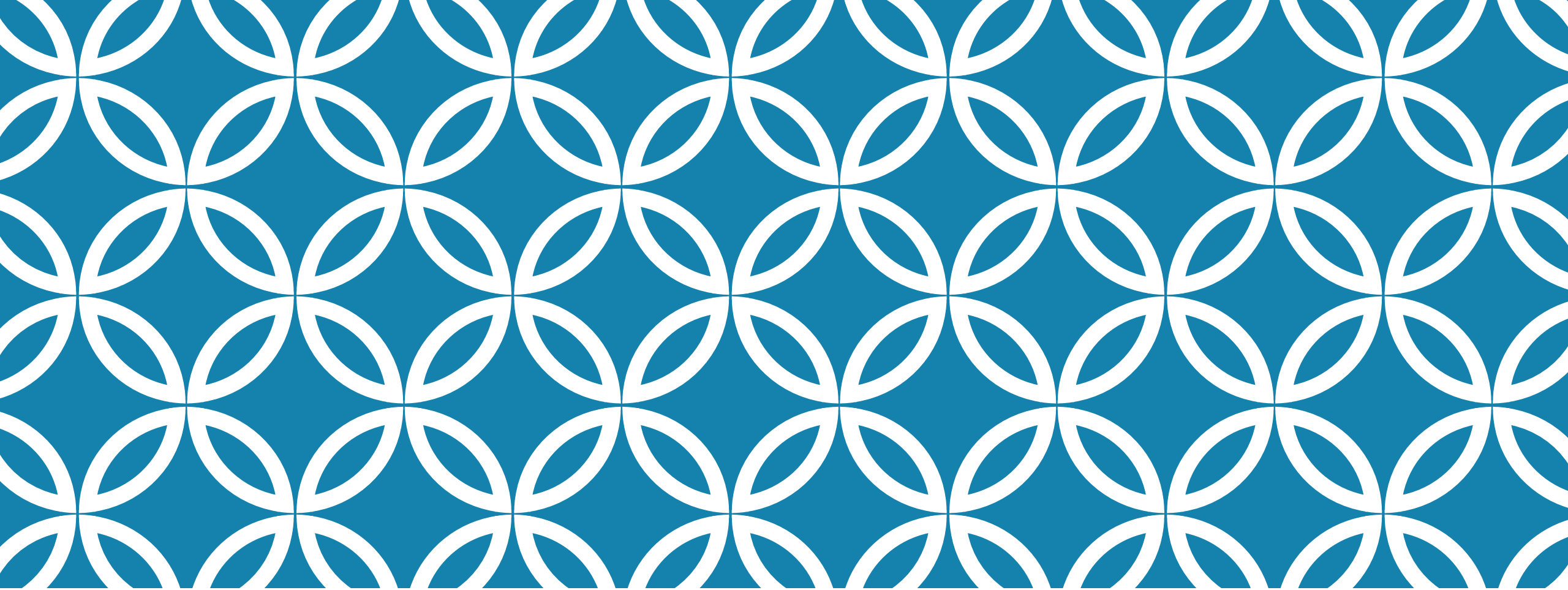


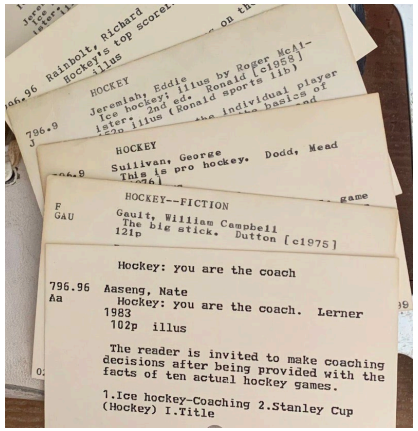
CS 5412/LECTURE 20: MAKING THE CLOUD FRIENDLIER FOR OBJECT-ORIENTED COMPUTING

Ken Birman

Fall, 2022

TODAY'S MAIN TOPIC

The cloud was built mostly from Linux systems, yet Linux was created to support interactive computing applications in offices and hospitals, databases (big files, but record-oriented), text editing.



Records are fixed size, format. Like an old-style library index with fixed fields and length limits. In the 1970's, a data file often would be a vector of these fixed-sized records.

This explains the POSIX **lseek** operation: "read/update the middle of a file"

TODAY'S MAIN TOPIC

Over time, file abstractions evolved. Today we often memory-map files, using the **mmap** file system operation. **lseek** is rarely used and very few programs ever open a file, modify something in the middle, then close it.

A modern file system might be used to hold objects.

- Serialize the object, then write the whole byte vector
- For updates, replace one version with a new version
- But in fact some applications do need "append" as an option, in order to add a row to a comma-separated value file (a CSV file is a common way to represent a spreadsheet or a database)

QUESTIONS THIS RAISES?

If objects, tuples with named fields, and relations with attributes are equivalent in some ways, which to pick for efficient code development?

If Linux is out of tune with object-oriented computing, what needs to be changed to make the match closer?

Should that old **lseek** option of reading or updating the middle of a file even be supported?

OBJECTS IN MULTI-LINGUAL APPLICATIONS

There are hundreds of programming languages, and libraries– but often coded in specialized languages different from what the caller is using.

This creates yet another question: How to share objects between code written in different languages.

- They could be separate programs, entirely coded in different languages
- But they could also be one program using libraries in flexible ways

TODAY'S TWO SUB-TOPICS

First half of the lecture: The Ceph object oriented file system. Looks like a normal POSIX file system, but optimized for object-oriented uses.

Second half: CORBA, and the costs of object-sharing in a complex setting (an actual air traffic control system Ken worked on many years ago)

OBJECT-ORIENTED FILE SYSTEMS

IF PEOPLE STORE OBJECTS AS FILES, WHAT GOES WRONG?

With one file per object, we quickly have a LOT of files. Linux file systems aren't so great for this (like a directory with a billion objects in it).

Objects tend to be very small or very large. The Linux file system is optimized for Linux file size distributions and lifetime distributions.

The POSIX API is designed for record-at-a-time file access. But with objects the application more often wants to read/write the whole file

CEPH PROJECT

Created by Sage Weihl, a PhD student at U.C. Santa Cruz

Later became a company and then was acquired into Red Hat Linux

Now the "InkStack" portion of Linux offers Ceph plus various tools to leverage it, and Ceph is starting to replace HDFS worldwide.

Ceph is similar in some ways to any standard cloud file system, but was created separately. Many big data systems are migrating to Ceph.

THREE PERSPECTIVES

First is the standard POSIX file system API. You can use Ceph in any situation where you might use GFS, HDFS, NFS, etc.

Second is the Ceph MetaData layer. This is a subsystem with its own API that manages objects… but doesn't store data

Third is the RADOS object storage layer. It holds the data but doesn't know about the folders (directories) in which data is organized.

CENTRAL CONCEPTS

Think of POSIX as a library: It has a standard set of file system operations but the storage system itself doesn't need to use the identical API

In Ceph, both the file system name space ("meta data store") and data plane ("object store") ultimately map to key-value infrastructures!

- POSIX is like an algorithm that "uses" key-value storage to mimic an old-style Linux file system
- Used in the recommended ways, Ceph is just a very thin layer mapping between these two abstractions

OBJECT STRUCTURES ARE "INVISIBLE" TO CEPH

The objects stored in the system are defined and "owned" by applications, not by Ceph. Ceph itself doesn't *need* the definitions. It cares about *size*

The application defines the actual layout of each object, and the logic to serialize objects to byte vectors, and later deserialize and construct objects

CEPH: A SCALABLE, HIGH-PERFORMANCE DISTRIBUTED FILE SYSTEM

Original slide set from OSDI 2006

Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrel D. E. Long

CONTENTS

Goals

System Overview

Client Operation

Dynamically Distributed Metadata

Distributed Object Storage

Performance

GOALS

Scalability

- Storage capacity, throughput, client performance. Emphasis on HPC.

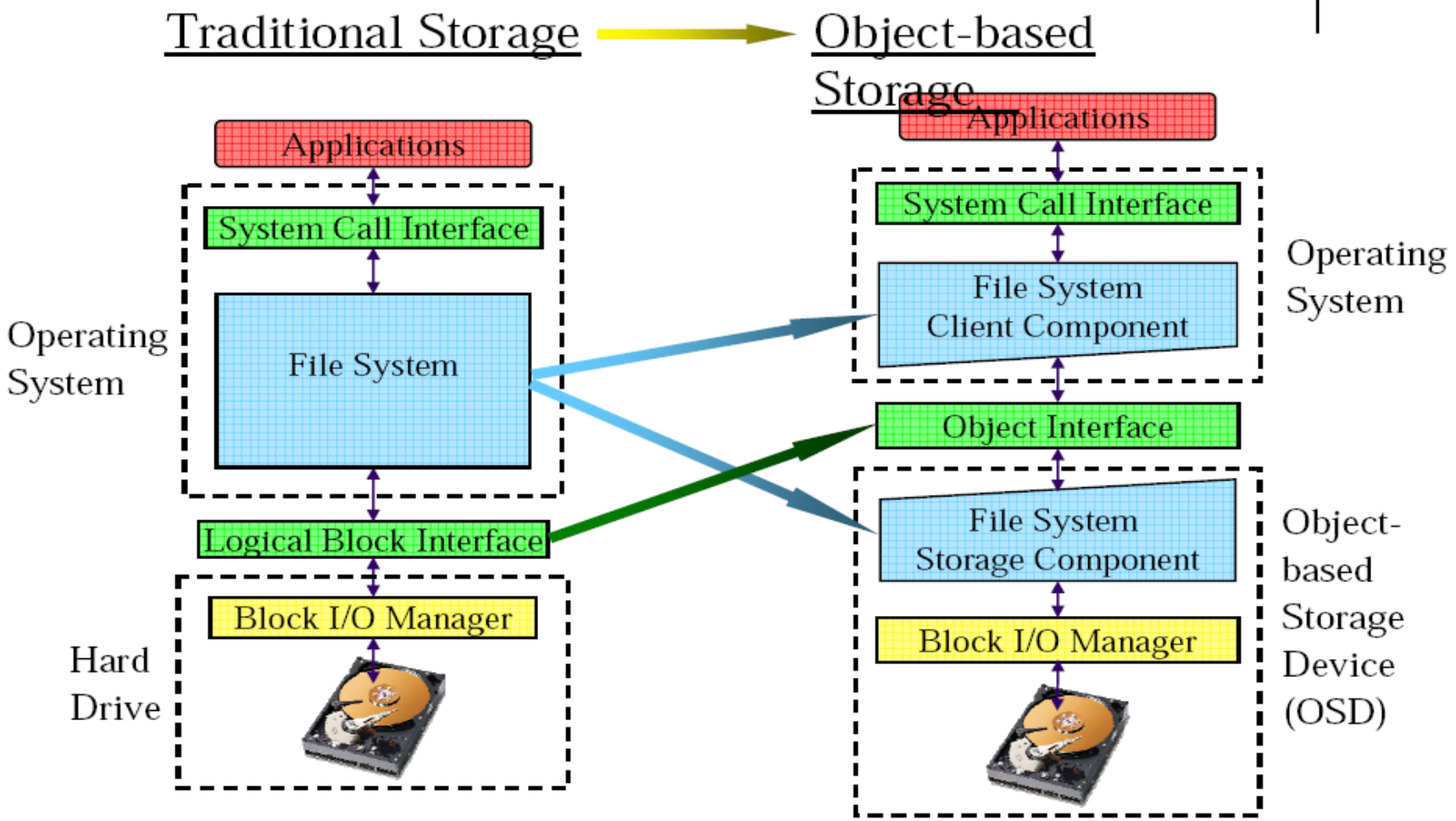
Reliability

- "…failures are the norm rather than the exception…"

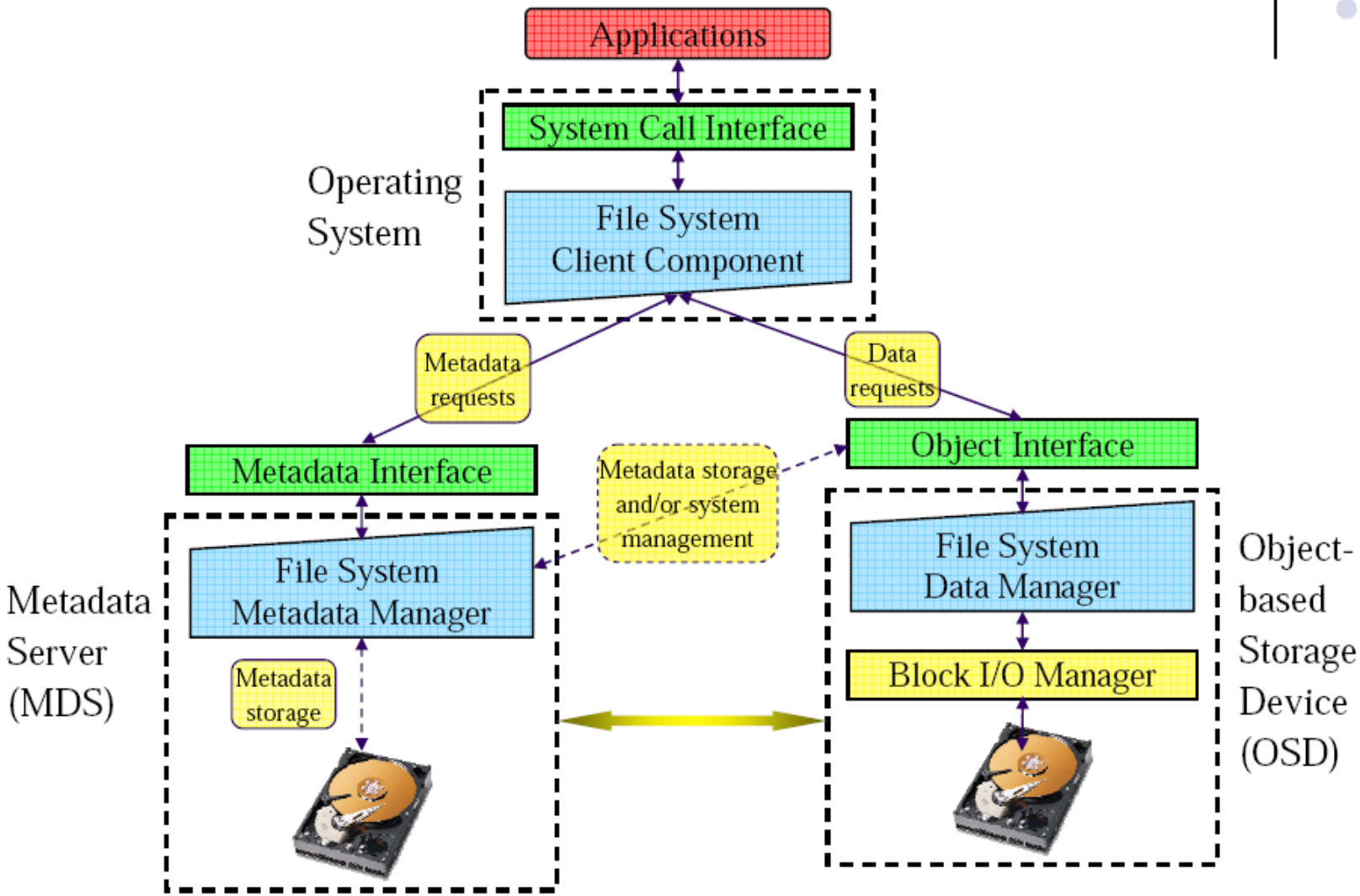
Performance

- Dynamic workloads

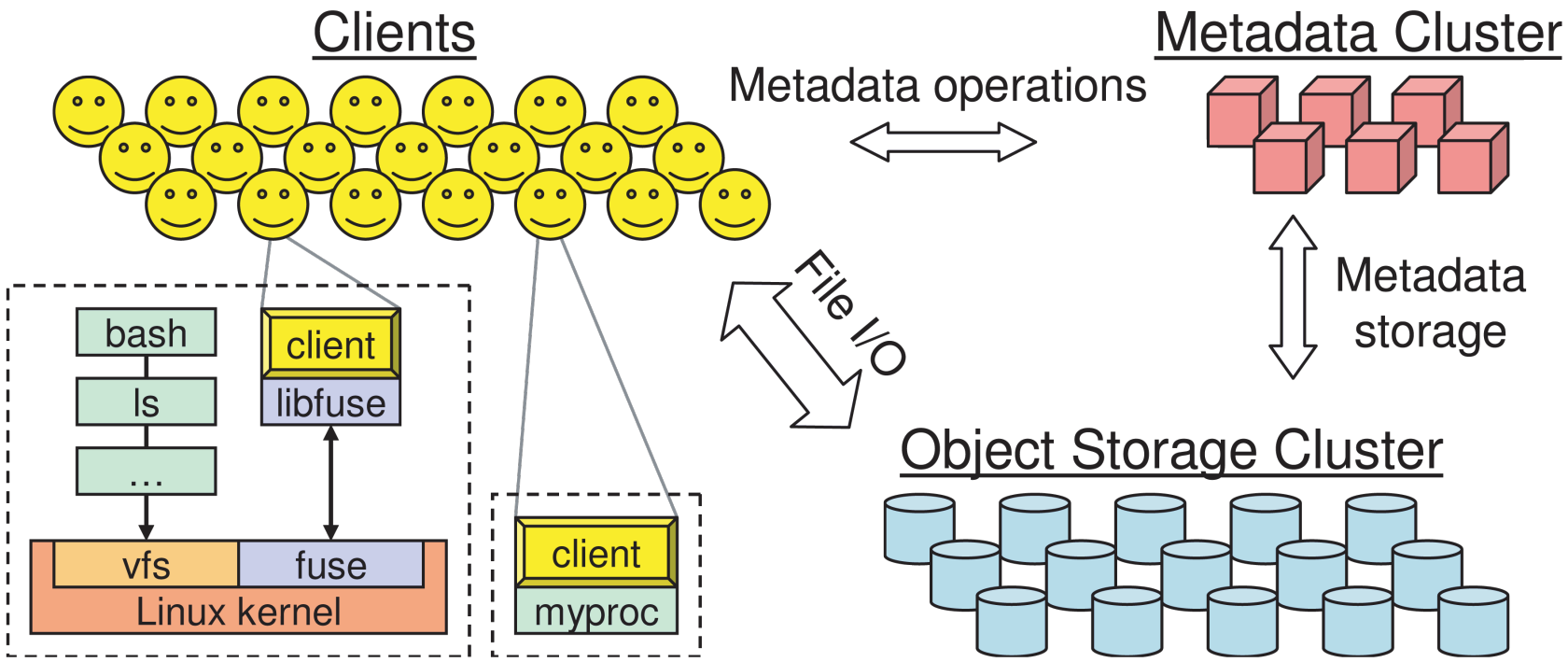
First Key Idea: Object-based Storage



Second Key Idea: Decoupled Data and Metadata



SYSTEM OVERVIEW



KEY FEATURES

Decoupled data and metadata

- CRUSH
  - Files striped onto predictably named objects
  - CRUSH maps objects to storage devices

Dynamic Distributed Metadata Management

- Dynamic subtree partitioning
- Distributes metadata amongst MDSs

Object-based storage

- OSDs handle migration, replication, failure detection and recovery

CLIENT OPERATION

Ceph interface

- Nearly POSIX
- Decoupled data and metadata operation

User space implementation

- FUSE or directly linked

FUSE is a software allowing to implement a file system in a user space

CLIENT ACCESS EXAMPLE

Client sends open request to MDS

MDS returns capability, file inode, file size and stripe information

Client read/write directly from/to OSDs

MDS manages the capability

Client sends close request, relinquishes capability, provides details to MDS

SYNCHRONIZATION

Adheres to POSIX

Includes HPC oriented extensions

- Consistency / correctness by default
- Optionally relax constraints via extensions
- Extensions for both data and metadata

Synchronous I/O used with multiple writers or mix of readers and writers

DISTRIBUTED METADATA

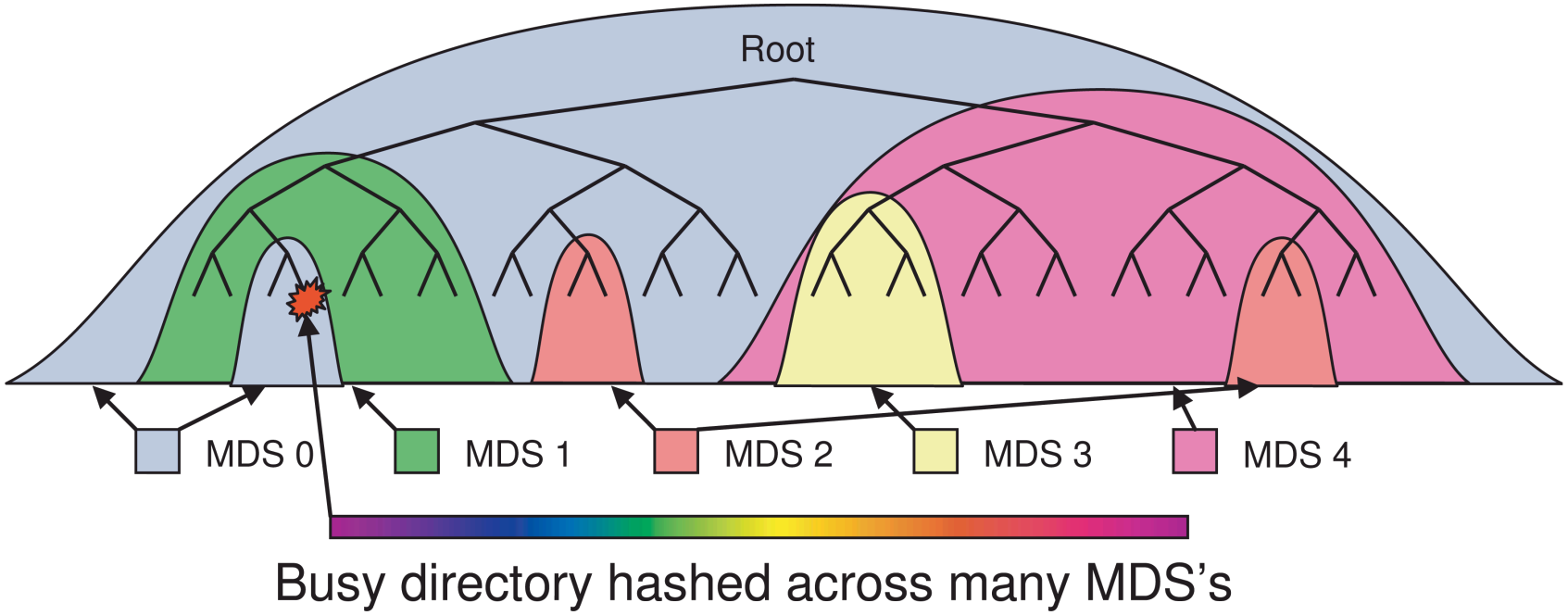
"Metadata operations often make up as much as half of file system workloads…"

MDSs use journaling

- Repetitive metadata updates handled in memory
- Optimizes on-disk layout for read access

Adaptively distributes cached metadata across a set of nodes

DYNAMIC SUBTREE PARTITIONING



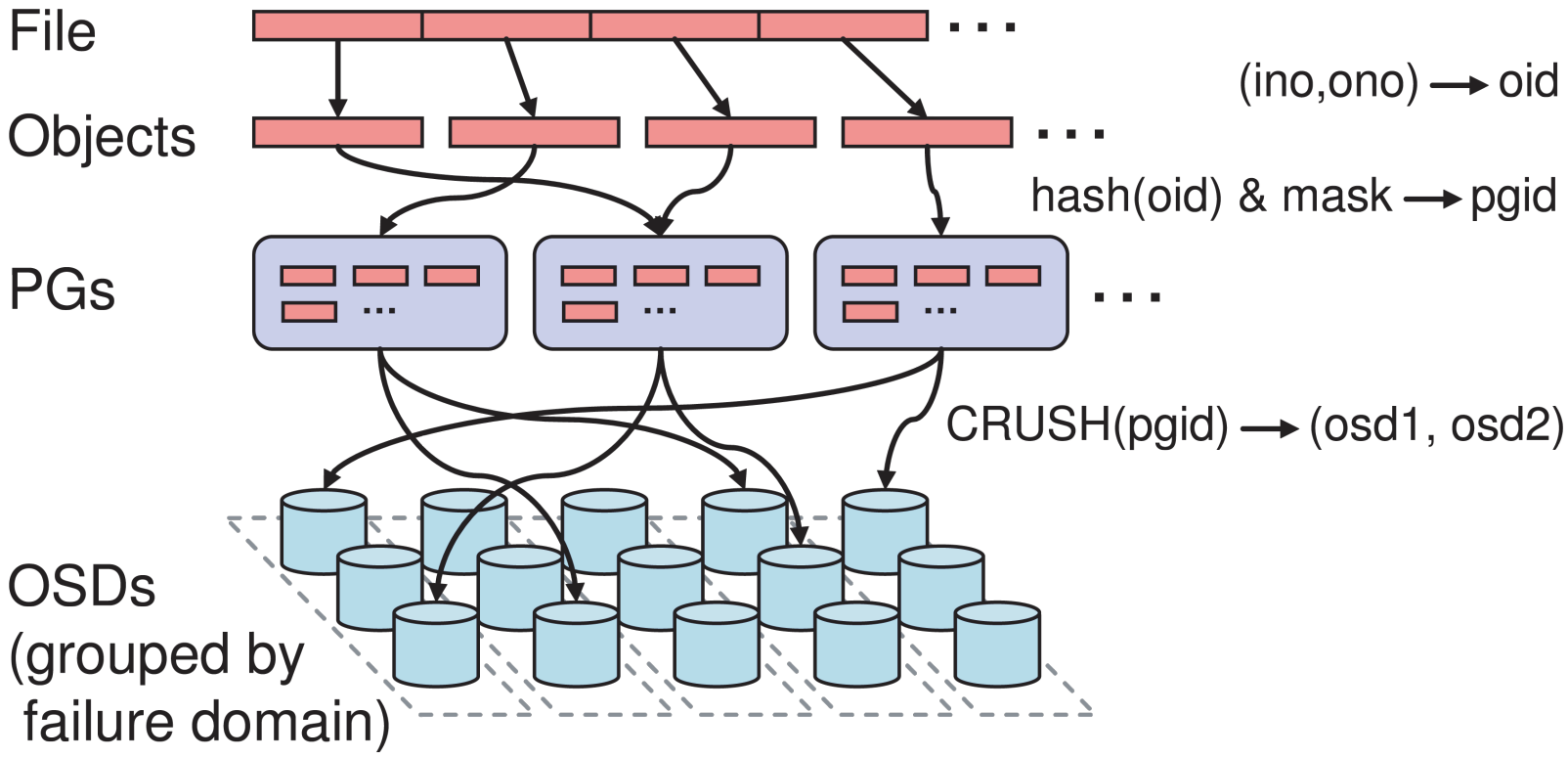
DISTRIBUTED OBJECT STORAGE

Large files are split into a set of objects

Objects are members of placement groups

Placement groups are distributed across OSDs.

DISTRIBUTED OBJECT STORAGE



CRUSH: A SPECIALIZED KEY HASHING FUNCTION

CRUSH(x): (osdn1, osdn2, osdn3)

- Inputs
  - x is the placement group
  - Hierarchical cluster map
  - Placement rules
- Outputs a list of OSDs

Advantages

- Anyone can calculate object location
- Cluster map infrequently updated

DATA DISTRIBUTION

(not a part of the original PowerPoint presentation)

Files are striped into many objects

- (ino, ono) → an object id (oid)

Ceph maps objects into placement groups (PGs).

- hash(oid) & mask → a placement group id (pgid)

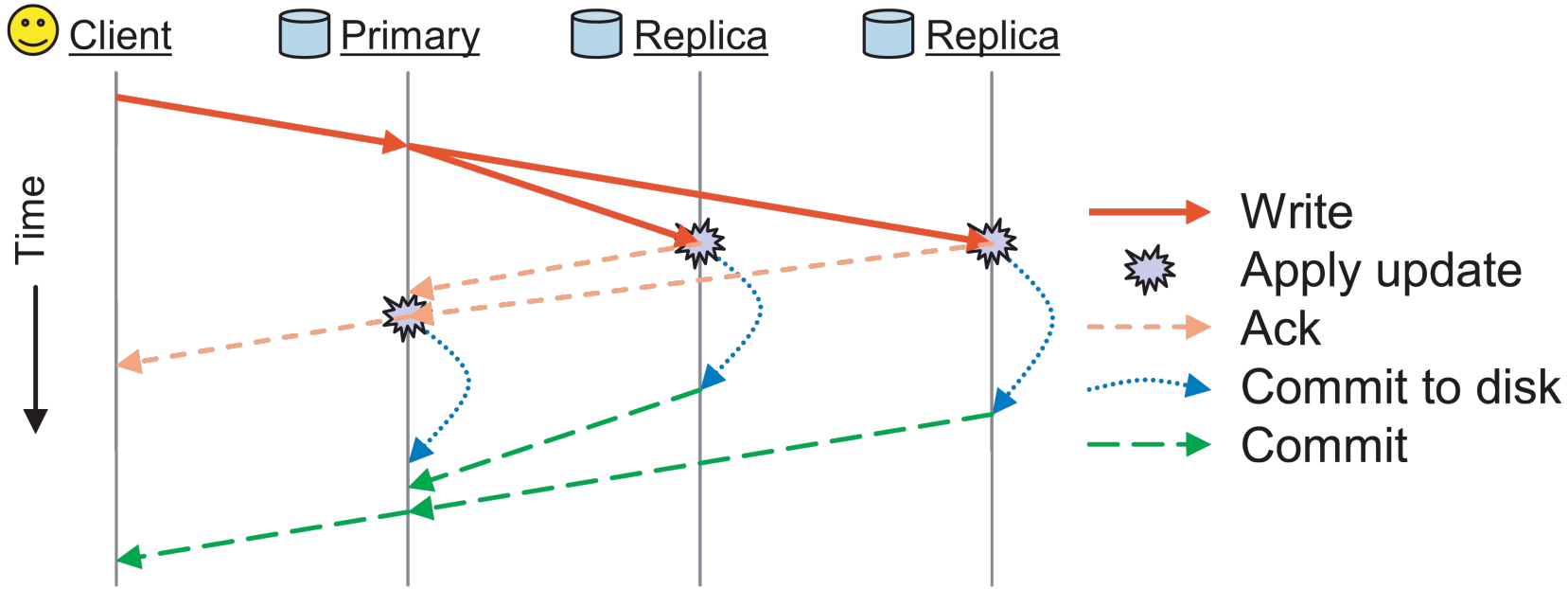
CRUSH assigns placement groups to OSDs, using what seems to be a “shard”

- CRUSH(pgid)→ a replication group, (osd1, osd2)

REPLICATION: RELIABLE BUT NOT PAXOS OR CHAIN REPLICATION

Objects are replicated on OSDs within same PG

- Client is oblivious to replication



FAILURE DETECTION AND RECOVERY

Down and Out

Monitors check for intermittent problems

New or recovered OSDs peer with other OSDs within PG

ACRONYMS USED IN PERFORMANCE SLIDES

CRUSH: Controlled Replication Under Scalable Hashing

EBOFS: Extent and B-tree based Object File System

HPC: High Performance Computing

MDS: MetaData server

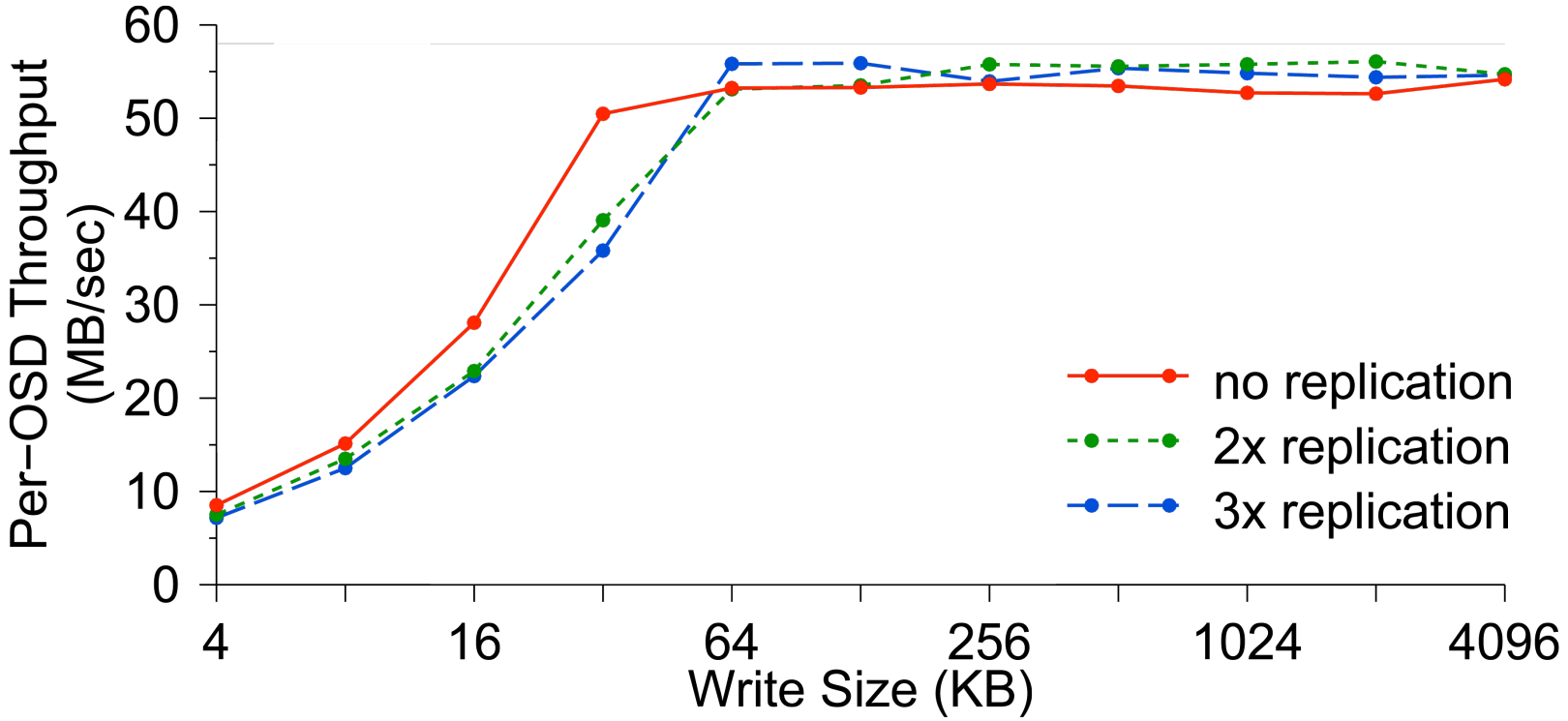
OSD: Object Storage Device

PG: Placement Group

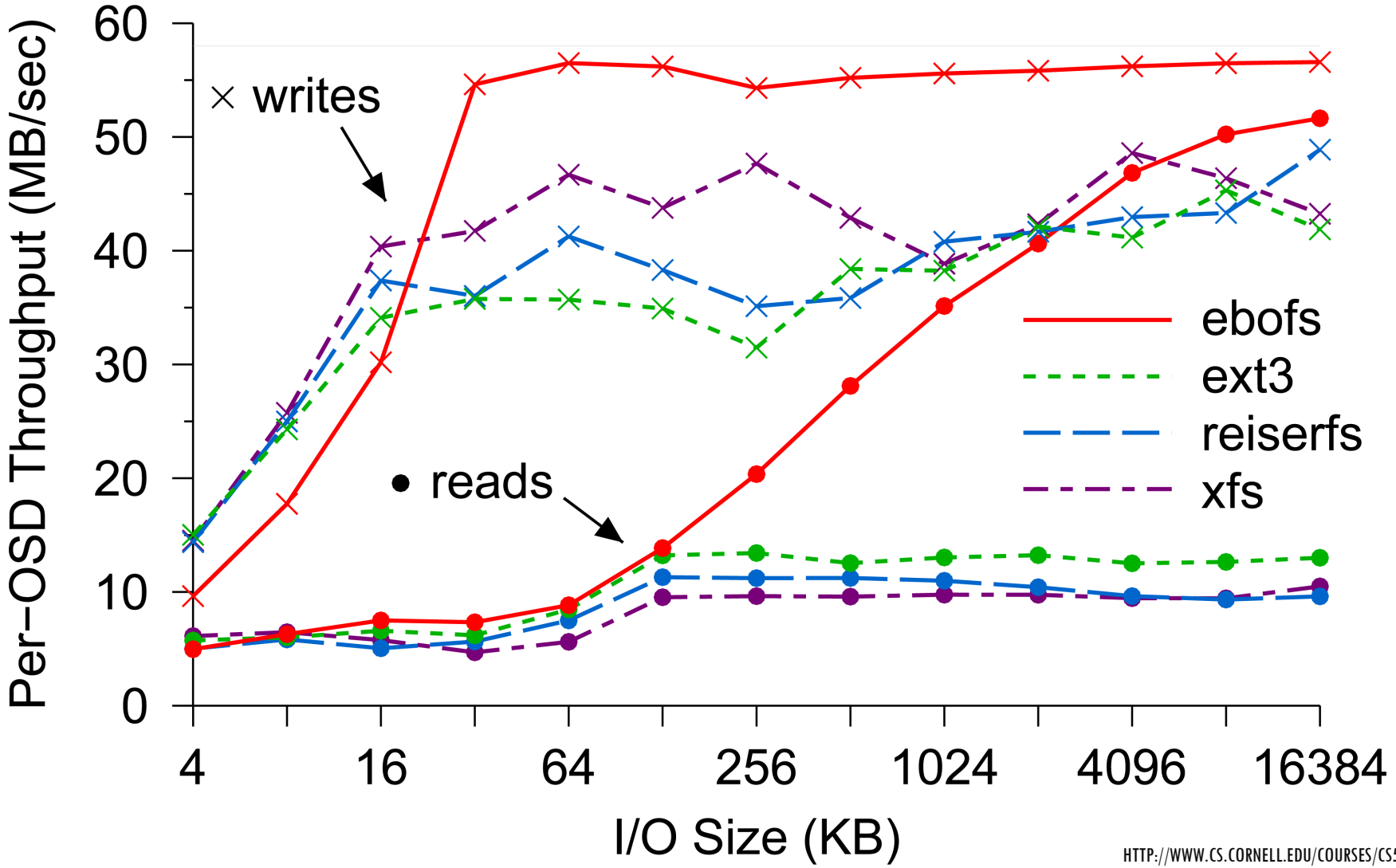
POSIX: Portable Operating System Interface for uniX

RADOS: Reliable Autonomic Distributed Object Store

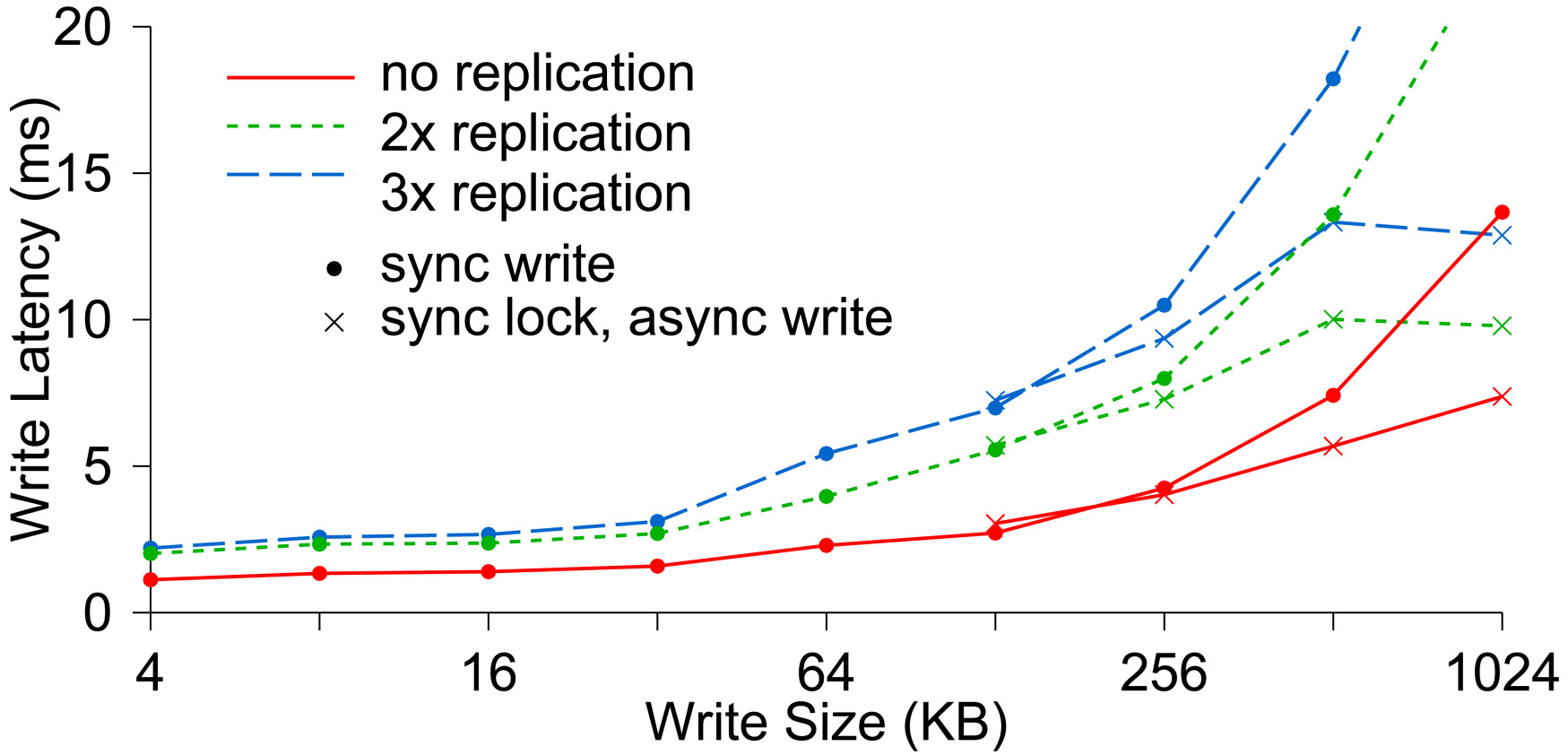
PER-OSD WRITE PERFORMANCE



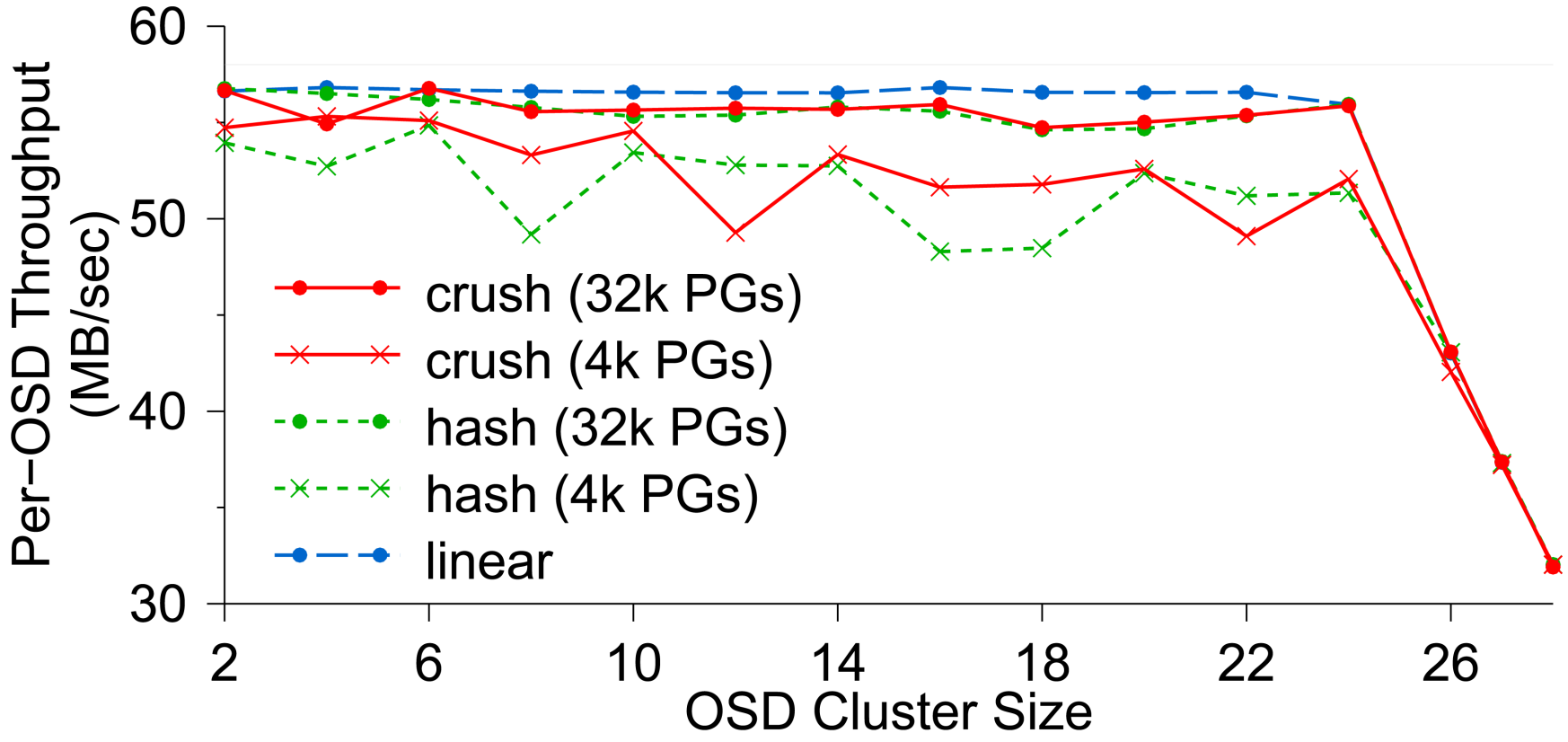
EBOFS PERFORMANCE



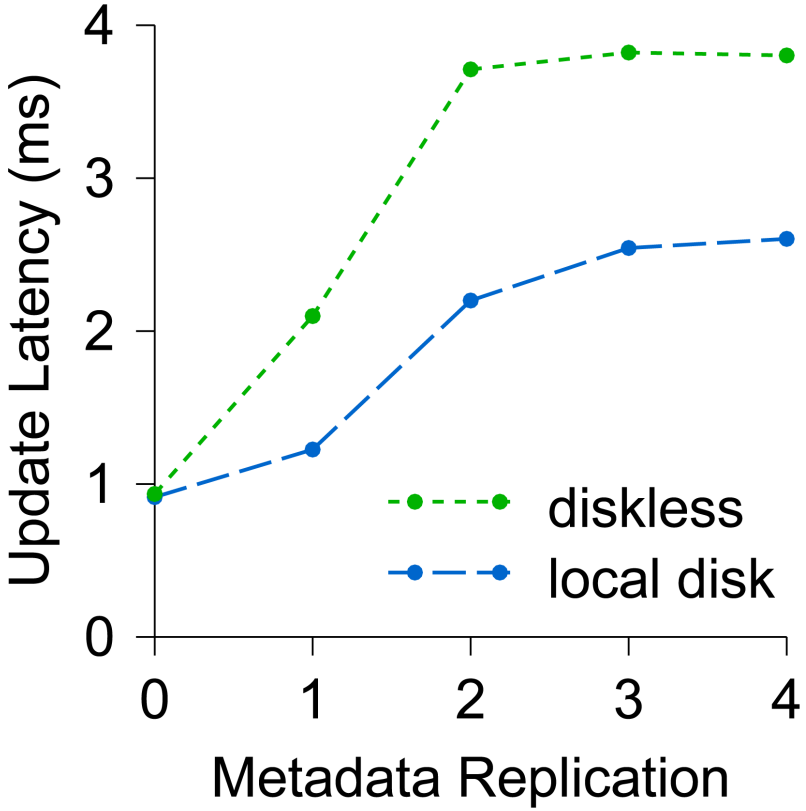
WRITE LATENCY

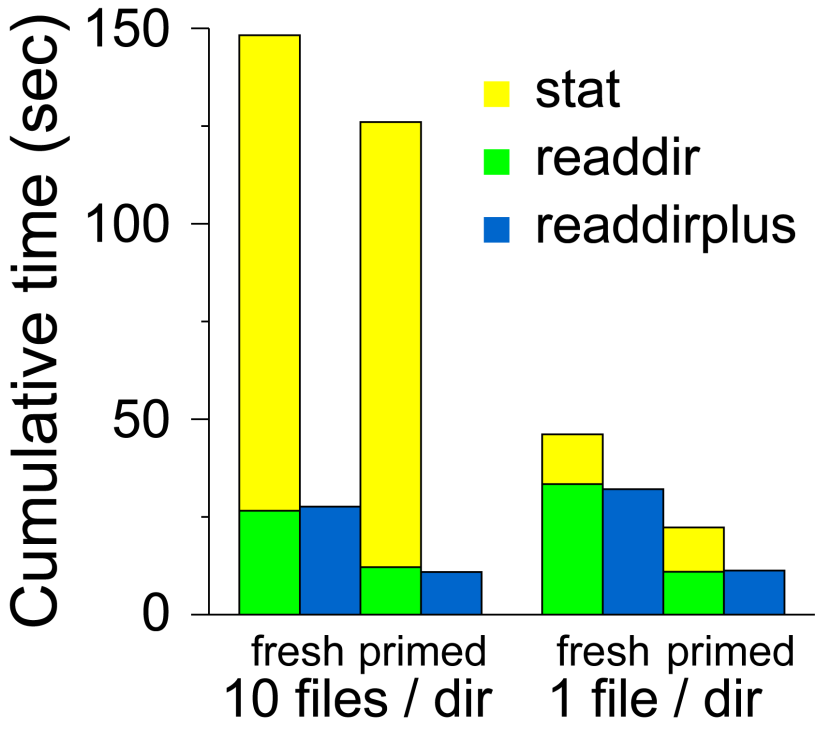


OSD WRITE PERFORMANCE



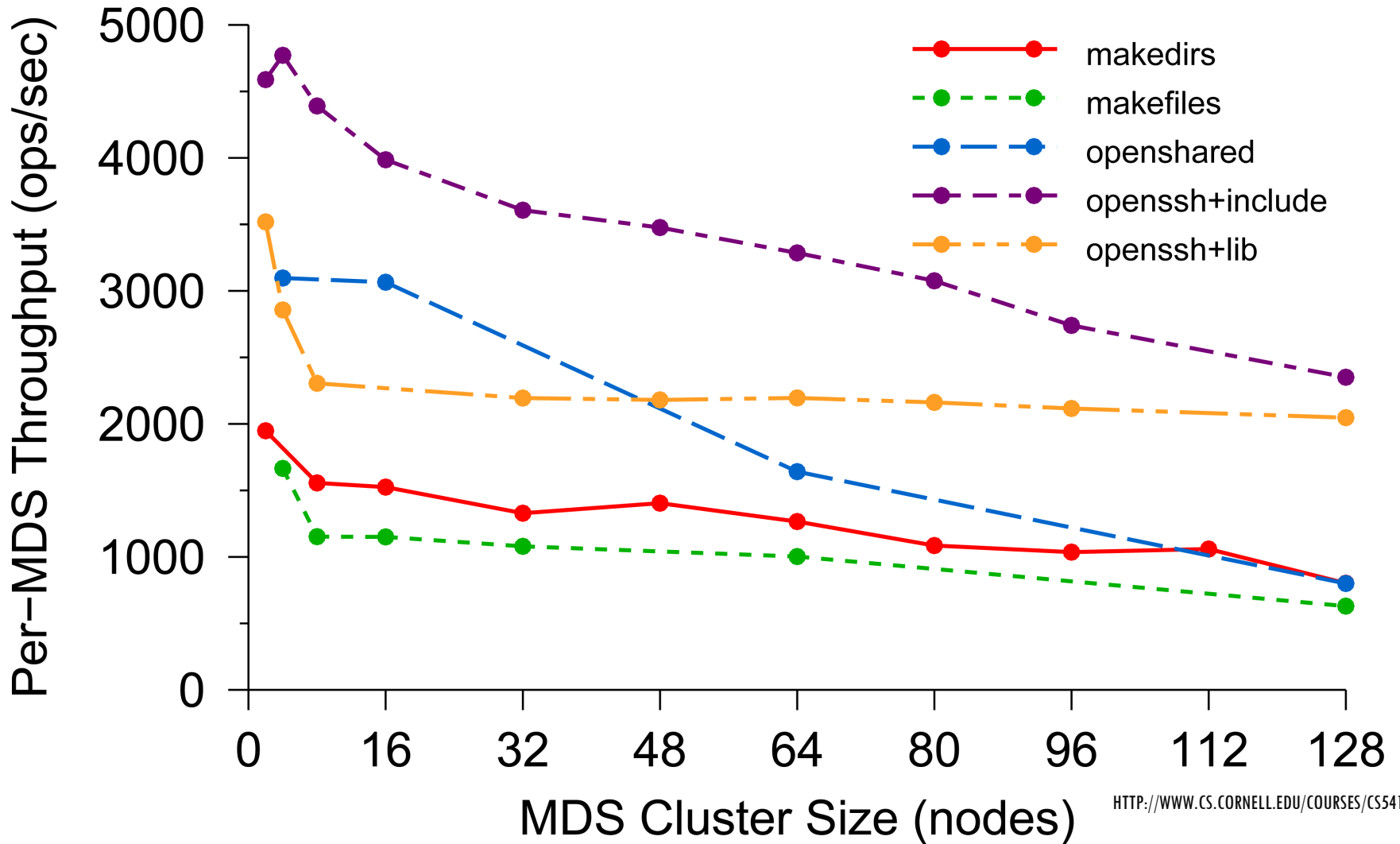
DISKLESS VS. LOCAL DISK



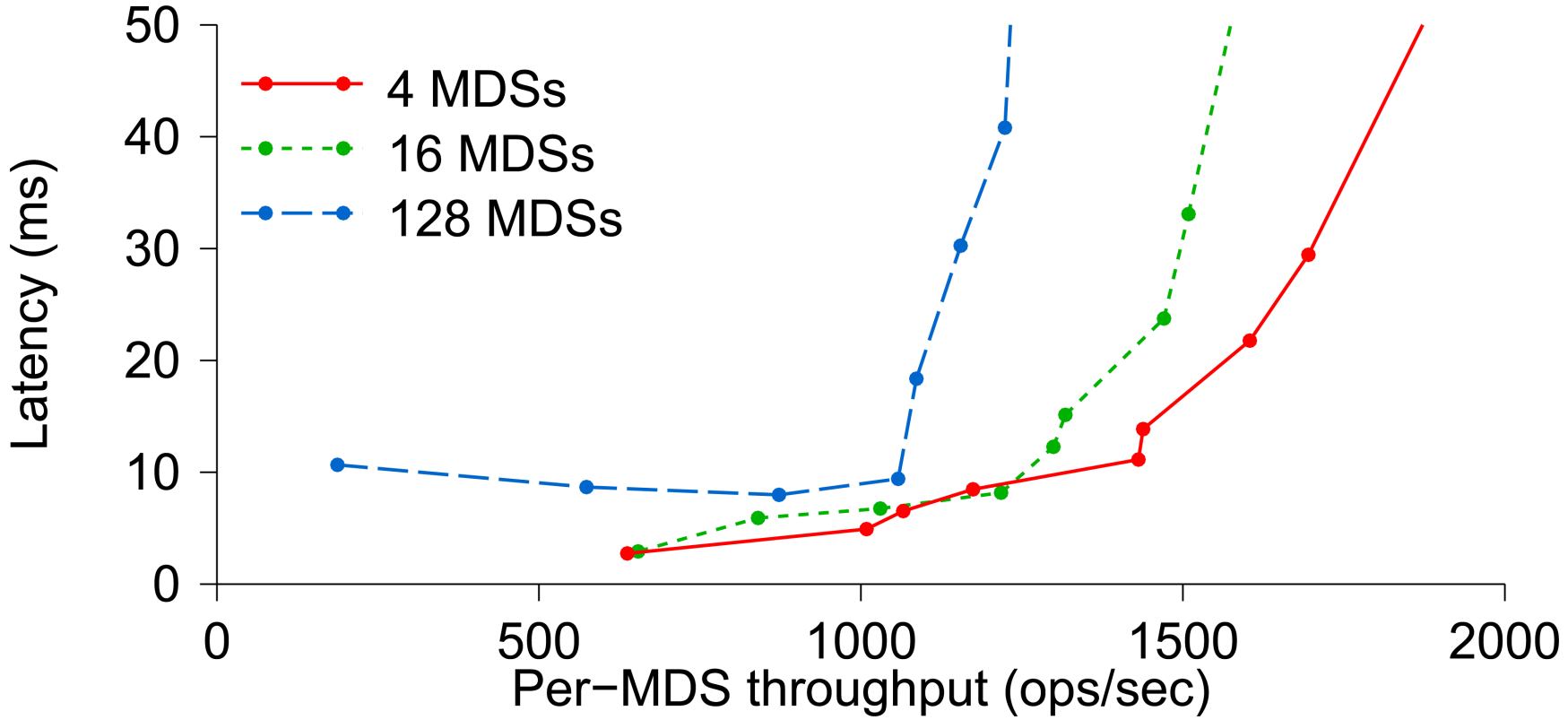


Compare latencies of (a) a MDS where all metadata are stored in a shared OSD cluster and (b) a MDS which has a local disk containing its journaling

PER-MDS THROUGHPUT



AVERAGE LATENCY



LESSONS LEARNED

If applications are object oriented, they will write huge numbers of variable-size records (some extremely large).

POSIX directories are awkward for large numbers of files, and POSIX is inefficient for whole-file read/write.

- A B+ tree index works much better than directory
- The library of POSIX API methods can "hide" the excess costs of **open/close**
- Object-oriented code won't use **lseek,** file "append" is all we really need

Ceph treats the records as byte arrays, track meta-data in one service and data in a second one. Both share the RADOS layer for actual data storage.

INTEROPERABILITY FOR OBJECTS

OMG AND CORBA

Like Ceph, CORBA is associated with the "Object Management Group" or OMG.

The original role of the OMG was to propose a standard way to translate between internal representations of objects and byte array external ones. They call this the Common Object Request Broker Architecture or CORBA.

WHY ADOPT OMG/CORBA?

Standardization is extremely valuable in large-scale settings, reduced vendor lock-in and increases options when searching for support

High quality software-engineering platforms, many tools and solutions

Easy of interoperation when object-oriented applications are created from subsystems coded in a variety of popular programming languages

YET STANDARDS AREN'T FREE

In the remaining section of this lecture we will focus on the overheads of using CORBA

We will see that they aren't really a "gotcha" in the sense that if you understand them, you can avoid extremely costly designs

But if you use CORBA and ignore its overheads, you could design a solution that might have unnecessarily high overheads

UNDERSTANDING COSTS FOR CORBA'S UNIVERSAL REPRESENTATIONS: ATC SYSTEM

A modern air traffic control system might have a structure like this:

Air traffic controllers update flight plans

Flight plan manager tracks current and past flight plan versions

Message bus

. . .

Microservices for various tasks, such as checking future plane separations, scheduling landing times, predicting weather issues, offering services to the airlines

Flight plan update broadcast service

WAN link to other ATC centers

UNDERSTANDING COSTS FOR CORBA'S UNIVERSAL REPRESENTATIONS: ATC SYSTEM

Think about objects in an ATC system:

- Flight plans: these are elaborate objects that might hold 10MB of data and could have a great many internal fields
- Many other kinds of objects are used too. Each microservice probably has a notifications channel of its own, and uses it to talk to individual controllers or sets of them about relevant issues
- "***Attention: In 2h 31m, BA 123 will approach US 654 on approach to CDG. Plan corrective action to avoid a violation of flight separation rules.***"

UNDERSTANDING COSTS FOR CORBA'S UNIVERSAL REPRESENTATIONS: ATC SYSTEM

An ATC system has many components, far more than were shown.

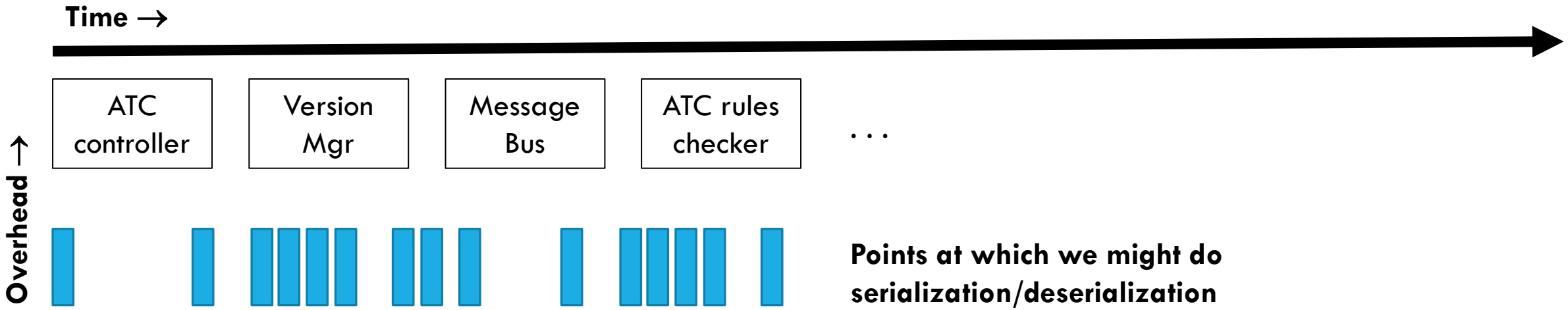
Often these are based on high-quality legacy versions and hence there can be many programming languages in simultaneous use.

- Often we will see C/C++, Java, C#, F#, O'Caml, etc.
- Some use of Python and Fortran and Ada.
- With CORBA, we can easily integrate many modules into a single system

BUT HOW OFTEN WILL WE (DE)SERIALIZE?

Each time an object is read or written (from disk or network)

Each time an object is passed from one module to another



CORBA SERIALIZATION: VERY ELABORATE

Some languages have "unique" data types. For example, Python integers are automatically implemented using BigNum arithmetic and have no size limit or risk of overflow/underflow.

But interoperability with C++ or Fortran, which have size-limited integers, becomes tricky: what format to use? What rules?

CORBA standardizes all such policies, at the cost of a fancy encoding

UNIVERSAL REPRESENTATIONS ARE COSTLY!

It is very easy for a CORBA application to spend *all* its time on this one action.

Ceph designers were aware of that, and decided it should only be done under application control.

This is one reason that Ceph forces the developer to serialize / deserialize objects: That policy eliminates the need for CORBA data representations

HOW DO ATC SYSTEMS AVOID THESE COSTS?

If they aren't careful… they will pay the high price! But there is a way around them, which is to use "lazy" record access.

The ATC record is the main object being shared. Suppose that we have two versions of an ATC object while in memory:

- Version A: The object is fully resident in memory and you can access all fields, edit it to create a new version, etc.
- Version B: All the same methods are offered, but the in-memory data is limited to a URL pointing to the record in the flight plan database

WHY TWO "IDENTICAL" OBJECT VARIANTS?

Notice how easy it is to switch from representation B to A (or back).

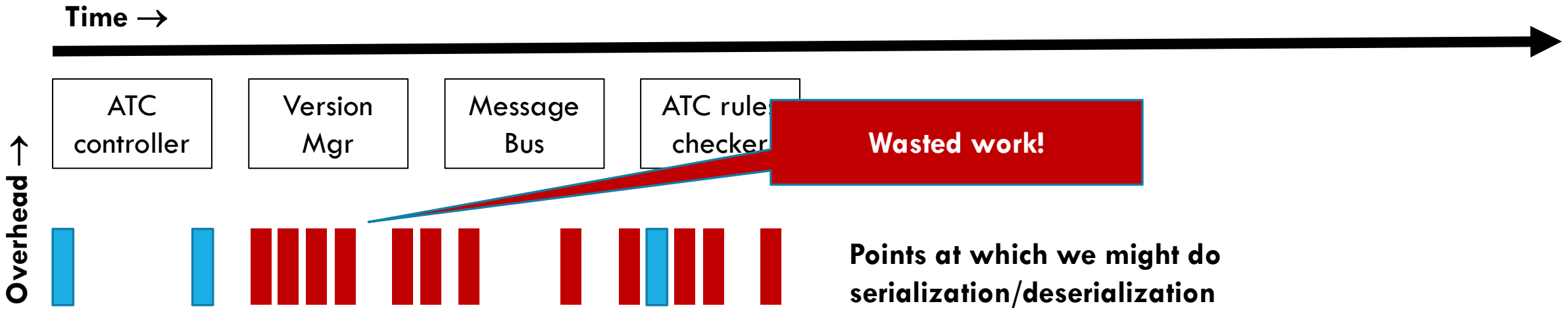
In an ATC system most components don't really look at the data fields and for this reason, most components would be happy with representation B. But a small object with just a URL in it is very cheap to serialize!

With "lazy deserialization", we would convert from form B to form A only when an application tries to touch the data.

OLD SINGLE VERSION APPROACH

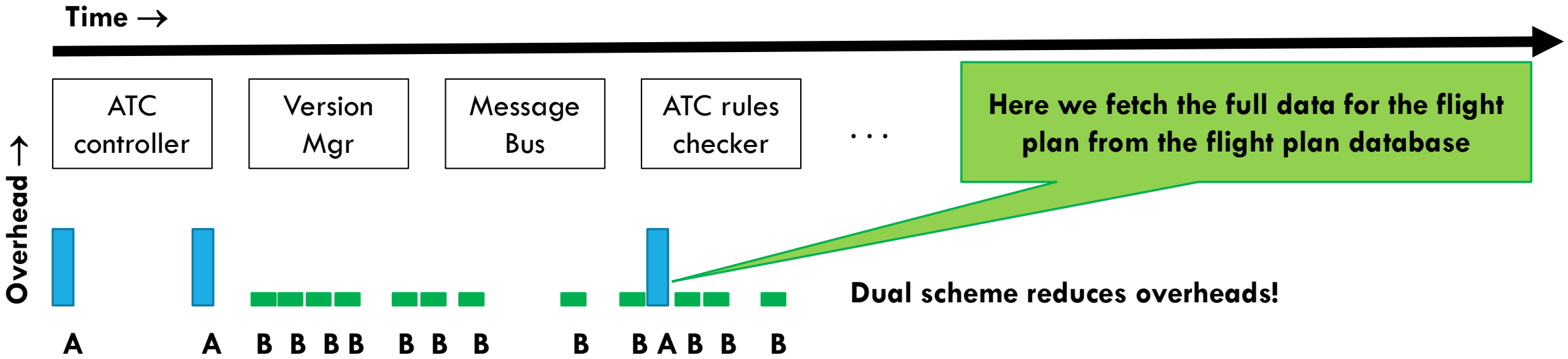
Each time an object is read or written (from disk or network)

Each time an object is passed from one module to another



DUAL VERSION APPROACH

We only do a costly action when the component will actually touch the inner data fields!



HOW SHOULD WE STORE THE FLIGHT PLAN RECORDS?

The need is for a very simple append-only log managed by the version manager.

It is easy to recognize this as a use case for state machine replication.

This situates the central safety question in one specific component, where we can formalize it and use mathematical tools to prove that each plan has just one sequence of versions, used consistently by all components.

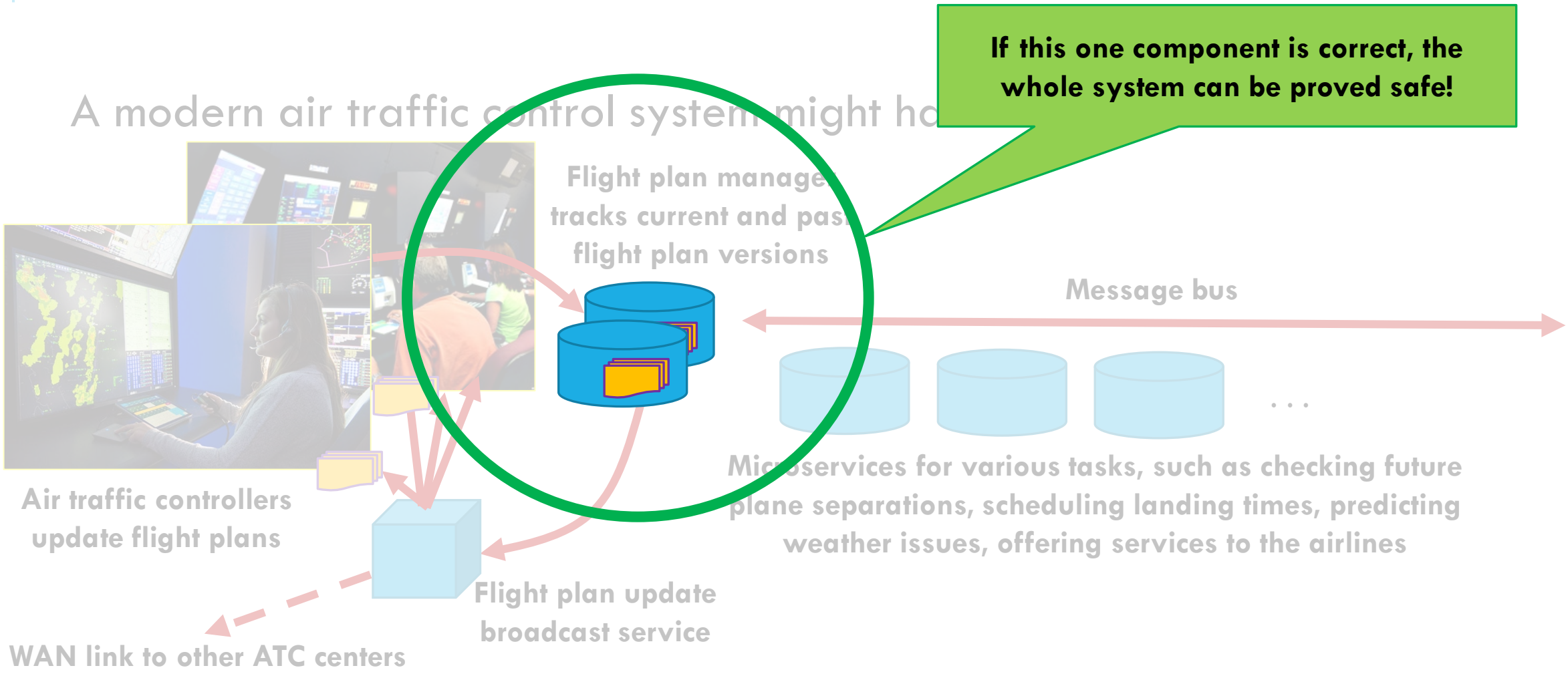
HOW SHOULD WE IMPLEMENT THE FLIGHT PLAN MANAGER COMPONENT?

A (key-value) sharded service built on Derecho would be an ideal choice.

Derecho has been proved correct in several ways: by hand, but also using a machine-verified proof in the Ivy protocol verification tool.

It is also scalable and extremely fast: important because this role is central.

REVISITING THE STRUCTURE



SUMMARY – CEPH

Ceph is a file system that was created by taking the HDFS model, but then extending it to be better matched to properties of object-oriented code.

This is very popular, although it does bring overheads.

Ceph uses a simple but "weak" form of data replication. It doesn't guarantee consistency.

SUMMARY – CORBA

Here we saw a different form of object-oriented overhead, arising in applications that adopt the standard CORBA approach to interoperability.

CORBA buys flexibility but brings steep costs.

Those costs can be managed by *understanding* the architecture and *designing* the application to avoid triggering costly serialization/deserialization.

BROADER INSIGHT LINKING THESE SUBTOPICS

Innovation brings challenges. And object orientation was a big innovation.

Strongly typed object oriented languages like Java, C++, Python reduce errors and improve productivity. But they sometimes clash with very old APIs that predate this style of computing.

- Modern services are evolving to enhance object-oriented support.
- Tuples held in SQL relations, NoSQL key-value pairs, rows in CSV files and many other things can all be viewed as "objects".