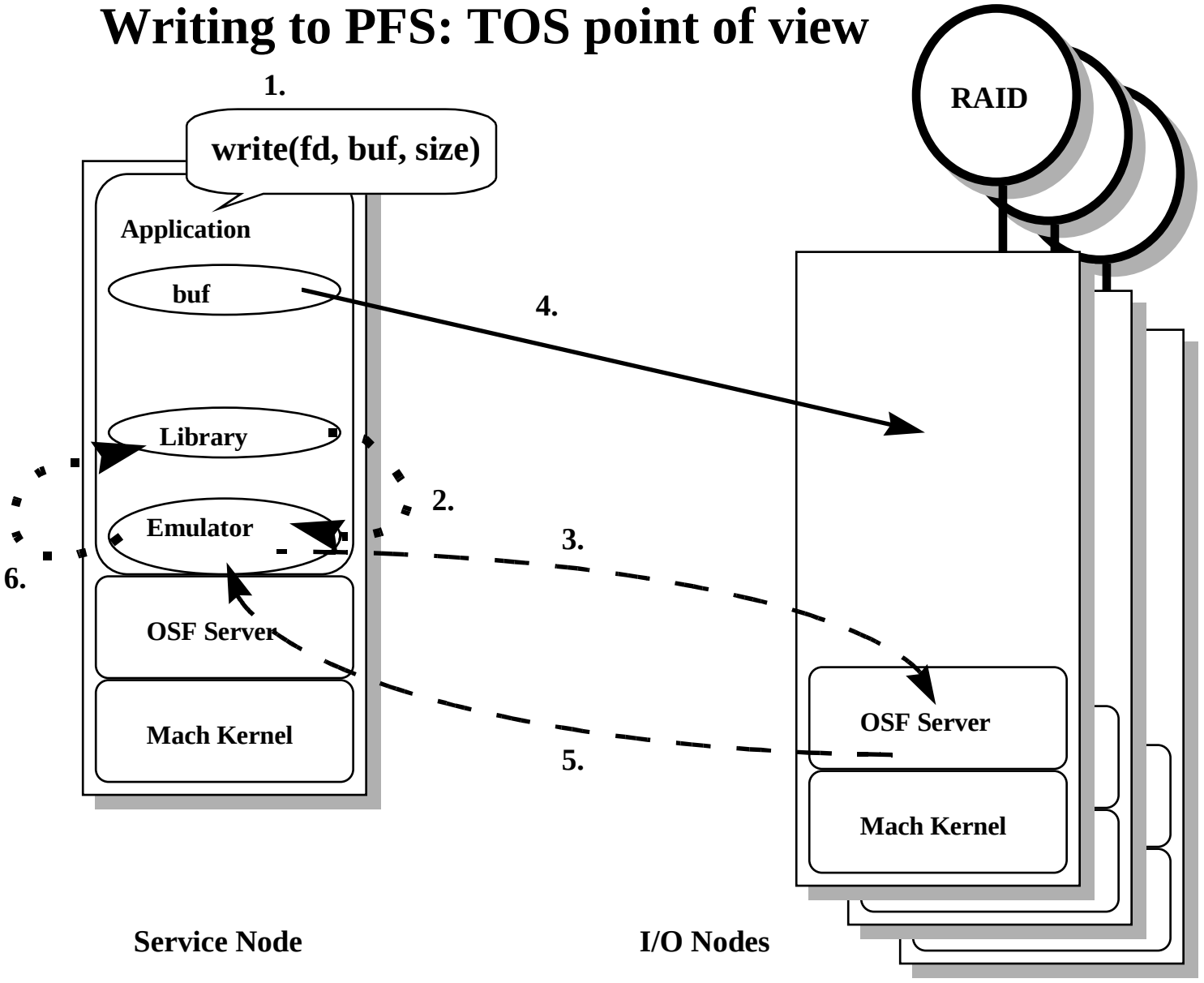


I/O Node

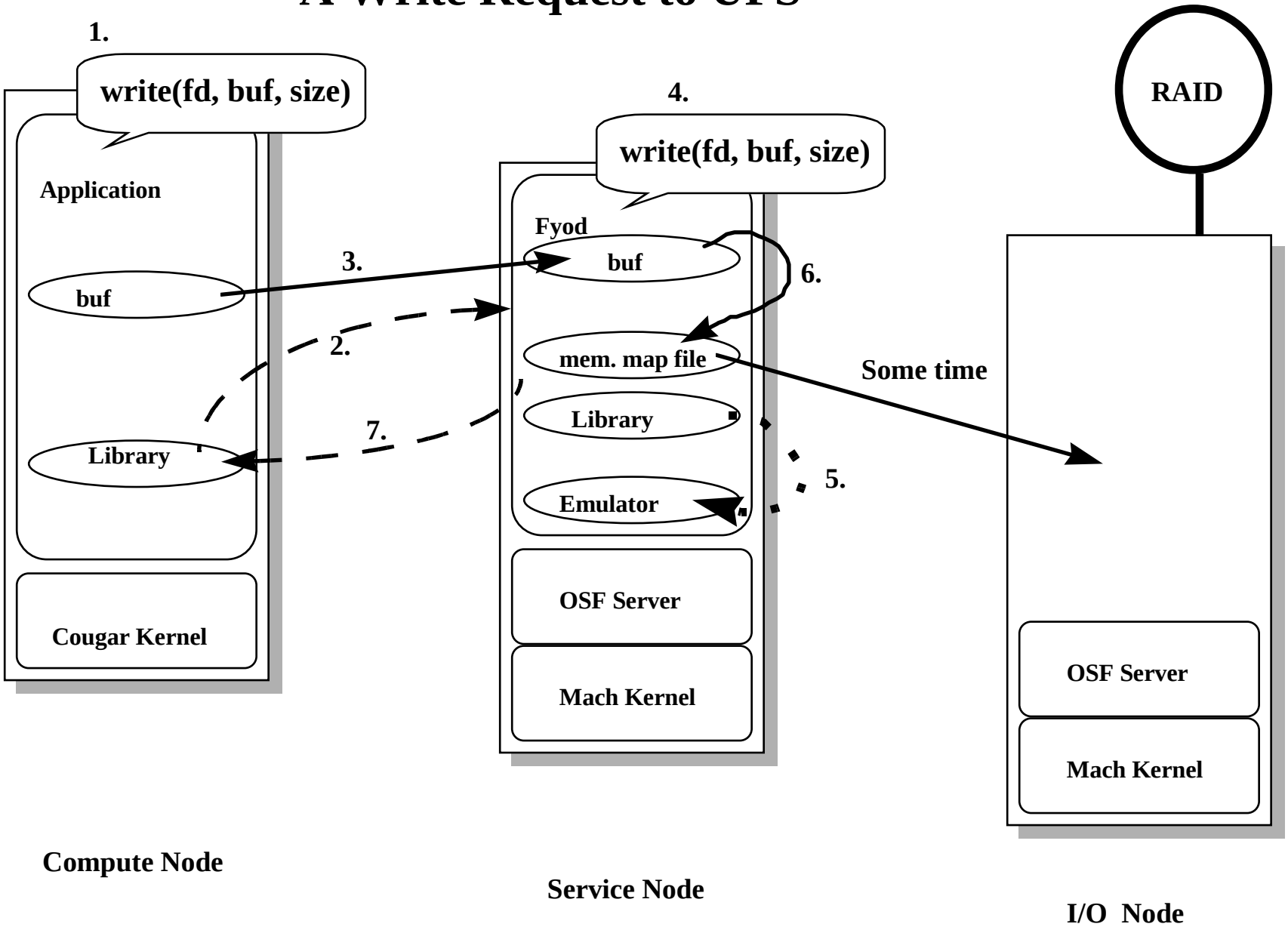
Writing to UFS: TOS point of view



Writing to PFS: TOS point of view



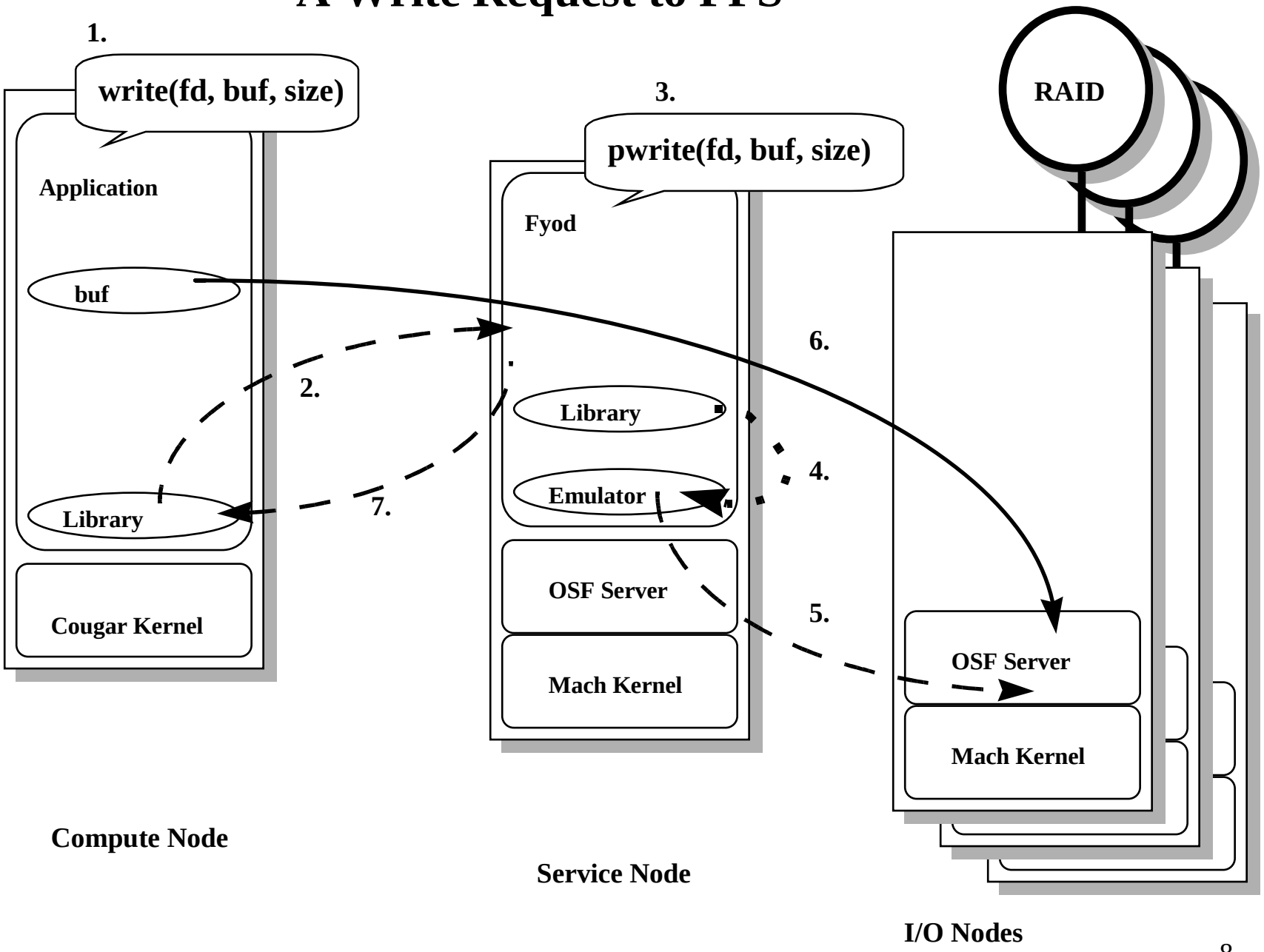
A Write Request to UFS



Notes on UFS Write

- Fyod allocates a buffer
- Fyod pulls data from the compute node to its buffer
- Then fyod does an ordinary write()
- Fyod pulls and writes data in 128KB chunks
- Reading is the same, in the opposite direction

A Write Request to PFS



Notes on PFS Write

- Fyod makes a special call: pwrite()
  - Just once per user call
- The emulator dispatches requests to one or more OSF servers
  - 4MB per stripe file at a time
- The server pulls data from the compute node
- The server writes data directly to the RAID
  - No O. S. buffering
- Read is the same, in the opposite direction

Summary of PFS

- Parallelism
  - For a single compute node, large operations
  - Multiple compute nodes in parallel
- Shorter data path => higher bandwidth
- No data buffering
- No size limit of 2GB per file
- Optimal only for large aligned I/O requests

Summary of UFS

- Optimal for small or unaligned requests
- System buffering via memory mapped files
  - This helps, if data is re-used
- Size limit of 2GB per file
- Problem with many nodes, many files, and multiple fyods:
  - Files are buffered on the fyod's service node
  - Can cause thrashing due to paging

UFS Memory-Mapped Files

- An O. S. buffer cache for UFS file data
- Behaves externally like a static buffer pool
- The buffer persists after a process exits
- Coherent across service nodes
  - Single system image
  - Behaves like distributed virtual memory

Part 2: User API Interfaces

Unix System Calls

- open, read, write, ...
- Available in C and C++
- Portable
- Non-scalable because they're synchronous
  - fyod is blocked until each call is done
  - Through each fyod, the requests are sequential

C Standard I/O Library

- Uses stdio.h
- fopen, fread, fwrite, ...
- Available in C and C++
- At least as portable as system calls
  - Specified by the C language
- Data is buffered in application space
  - “stream buffer”

Implications of stdio buffering

- Better than system calls for small or unaligned requests - due to stream buffering
- Bandwidth is limited by copying data to the stream buffer
  - For large requests, fwrite bypasses the buffer
  - fread does not bypass the buffer
- Use setbuffer to create a PFS block-sized buffer for optimal performance to PFS files

Scalability of stdio

- fwrite can be equivalent to cwrite
  - fwrite calls cwrite
  - Use a PFS block-sized stream buffer
  - Do buffer-sized fwrites to bypass the buffer
- fread is not scalable
  - It calls read, not cread
  - It does not bypass the stream buffer

C++ fstream class

- Uses iostream.h and fstream.h
- Portable
- Dependent on C++ implementation
- Probably roughly equivalent to stdio
- It’s not obvious how to control the buffer size

FORTRAN Runtime Library

- open, read, write, ...
  - These are distinct from Unix system calls with similar names
  - E. g., Unix uses file descriptors, not unit numbers
- These functions make stdio calls
  - Non-scalable
  - You can’t call setbuffer to control the buffer size
- Async I/O calls also available in FORTRAN

Asynchronous I/O calls

- cread, cwrite, iread, ...
- Available for C, C++, and FORTRAN
- Not portable
- Parallelism at the fyod level
  - If multiple compute nodes do a *iwrite* concurrently, an fyod can handle them concurrently
  - Requests are dispatched to multiple I/O nodes in parallel

Asynchronous I/O to UFS

- UFS buffers data on the service node
  - Buffering = memory mapped files = demand paging
  - Multiple nodes doing async. operations to UFS can thrash the service node with paging
- To avoid this, use PFS when doing async. I/O

MUNIX mode

- In MUNIX mode, reads and writes to a file are atomic
  - Standard Unix semantics
  - The current default
- Implemented by serializing I/O per file, per fyod
- The fyod is the Unix process.
  - One fyod at a time can read/write a file

MASYNC Mode

- MASYNC allows parallel access to a file
  - Two processes can partially overwrite each other
  - A process can read a record that has been partially updated by another process
- Multiple fyods can read/write a file in parallel
- For a PFS file, allows multiple fyods to access multiple stripes in parallel

How can MASYNC mode help performance?

- If there is more than one fyod per application...
- If multiple compute nodes share a file...
- Then MASYNC mode allows concurrent access by multiple fyods.

The MASYNC Bug

- If a file written in MASYNC mode is sparse and has a hole at the end of a PFS stripe, then the file size is wrong.
- Data can be lost.
- I’ll explain this with a whiteboard.
- This bug will be fixed, but not in R2.3.

Workarounds to the MASYNC Bug

- Avoid sparse files, fill in the gaps
- Write something to the end of each PFS stripe
- Don’t share files, use a file per node
- Use MUNIX mode