AWS re:Invent

STG405

Maximizing Amazon EFS performance for Linux workloads

Geert Jansen
Sr. Product Manager
Amazon EFS
AWS

Darryl Osborne
Solutions Architect
File Storage
AWS

AWS re:Invent

aws

Agenda

- Amazon EFS Overview
- Using Amazon EFS
- Network File System
- Close-to-open semantics
- Optimizing NFS client settings
- Optimizing scripts and applications
- Summary

Related breakouts

STG201	AWS leadership session: Storage state of the union
STG202	What’s new in AWS file storage
STG304	Network File System (NFS) evolved: Deep dive on Amazon EFS
STG312	Securing Amazon EFS for modern applications and data science
STG403	Security best practices with Amazon EFS

Amazon EFS overview

Amazon Elastic File System (Amazon EFS)

Amazon EFS
is a fully managed file system that is

- **Cloud native**
- **Highly reliable**
- **Cost optimized**

Elastic

- Grow & shrink on demand
- No need to provision and manage infrastructure & capacity
- Pay as you go, pay only for what you use

Scalable

- Grow up to petabytes
- Performance modes for low latencies and maximum I/O
- Throughput that scales with storage
- Provisioned throughput available

Integrated

- Shared access from on-premises, inter-region, and cloud native applications
- Integrated with various AWS computing models
- Access concurrently from thousands of Amazon EC2 instances
- Attach to containers launched by both Amazon ECS and Amazon EKS
- Use with Amazon SageMaker notebooks

Cloud native

Highly available, durable

- Stores data across three Availability Zones for high availability and durability
- 99.9% availability SLA
- Designed for 11 9's of durability

Secure

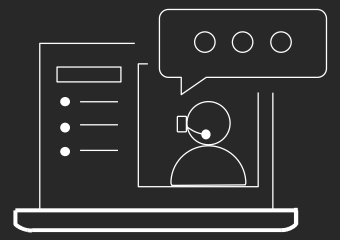
- Control network traffic
- Control file and directory access
- Control administrative (API) access
- Encrypt data at rest and in transit

Global footprint

- Amazon EFS is available in 19 Regions
- New Regions recently added: Bahrain, Sao Paulo, Stockholm, Hong Kong

Highly reliable

No minimum commitments or upfront fees



No need to provision storage

Use with Spot Instances

Automatic lifecycle management

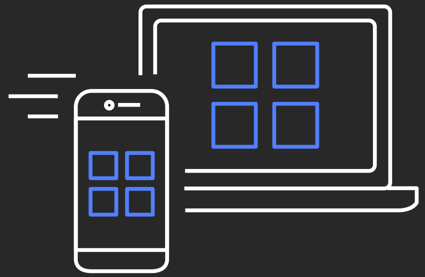
Cost optimized

Use cases for Amazon EFS

Home directories

Container storage

Application test/dev

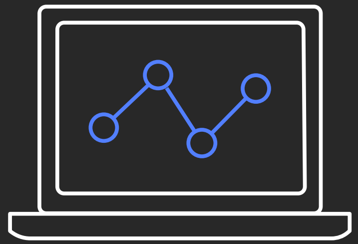


Lift and shift enterprise apps

Web serving

Content management

Database backups



Analytics

Media workflows

Metadata-intensive jobs

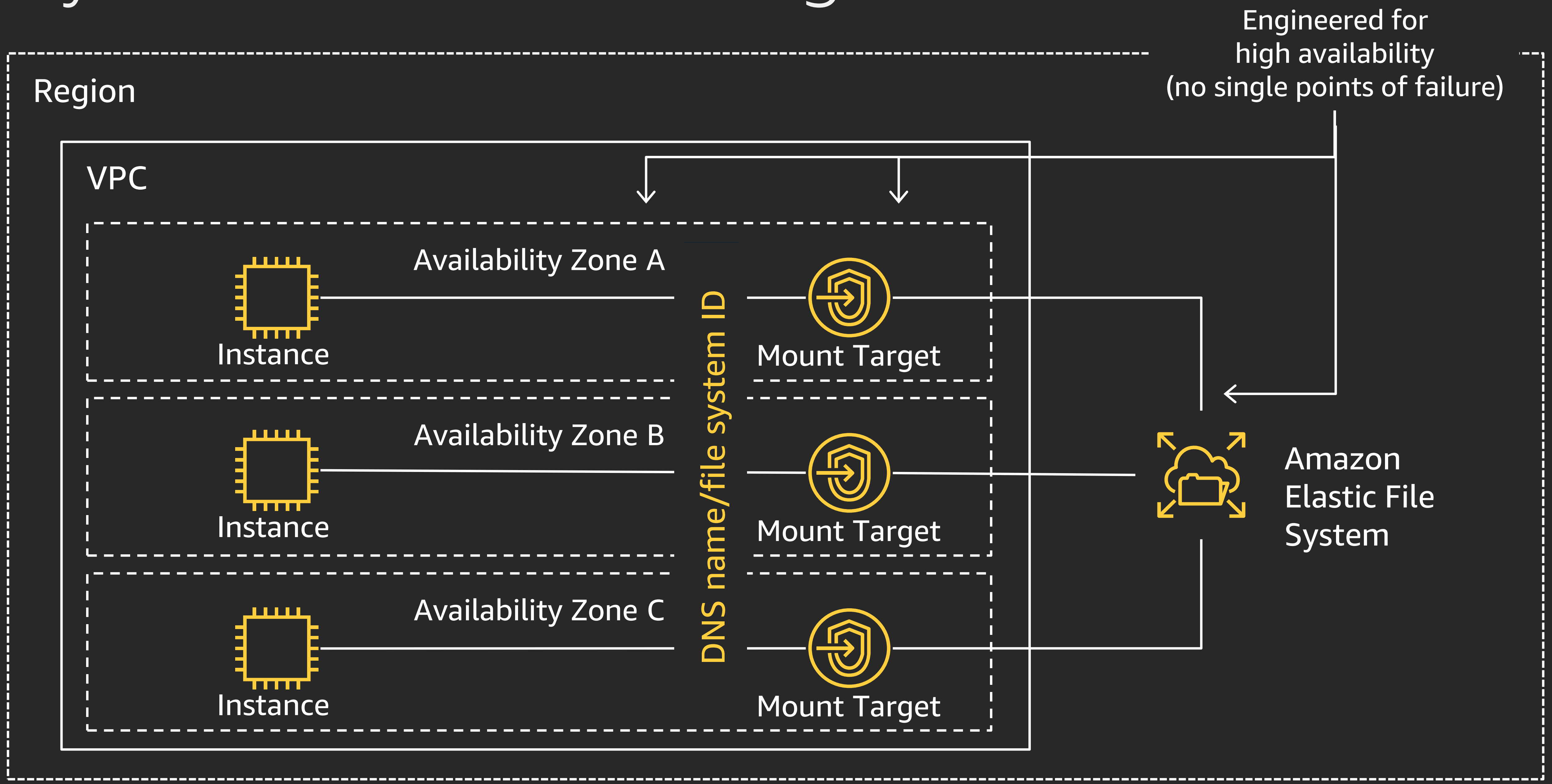
Low latency and serial I/O

Scale-out jobs

High throughput and parallel I/O

Using Amazon EFS

File Systems and Mount Targets



Mounting a File System

Recommended: Use the Amazon EFS mount helper to mount a file system

```
$ sudo mount -t efs fs-deadbeef /efs
```

Equivalent to:

```
$ sudo mount -t nfs4 \
   -o vers=4.1,rsize=1048576,wsize=1048576,hard,noresvport \
   fs-deadbeef.efs.us-east-1.amazonaws.com /efs
```

NFS overview

Network File System – History

1984: Sun Network Filesystem
- Presented at USENIX 1985
- Used internally at Sun Microsystems

1989: NFSv2
- RFC1094
- UDP, Stateless

1995: NFSv2
- RFC1813
- TCP, 64-bit file sizes
- READDIRPLUS

2000: NFSv4
- RFC3010, 3530, 7530
- Single protocol (port 2049).
- Introduces minor versions
- Compound operations, delegations, backchannels
- ACLs, Windows support
- RPCSEC_GSS (Kerberos)

2010: NFSv4.1
- RFC5661
- State management (sessions).
- Parallel NFS

2016: NFSv4.2
- RFC7862
- Optional data management features: Server-side copy, application data block, sparse file hole seeking

Today

Version supported by Amazon EFS

Network File System – Layers

Network File System
Open Network Computing Remote Procedure Calls (ONC RPC)
External Data Representation (XDR)

Network File System – Visualizing

Visualizing network traffic essential to getting a good understanding of NFS and learning to troubleshoot

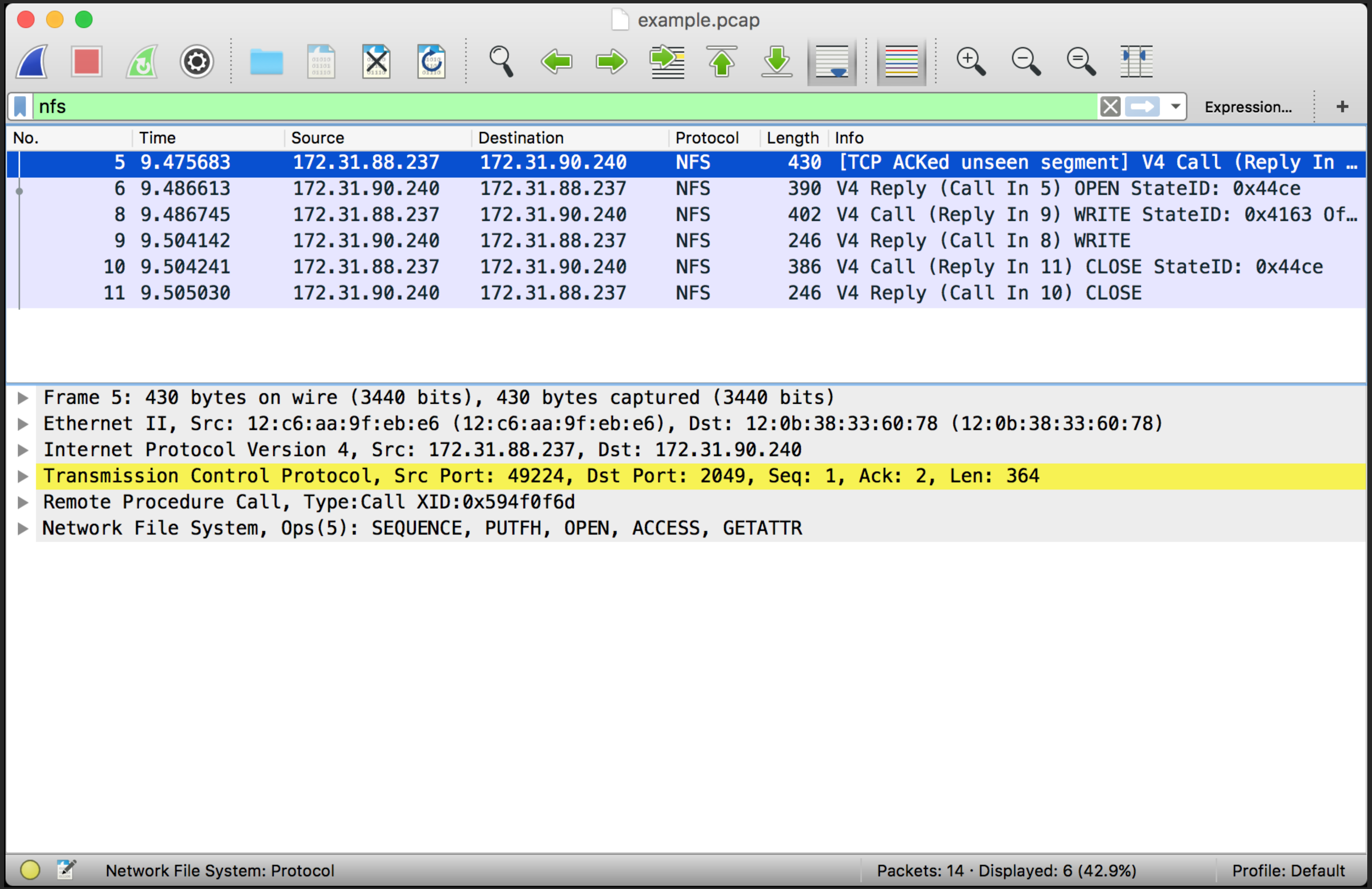
On EC2 instance:

```
$ sudo tcpdump –w out.pcap port 2049 &
$ # run your command
$ fg + <CTRL-C>
```

On your desktop:

```
$ scp instance:out.pcap .
$ wireshark out.pcap
```

Wireshark example



$ echo bar > /efs/foo.txt

Wireshark NFS details

Compound #1: OPEN

```
▼ Network File System, Ops(5): SEQUENCE PUTFH OPEN ACCESS GETATTR
    [Program Version: 4]
    [V4 Procedure: COMPOUND (1)]
    Status: NFS4_OK (0)
  ▶ Tag: <EMPTY>
  ▼ Operations (count: 5)
    ▶ Opcode: SEQUENCE (53)
    ▶ Opcode: PUTFH (22)
    ▶ Opcode: OPEN (18)
    ▶ Opcode: ACCESS (3), [Access Denied: XE], [Allowed: RD MD
    ▶ Opcode: GETATTR (9)
    [Main Opcode: OPEN (18)]
```

Compound #1: WRITE

```
▼ Network File System, Ops(4): SEQUENCE, PUTFH, WRITE, GETATTR
    [Program Version: 4]
    [V4 Procedure: COMPOUND (1)]
  ▶ Tag: <EMPTY>
    minorversion: 1
  ▼ Operations (count: 4): SEQUENCE, PUTFH, WRITE, GETATTR
    ▶ Opcode: SEQUENCE (53)
    ▶ Opcode: PUTFH (22)
    ▼ Opcode: WRITE (38)
      ▶ StateID
        offset: 0
        stable: FILE_SYNC4 (2)
        Write length: 4
      ▼ Data: <DATA>
          length: 4
          contents: <DATA>
    ▶ Opcode: GETATTR (9)
    [Main Opcode: WRITE (38)]
```

Close-to-open consistency

NFS close-to-open semantics

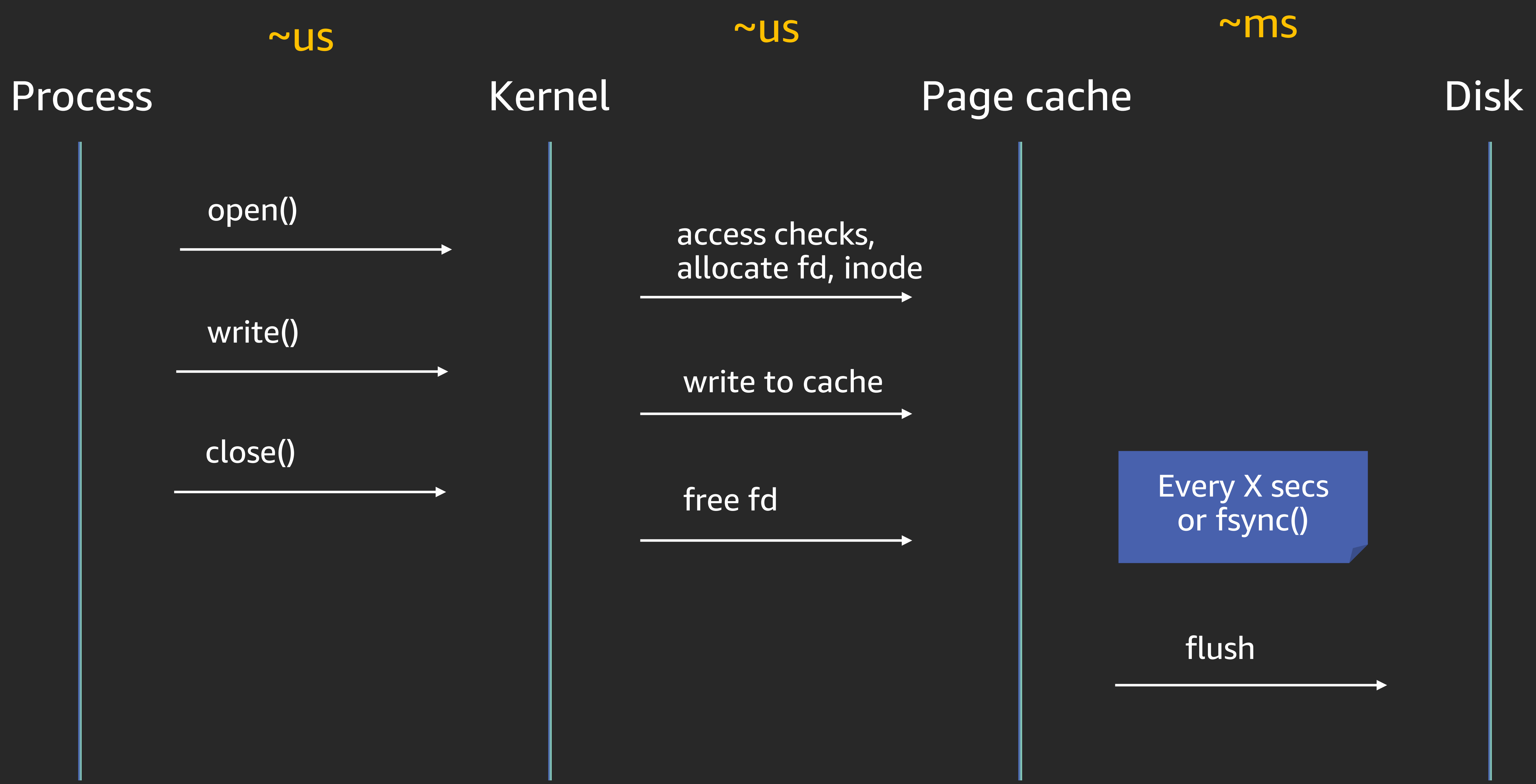
- Why does a local file system on an HDD @10ms latency have better small file performance than an NFS server @1ms

- NFS ‘chattiness’ is one contributing factor. Still requires 3–7 round trips to read or write to small file depending on options used

```
LOOKUP DH: 0x65d4f94b/pre-push.sample
LOOKUP Status: NFS4ERR_NOENT
OPEN DH: 0x65d4f94b/pre-push.sample
OPEN StateID: 0x23d6
SETATTR FH: 0x471d3813
SETATTR
WRITE StateID: 0x267b Offset: 0 Len: 1348
WRITE
CLOSE StateID: 0x23d6
CLOSE
```

But there’s a more important reason

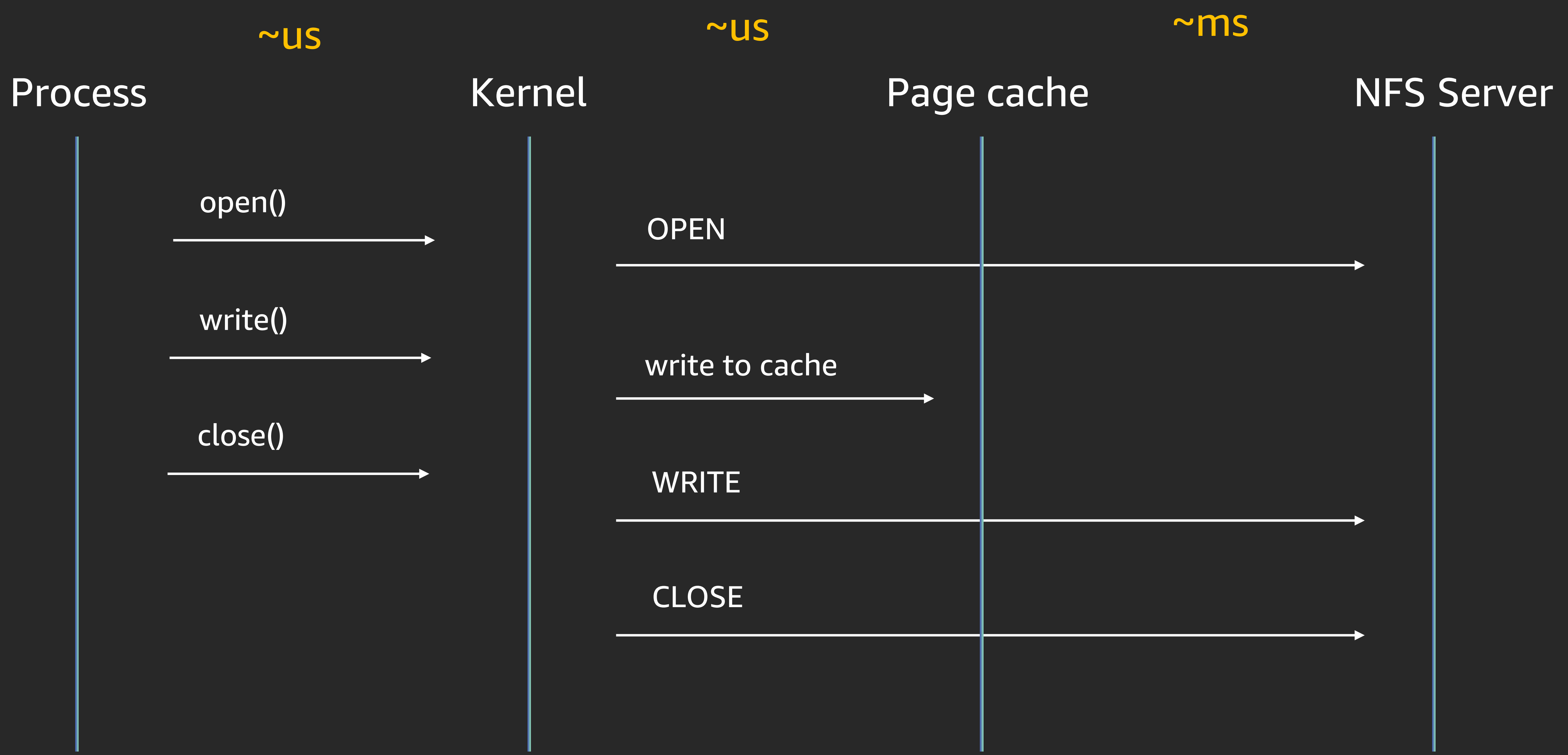
Writing a file on a local file system



Close-to-open semantics

- With a local file system, all processes share the cache, so the results of a write() are immediately visible even if they are not on disk yet
- Processes on different NFS clients do not share a cache. How do other processes learn about changes made by other clients?
- POSIX does NOT provide an answer here
- Disabling the cache? No, would be prohibitively expensive.
- NFS answer: Close to open semantics
  - Flush all data when a file is closed
  - Revalidate cache when it’s opened

Writing a file on NFS



Optimizing NFS client settings

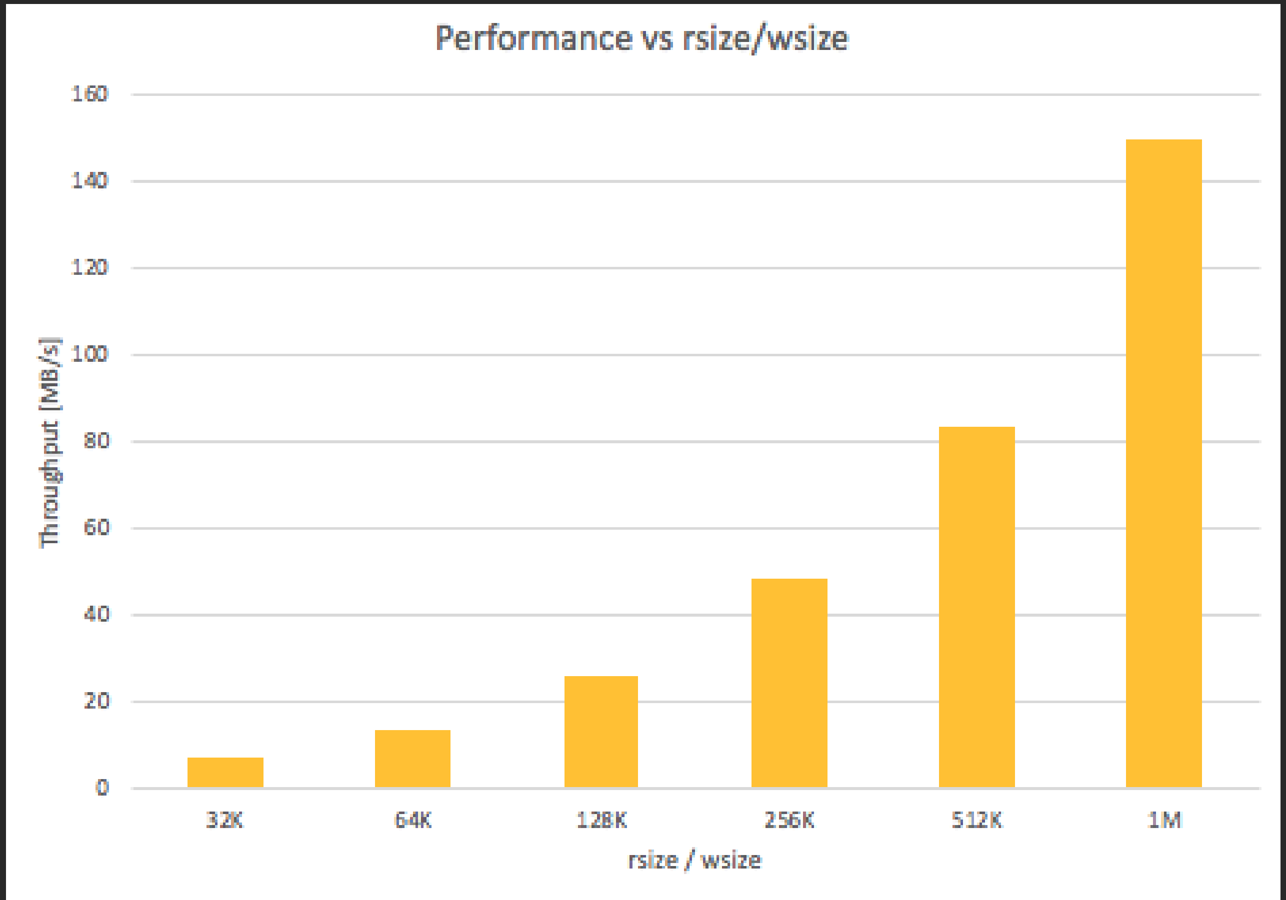
AWS re:Invent

aws

Mount options – rsize= / wsize=

- Maximum read and write size for NFS client
- Too small I/O sizes will limit throughput on large files
- Recommended: 1M / 1M

```
▼ Network File System, Ops(3): PUTFH, WRITE, GETATTR
    [Program Version: 4]
    [V4 Procedure: COMPOUND (1)]
  ▶ Tag: <EMPTY>
    minorversion: 0
  ▼ Operations (count: 3): PUTFH, WRITE, GETATTR
    ▶ Opcode: PUTFH (22)
    ▼ Opcode: WRITE (38)
      ▶ StateID
        offset: 5242880
        stable: UNSTABLE4 (0)
        Write length: 1048576
      ▶ Data: <DATA>
    ▶ Opcode: GETATTR (9)
```



$ dd if=/dev/zero of=/efs/1GB.bin bs=1M count=1024

Mount options – hard versus soft

- “hard”: continue retrying NFS operations indefinitely
- “soft”: retry up to “retrans” times (default: 3)
- Recommend: “hard” (unless you want silent data corruption)

$ man 5 nfs

```
NB: A so-called "soft" timeout can cause silent data corruption in certain cases. As such,  use  the
soft  option  only when client responsiveness is more important than data integrity.  Using NFS over
TCP or increasing the value of the retrans option may mitigate some of the risks of using  the  soft
option.
```

Mount options – noresvport

- The “noresvport” option uses a high port (> 1024) on the client
- This option has the side-effect that the source port number will change on reconnects
- Critical when using stateful network filtering like security groups or iptables
- Currently a fix is in linux-next:

```
commit e6237b6feb37582fbd6bd7a8336d1256a6b4b4f9
Author: Trond Myklebust <trond.myklebust@hammerspace.com>
Date:   Thu Oct 17 11:13:54 2019 -0400

    NFSv4.1: Don't rebind to the same source port when reconnecting to the server
```

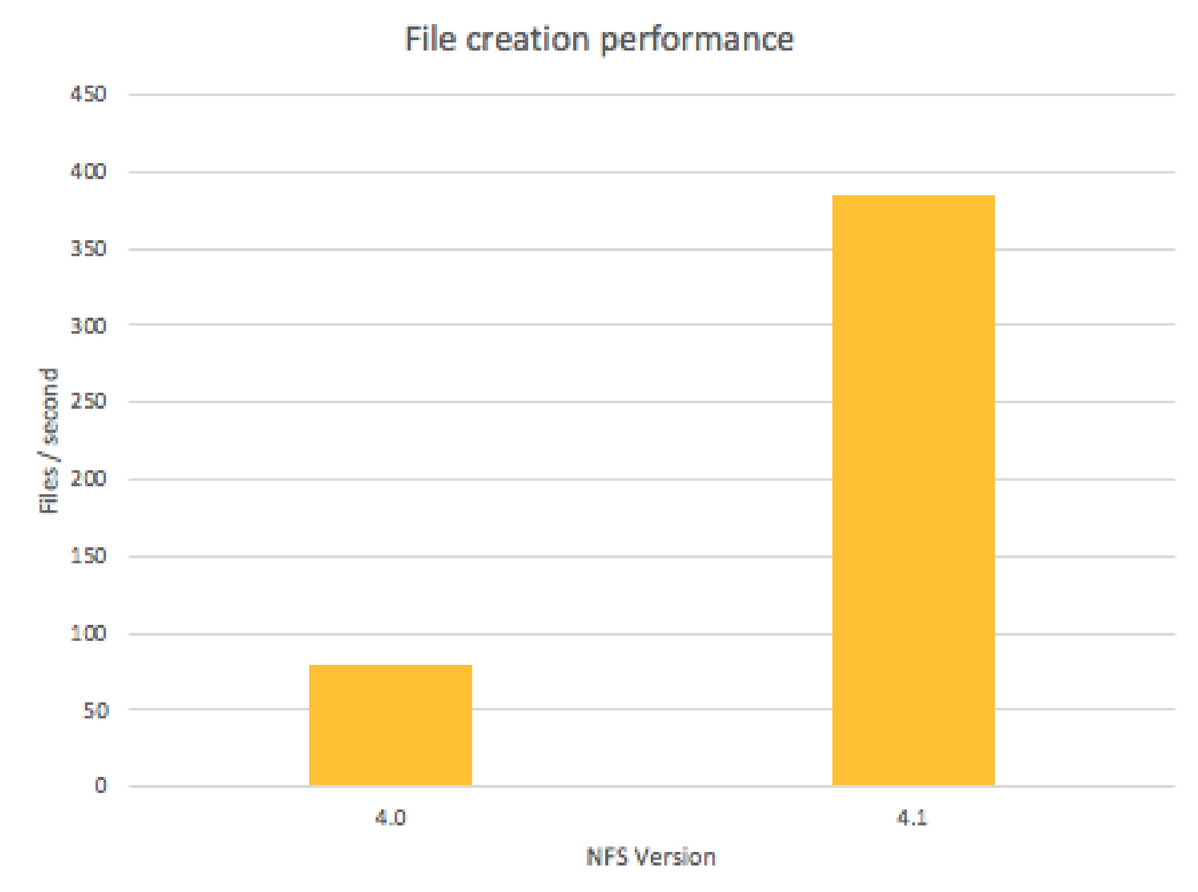
To avoid that problem, the sunrpc code in recent Linux kernels works around this by using a little known method for disconnecting a TCP socket without going into `TIME_WAIT`, which allows it to reuse the same port immediately.

Strictly speaking, this is in violation of the TCP specification. While this avoids the problem with the reply cache, it remains to be seen whether this entails any negative side effects—for instance, how gracefully intermediate firewalls may deal with seeing SYN packets for a connection that they think ought to be in `TIME_WAIT`.

Olaf Kirsch, 2006, Proceedings of Linux Symposium, p56

NFS version

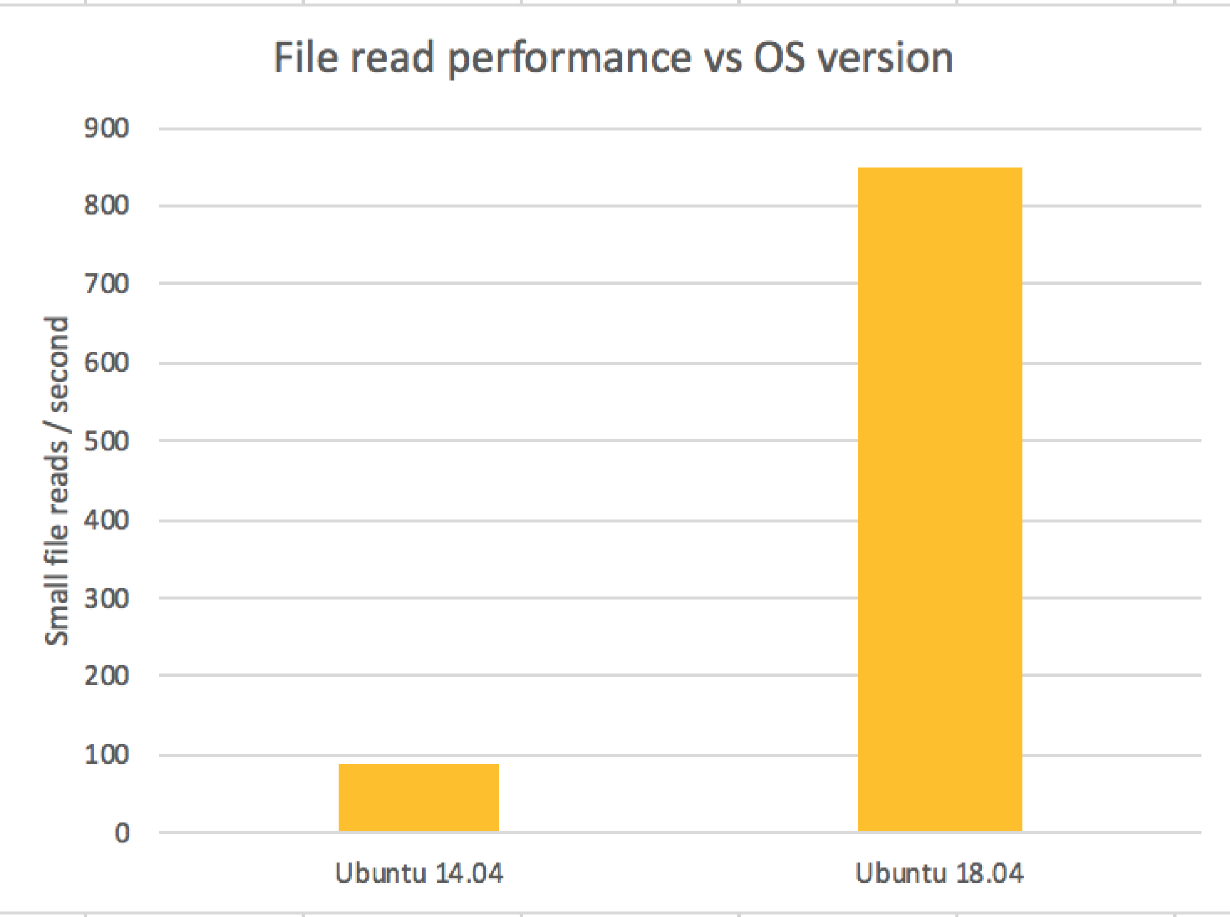
- Always use NFSv4.1
- 4.1 client is more stable and performant than 4.0



Parallel file creation benchmark,
AWS Internal, Amazon Linux 2

Kernel versions

- Use a recent kernel version. Generally, the newer the better.
- For example, 2,000+ changes to NFS client since RHEL 7 was released in 2014
- Recommended distros / kernels:
    - Amazon Linux 2, or 2015.09 and newer
    - RHEL 7.3 or newer
    - Ubuntu 16.04 or newer
    - SLES 12 Sp2 or later
    - Any distributions with kernel 4.3 or higher




```
$ time find /efs/dir -type f \
  | xargs -n 100 -P <parallel> cat > /dev/null
```

Not recommended

- FScache
  - Fscache can help for bandwidth constrained system but does not improve latencies
- sync, actimeo=0, acregmax=0, and others
  - Makes file system operations synchronous (sync), disables attribute caching (actimeo)
  - Large performance penalty
  - Nothing you can do will make NFS behave exactly like a local file system across multiple instances. Multi-instance applications will need to be aware they are working with an NFS file system.
- lookupcache=pos, lookupcache=none
  - Disables negative and full lookup cache respectively
  - Large performance penalty
  - Sometimes suggested by vendors

Optimizing Scripts and Applications

Small file performance
Throughput
Other optimizations

AWS re:Invent

Small file performance

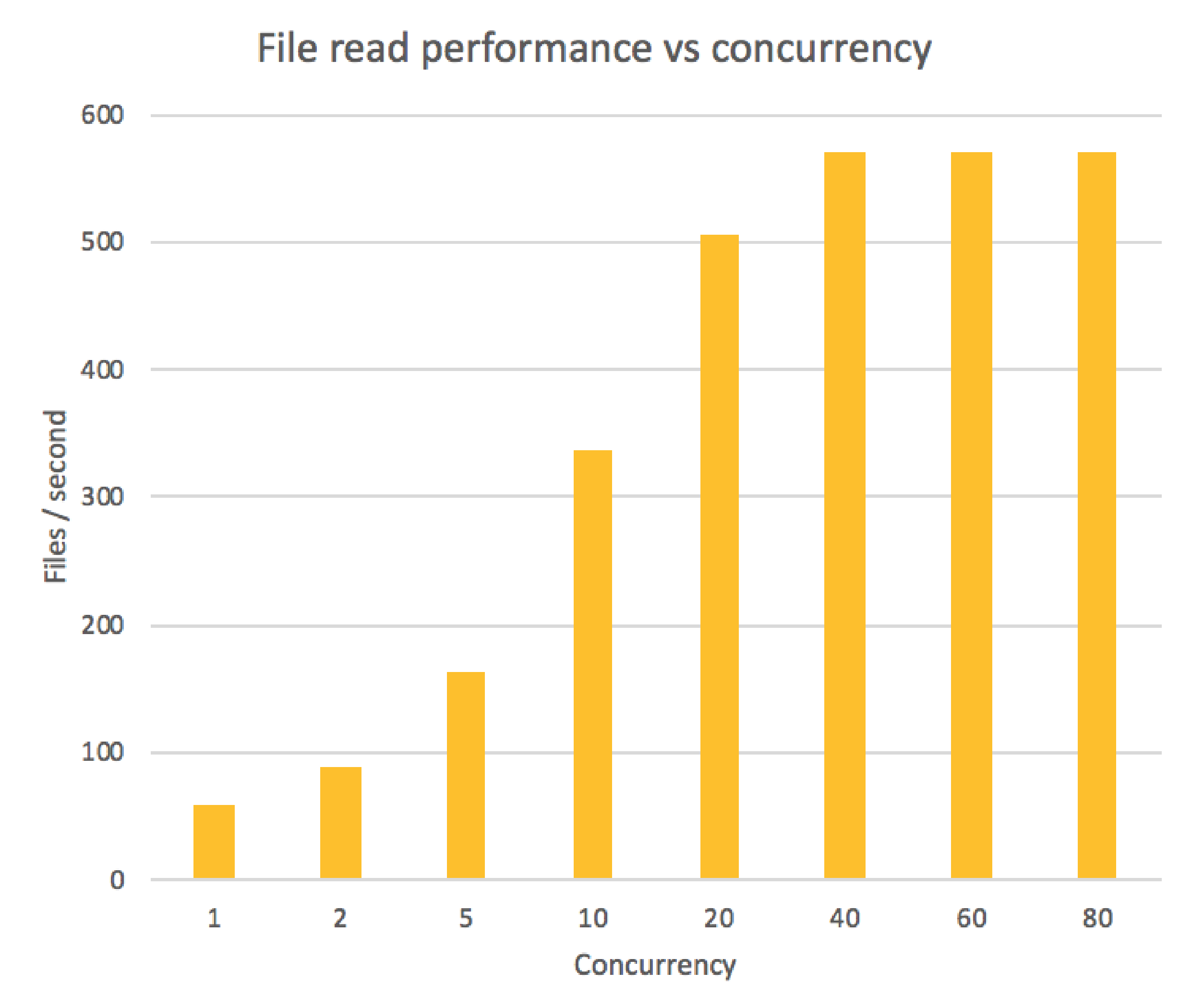
- In general, close-to-open semantics mean that for serial workloads, NFS will have lower performance for small file workloads than a local file system
- Choose correct file system performance mode
  - General purpose: Lowest metadata latencies, 7,000 IOPS per file system (announcement!)
  - MaxIO: Higher metadata latencies, 500K+ IOPS
- Parallelize to benefit from Amazon EFS scale-out architecture
  - Shell: xargs -P, GNU parallel
  - Software: thread pools, async I/O
  - Multi-instance applications

Scaling small file performance – single instance

- Simple read benchmark:

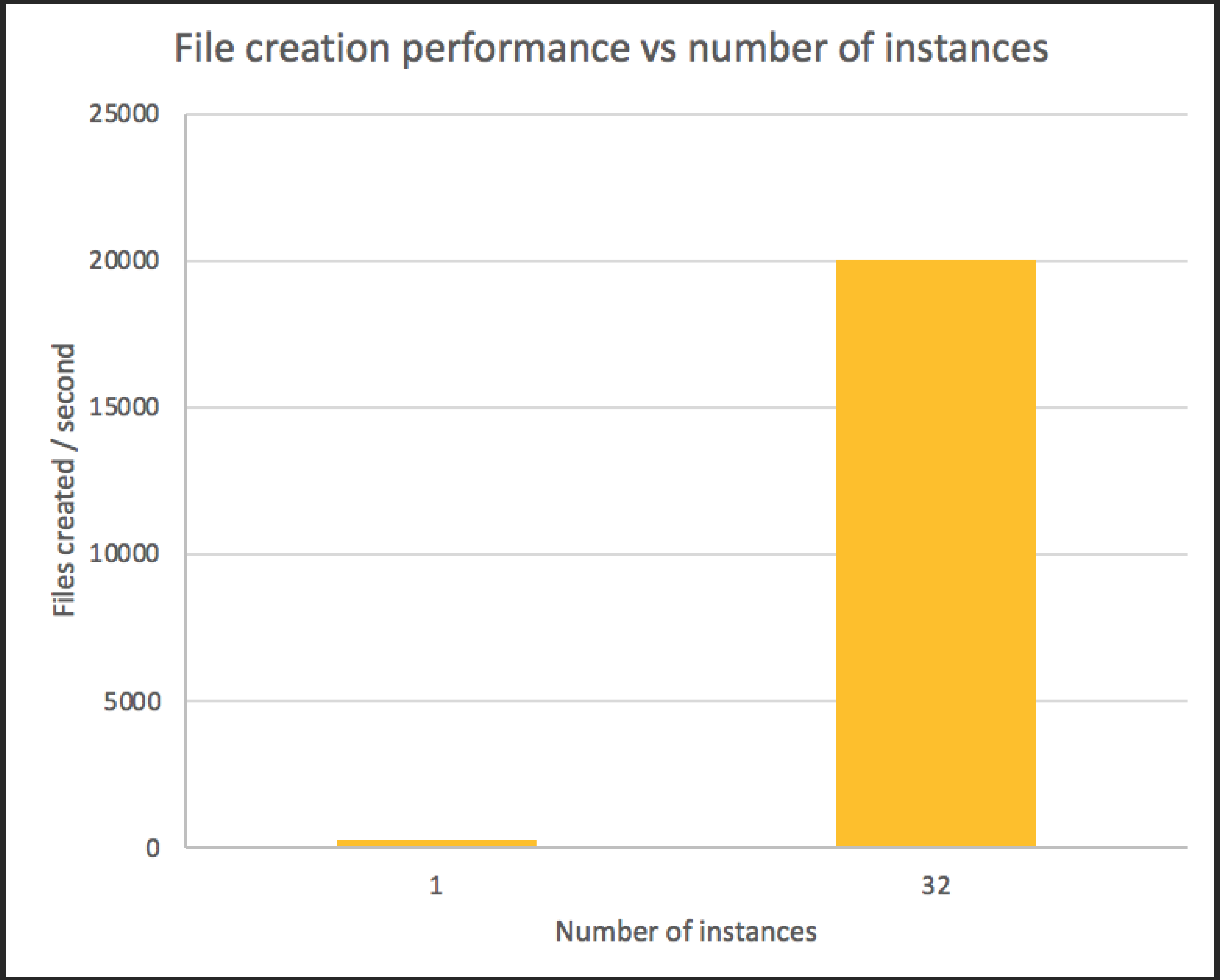
```
$ time find /efs/<dir> -type f \
  | xargs -n 100 -P <parallel> \
    cat > /dev/null
```

- Used 10G file system with average file size of 80K and 20 files/directory
- Recommendation: use a concurrency of up to 40 per NFS client



Scaling small file performance – multiple instances

- Once you saturate the IOPS of a single instance, move to a multi-instance deployment
- With 2-3 instances, you will likely exhaust the IOPS capacity of a GP mode file system. Need to move to MaxIO to scale further.
- To get the maximum IOPS possible, use multiple AZs



Parallel file creation benchmark, AWS Internal, Amazon Linux 2

Optimizing throughput

- Amazon EFS throughput limits
  - 1 GB/s or 3 GB/s of aggregate throughput per file system (depending on region)
  - 250 MB/s per client throughput
- Choose correct throughput mode for your file system
  - Bursting mode: larger file systems get higher throughput. Base rate of 50 MB/s per TB of storage, bursting rate of 100 MB/s, minimum rate: 100 MB/s
  - Provisioned throughput: up to 1 GB/s independent of file system size
- Usually MaxIO is the best match for high throughput applications
- Use large IO sizes (but client will usually coalesce and read ahead)
- A single thread will be able to drive the per-client limit of 250 MB/s
- To get higher throughput, use multiple instances

Other optimizations

- Directory sizes
  - Directory lookups are O(1), directory scans are O(n)
  - Large directories are awkward to work with (e.g. delete, copy) and sensitive to client bugs.
  - Adding a file to a directory is synchronized operation
  - Recommendation: keep directory sizes < 10K

Summary

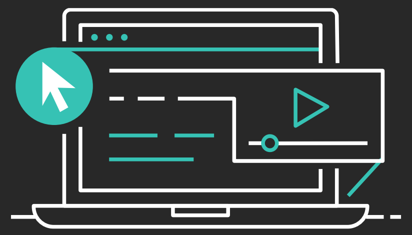
Summary

- Client optimizations
  - Use Amazon EFS mount helper to automatically get recommended mount options
  - Use recent kernel
- Small file optimizations
  - Choose correct performance mode: GP or MaxIO
  - Parallelize IO with a max concurrency of 40 per instance
  - Scale out the number of instances to achieve higher throughput
- Throughput optimizations
  - Choose correct throughput mode: bursting or provisioned
  - Use large IO sizes
  - Scale out number of instances.
- Keep directory sizes < 10K

Q&A

Learn storage with AWS Training and Certification

Resources created by the experts at AWS to help you build cloud storage skills



45+ free digital courses cover topics related to cloud storage, including:

- Amazon S3
- AWS Storage Gateway
- Amazon S3 Glacier
- Amazon Elastic File System (Amazon EFS)
- Amazon Elastic Block Store (Amazon EBS)

Classroom offerings, like Architecting on AWS, feature AWS expert instructors and hands-on activities

Visit aws.amazon.com/training/path-storage/

aws training and certification

Thank you!

Geert Jansen
gerardu@amazon.com

Darryl Osborne
darrylo@amazon.com

Please complete the session survey in the mobile app.