The basics of file systems

Neapolis
University
Pafos

JET
BRAINS

Today we are going to introduce NFS

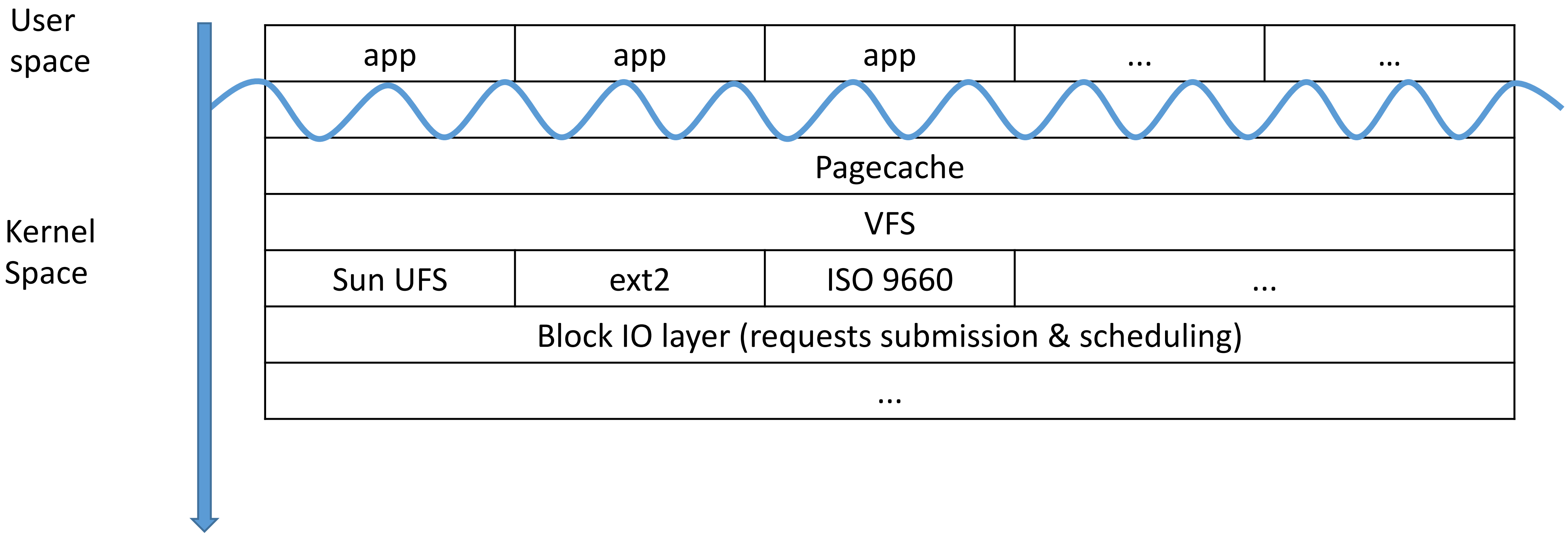
NFS (Network File System) is a mechanism to make a local FS accessible over the network.

Design requirements

- Accessing a FS over the network must work the same way as accessing a local FS, even from the point of view of the OS kernel.

Design requirements

- Accessing a FS over the network must work the same way as accessing a local FS, even from the point of view of the OS kernel.



User space

Kernel Space

app

app

app

...

...

Pagecache

VFS

Sun UFS

ext2

ISO 9660

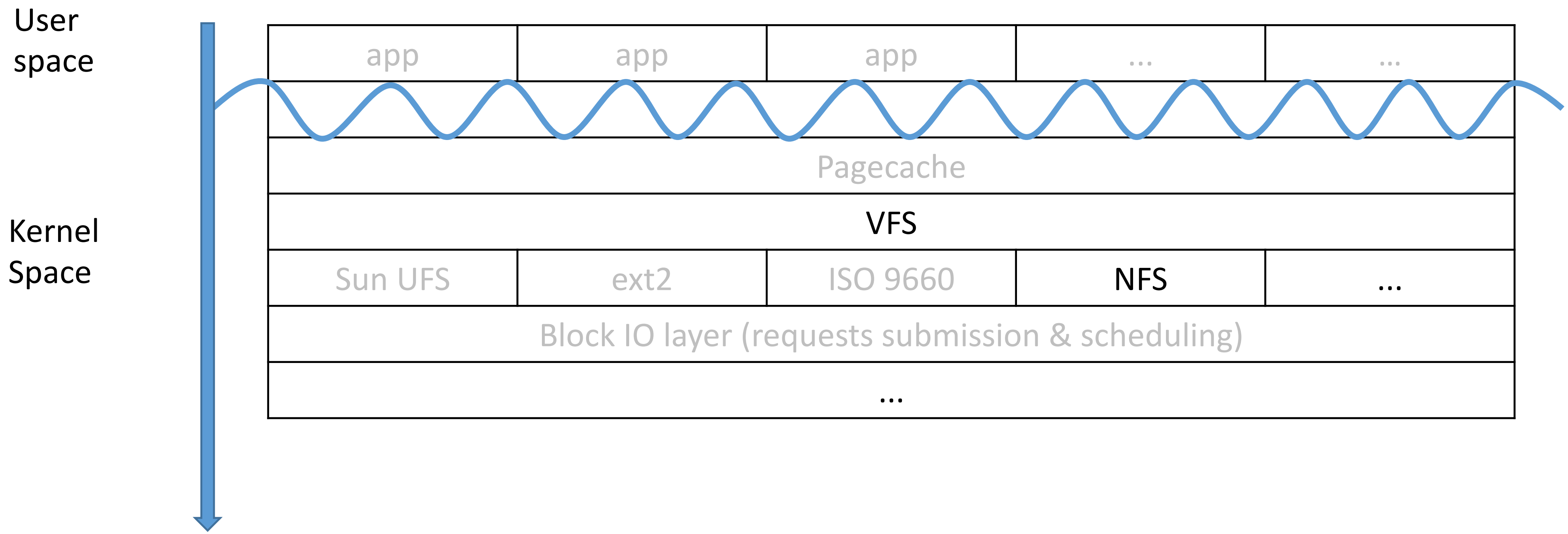
...

Block IO layer (requests submission & scheduling)

...

Design requirements

- Accessing a FS over the network must work the same way as accessing a local FS, even from the point of view of the OS kernel. **NFS must be accessed with the same VFS callbacks as a local file system**.



POSIX API and VFS callbacks

POSIX API	VFS
Calls that take a file path handle the whole path: • `open(“./dev/pstorage-fes/main.c”, O_RDONLY)`	

POSIX API and VFS callbacks

POSIX API	VFS
Calls that take a file path handle the whole path: - `open(“./dev/pstorage-fes/main.c”, O_RDONLY)`	The kernel walks the path one component at a time: - `t0 = openat(AT_FDCWD, “.”),` - `t1 = openat(t0, “dev”),` - `t2 = openat(t1, “pstorage-fes”),` - `t3 = openat(t2, “main.c”).` Why do we need this?

POSIX API and VFS callbacks

POSIX API	VFS
Calls that take a file path handle the whole path: • `open(“./dev/pstorage-fes/main.c”, O_RDONLY)`	The kernel walks the path one component at a time: • `t0 = openat(AT_FDCWD, “.”),` • `t1 = openat(t0, “dev”),` • `t2 = openat(t1, “pstorage-fes”),` • `t3 = openat(t2, “main.c”).` Why do we need this? At every step, the kernel needs to verify the type of the next component and handle the following cases: • a subdirectory, • a symbolic link, • a mount point.

POSIX API and VFS callbacks

POSIX API	VFS
Calls that take a file path handle the whole path: • `open(“./dev/pstorage-fes/main.c”, O_RDONLY)`	The kernel walks the path one component at a time: • `t0 = openat(AT_FDCWD, “.”),` • `t1 = openat(t0, “dev”),` • `t2 = openat(t1, “pstorage-fes”),` • `t3 = openat(t2, “main.c”).` Why do we need this? At every step, the kernel needs to verify the type of the next component and handle the following cases: • a subdirectory, • a symbolic link, • a mount point.

Quiz: what directory does a call `open(“./dev/pstorage-fes/..”)` open in the following two cases:

1. both “dev” and “pstorage-fes” are directories,
2. “dev” is a directory and “pstorage-fes” is a symbolic link?

POSIX API and VFS callbacks

Client

```
/
|- /home
|  |
|  |- /artem
|     |
|     |- hello.txt
|
|- /mnt <--- the mount point for /exports
```

Server

```
/
|- /home
|  |
|  |- /artem
|     |
|     |- hello.txt
|
|- /exports <--- mounted to /mnt on the client
   |
   |- /home
      |
      |- /artem
         |
         |- hello.txt
         |- symlink <-- points to “/home/artem/hello.txt”
```

Which of files named “`hello.txt`” does the following command read when run on the client machine:

```
# cat /mnt/home/artem/symlink
```

The protocol of NFSv2

- Walking a path:
  - `lookup(dirfh, name)          -> (fh, attr)`
  - `getattr(dirfh)               -> attr`
  - `readdir(dirfh, cookie, count) -> [direntry]`

Arguments named `fh` (File Handle) may be regarded as file descriptors.

`Attr` is similar to `struct stat`. It describes properties of a file or a directory like the type, the size, the access permissions, etc.

The protocol of NFSv2

- Walking a path:
  - `lookup(dirfh, name)                -> (fh, attr)`
  - `getattr(dirfh)                     -> attr`
  - `readdir(dirfh, cookie, count)      -> [direntry]`

- Working with directories and symlinks:
  - `mkdir(dirfh, name, attr)           -> (fh, newattr)`
  - `rmdir(dirfh, name)                 -> status`
  - `symlink(dirfh, name, string)       -> status`
  - `readlink(fh)                       -> string`
  - `link(dirfh, name, tofh, toname)    -> status`
  - `rename(dirfh, name, tofh, toname) -> status`

The protocol of NFSv2

- Walking a path:
  - `lookup(dirfh, name)                 -> (fh, attr)`
  - `getattr(dirfh)                      -> attr`
  - `readdir(dirfh, cookie, count)       -> [direntry]`

- Working with directories and symlinks:
  - `mkdir(dirfh, name, attr)            -> (fh, newattr)`
  - `rmdir(dirfh, name)                  -> status`
  - `symlink(dirfh, name, string)        -> status`
  - `readlink(fh)                        -> string`
  - `link(dirfh, name, tofh, toname)     -> status`
  - `rename(dirfh, name, tofh, toname)   -> status`

- Creating and removing files:
  - `create(dirfh, name, attr, flags)    -> (newfh, attr)`
  - `remove(dirfh, name)                 -> status`

The protocol of NFSv2

- Walking a path:
  - `lookup(dirfh, name)              -> (fh, attr)`
  - `getattr(dirfh)                   -> attr`
  - `readdir(dirfh, cookie, count)    -> [direntry]`

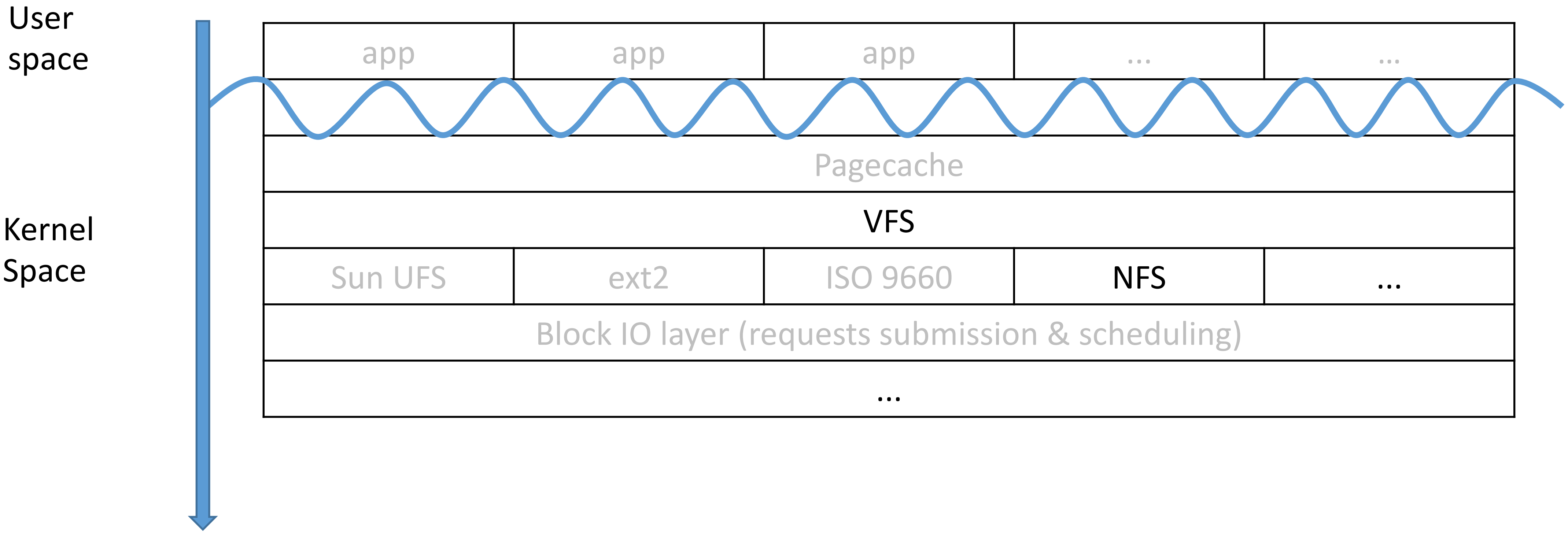
- Working with directories and symlinks:
  - `mkdir(dirfh, name, attr)         -> (fh, newattr)`
  - `rmdir(dirfh, name)               -> status`
  - `symlink(dirfh, name, string)     -> status`
  - `readlink(fh)                     -> string`
  - `link(dirfh, name, tofh, toname)  -> status`
  - `rename(dirfh, name, tofh, toname) -> status`

- Creating and removing files:
  - `create(dirfh, name, attr, flags) -> (newfh, attr)`
  - `remove(dirfh, name)              -> status`

- Reading and writing files:
  - `read(fh, offset, count)          -> (attr, data)`
  - `write(fh, offset, count, data)   -> attr`

Design requirements

- Accessing a FS over the network must work the same way as accessing a local FS, even from the point of view of the OS kernel. **NFS must be accessed with the same VFS callbacks as a local file system**.



Design requirements and networking-related issues

- Accessing a FS over the network must work the same way as accessing a local FS, even from the point of view of the OS kernel. **NFS must be accessed with the same VFS callbacks as a local file system**.
- An NFS server may reboot at any moment, but clients must be able to proceed with file handles that they already have.

- `t0 = openat(AT_FDCWD, “.”),`
- `t1 = openat(t0, “dev”),`
- `t2 = openat(t1, “pstorage-fes”),`
- `t3 = openat(t2, “main.c”).`

The server restarted here, but t2 must remain valid.

Design requirements and networking-related issues

- Accessing a FS over the network must work the same way as accessing a local FS, even from the point of view of the OS kernel. **NFS must be accessed with the same VFS callbacks as a local file system**.

- An NFS server may reboot at any moment, but clients must be able to proceed with file handles that they already have.
  File handles must persist across reboots of NFS servers.
  Many operations like `create()` may be confirmed by a server only after a `syncfs()`.

- `t0 = openat(AT_FDCWD, “.”),`
- `t1 = openat(t0, “dev”),`
- `t2 = openat(t1, “pstorage-fes”),`
- `t3 = openat(t2, “main.c”).`

The server restarted here, but t2 must remain valid.

Stateless server

File handles must persist across reboots of a NFS server.

One way to have the state persist across reboots is to have no state at all.

Stateless server

File handles must persist across reboots of a NFS server.

One way to have the state persist across reboots is to have no state at all.

In the Unix File System files can be accessed by their inode number. One can use inode numbers as file handles*.

** How do we deal with deleted files? In this scenario an inode may be reused for a different file.*

Stateless server

File handles must persist across reboots of a NFS server.

One way to have the state persist across reboots is to have no state at all.

In the Unix File System files can be accessed by their inode number. One can use inode numbers as file handles.

Problem: inodes are visible to the kernel only. There are no APIs in POSIX to open a file by its inode number. Is it possible to implement a NFS server in the user space?

Stateless server

File handles must persist across reboots of a NFS server.

One way to have the state persist across reboots is to have no state at all.

In the Unix File System files can be accessed by their inode number. One can use inode numbers as file handles.

Problem: inodes are visible to the kernel only. There are no APIs in POSIX to open a file by its inode number. Is it possible to implement a NFS server in the user space?

- `man 2 name_to_handle_at`
- `man 2 open_by_handle_at`

Stateless server

Many requests like `create()` may be confirmed only after a `syncfs()`.

It is ok-ish for infrequent requests. However, `write()`s need a `fsync()` which makes all of them synchronous. That is too slow.

Stateless server

Many requests like `create()` may be confirmed only after a `syncfs()`.

It is ok-ish for infrequent requests. However, `write()`s need a `fsync()` which makes all of them synchronous. That is too slow.

NFSv3 proposed a way to implement asynchronous writes:

- Upon startup, a NFS server generates a random number.
- Responses to `write()`s communicate this number to the client.
- When the client wants to persist the data, it calls `commit()` and passes a number received from `write()`s. By looking at this number the NFS server can understand whether it was restarted between `write()`s and `commit()`, or not. If it was, the client must resend all `write()`s.

Stateless server and issues with POSIX FS behaviours

There may be files without a name. They exist as long as they have an open file descriptor.

Stateless server and issues with POSIX FS behaviours

There may be files without a name. They exist as long as they have an open file descriptor.

Problem: if a NFS server exits, its open file descriptors are closed, and all unnamed files are deleted automatically. Unnamed files cannot persist across NFS server restarts.

Stateless server and issues with POSIX FS behaviours

There may be files without a name. They exist as long as they have an open file descriptor.

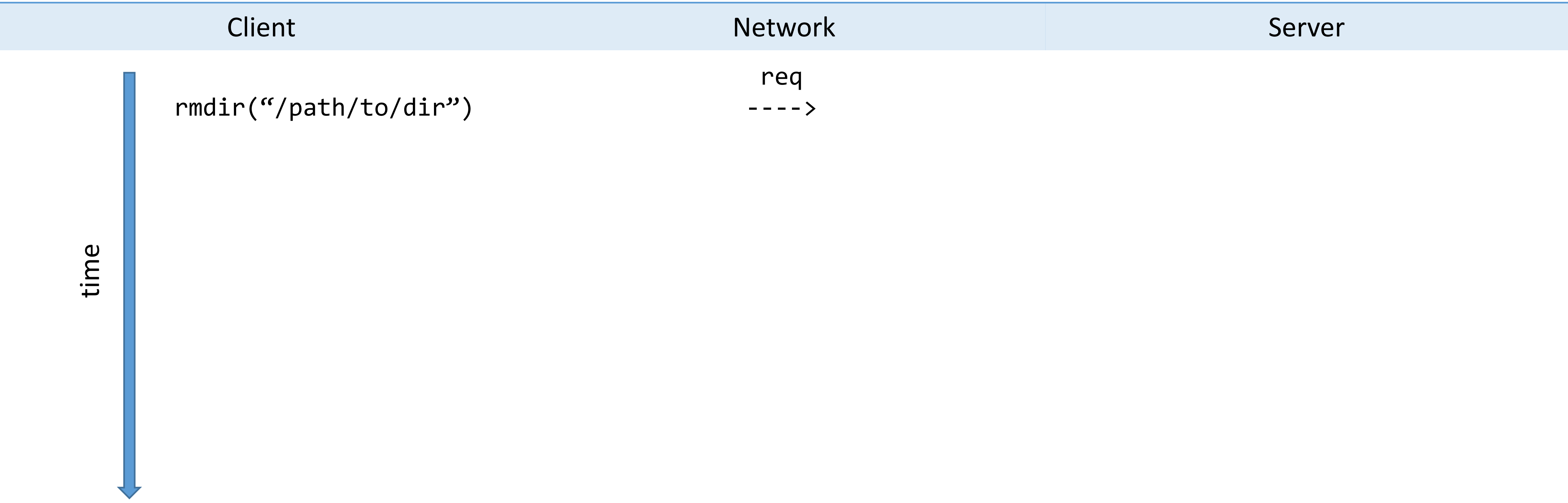
Problem: if a NFS server exits, its open file descriptors are closed, and all unnamed files are deleted automatically. Unnamed files cannot persist across NFS server restarts.

This issue is solved by NFS clients (silly rename):

- when an open file is unlink()ed, a NFS client renames that file to “.nfs_random_suffix”,
- the client removes those files when they no longer have open file descriptors.

If a client crashes or abruptly disconnects from the network, these files remain as garbage.

Issues related to non-idempotency of requests



Client

Network

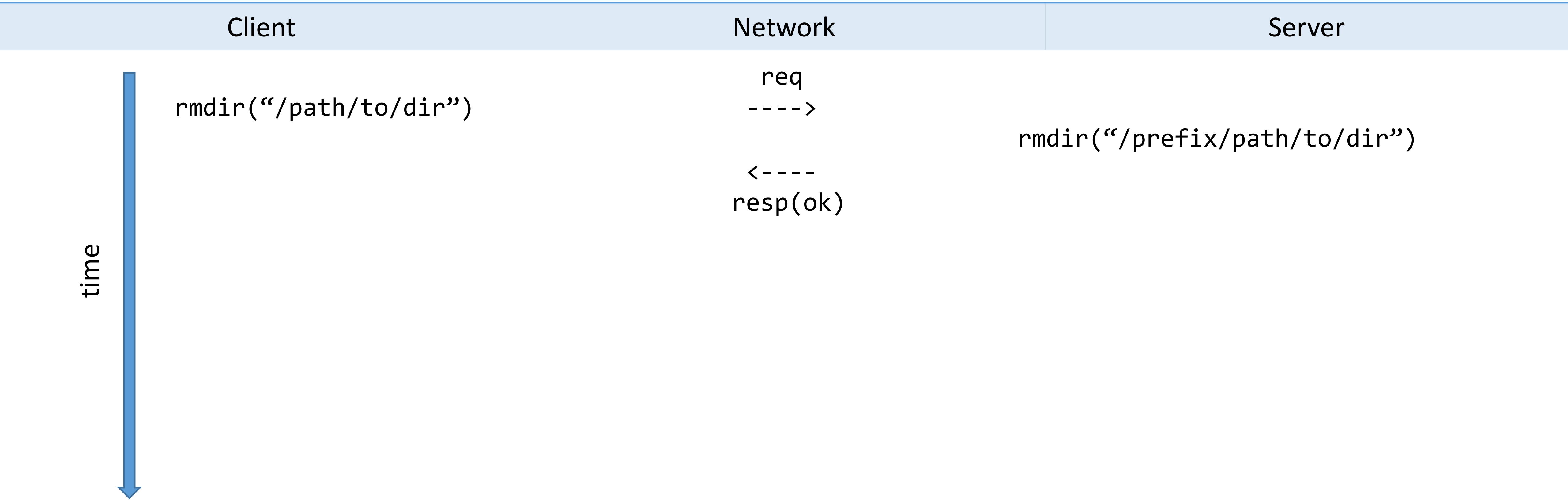
Server

time

```
rmdir("/path/to/dir")
```

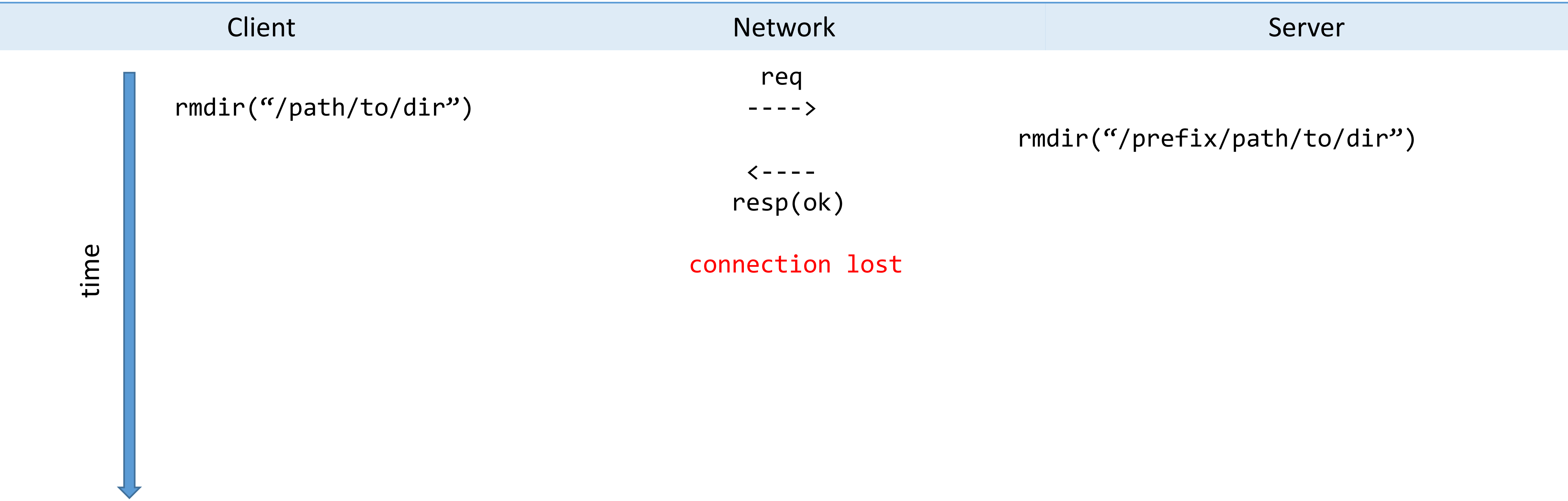
```
req
---->
```

Issues related to non-idempotency of requests



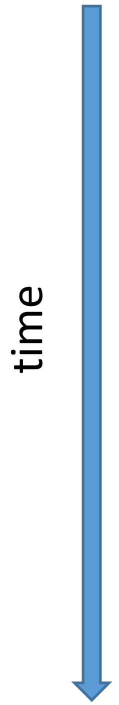
Client	Network	Server
rmdir(“/path/to/dir”)	req ---->	
		rmdir(“/prefix/path/to/dir”)
	<---- resp(ok)	

Issues related to non-idempotency of requests



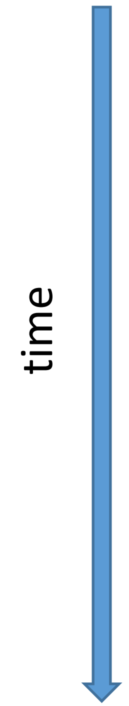
Client	Network	Server
	req ---->	
rmdir(“/path/to/dir”)		
		rmdir(“/prefix/path/to/dir”)
	<---- resp(ok)	
	connection lost	

Issues related to non-idempotency of requests



Client	Network	Server
rmdir(“/path/to/dir”)	req ---->	
		rmdir(“/prefix/path/to/dir”)
	<---- resp(ok)	
	connection lost	
timeout		

Issues related to non-idempotency of requests



Client	Network	Server
rmdir(“/path/to/dir”)	req ---->	
		rmdir(“/prefix/path/to/dir”)
	<---- resp(ok)	
	connection lost	
timeout		
retry rmdir(“/path/to/dir”)	---->	
		rmdir(“/prefix/path/to/dir”)

Issues related to non-idempotency of requests

time

Client	Network	Server
rmdir(“/path/to/dir”)	req ---->	
		rmdir(“/prefix/path/to/dir”)
	<---- resp(ok)	
	connection lost	
timeout		
retry rmdir(“/path/to/dir”)	---->	
		rmdir(“/prefix/path/to/dir”)
	<---- resp(not found)	

More networking-related issues

How much time does a `lookup()` take locally and in NFS?

More networking-related issues

How much time does a `lookup()` take locally and in NFS?

`lookup()` in NFS cannot take less time than 1 RTT between the client and the server.

More networking-related issues

How much time does a `lookup()` take locally and in NFS?

`lookup()` in NFS cannot take less time than 1 RTT between the client and the server.

Figures for reference:

- reading 50KB from a 2ch DDR4 3200 RAM takes ≈1μs,
- RTT between machines connected to the same 10Gbps switch is ≈50μs,
- RTT between VMs in the same GCP zone is ≈500μs,
- RTT between Limassol and Warsaw is ≈80ms.

More networking-related issues

How much time does a `lookup()` take locally and in NFS?

`lookup()` in NFS cannot take less time than 1 RTT between the client and the server.

Figures for reference:

- reading 50KB from a 2ch DDR4 3200 RAM takes ≈1μs,
- RTT between machines connected to the same 10Gbps switch is ≈50μs,
- RTT between VMs in the same GCP zone is ≈500μs,
- RTT between Limassol and Warsaw is ≈80ms.

Reminder: what are the reasons to have `openat()` even though `open()` is already available?

More networking-related issues

How much time does a `lookup()` take locally and in NFS?

`lookup()` in NFS cannot take less time than 1 RTT between the client and the server.

Figures for reference:

- reading 50KB from a 2ch DDR4 3200 RAM takes ≈1μs,
- RTT between machines connected to the same 10Gbps switch is ≈50μs,
- RTT between VMs in the same GCP zone is ≈500μs,
- RTT between Limassol and Warsaw is ≈80ms.

sunrpc portmap service uses UDP and a response to a request may be much longer than the request itself.

More networking-related issues

How much time does a `lookup()` take locally and in NFS?	`lookup()` in NFS cannot take less time than 1 RTT between the client and the server. Figures for reference: • reading 50KB from a 2ch DDR4 3200 RAM takes ≈1μs, • RTT between machines connected to the same 10Gbps switch is ≈50μs, • RTT between VMs in the same GCP zone is ≈500μs, • RTT between Limassol and Warsaw is ≈80ms.
sunrpc portmap service uses UDP and a response to a request may be much longer than the request itself.	This enabled “traffic amplification” attacks.

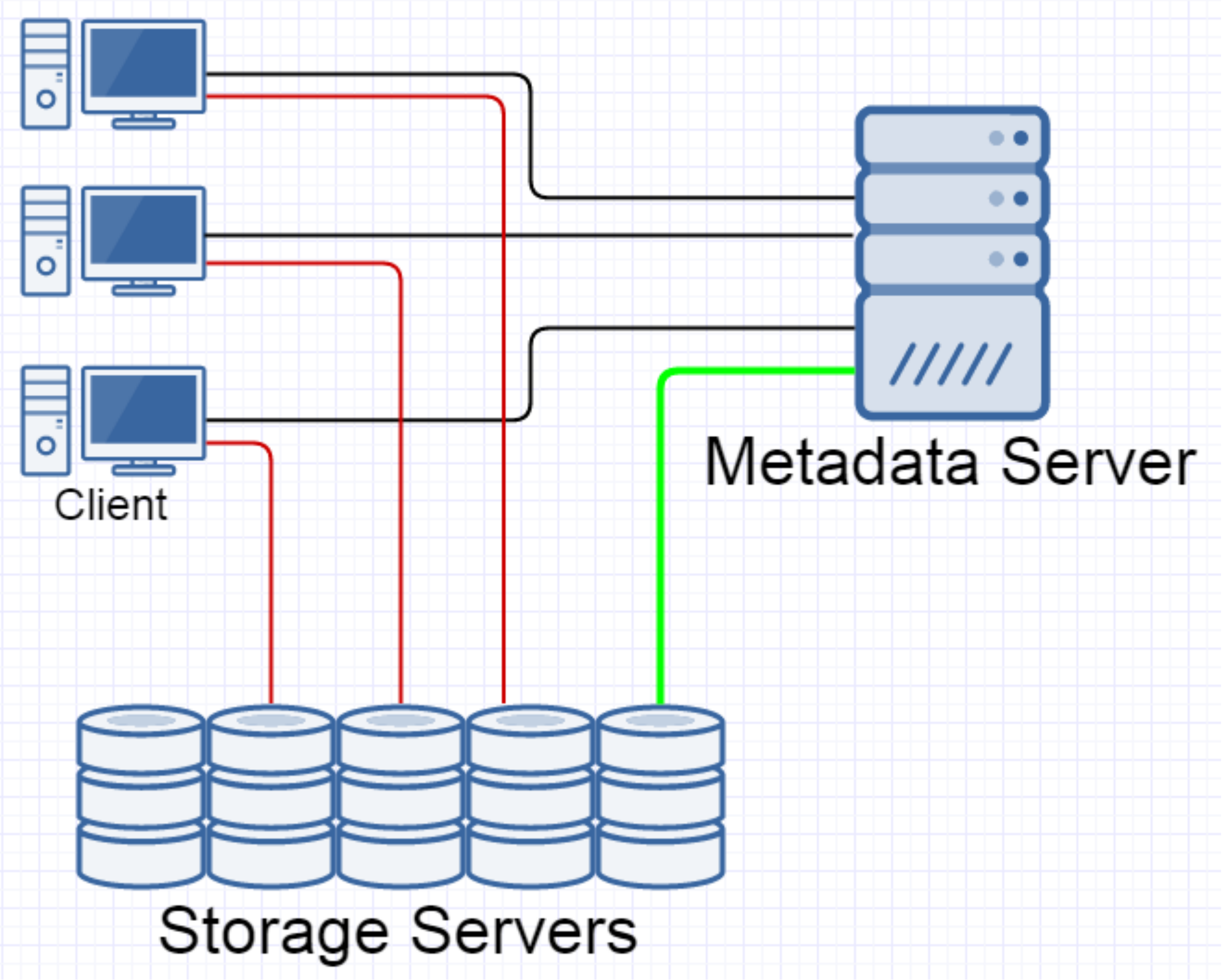
Single point of failure

A NFS server is both a single point of failure, and a bottleneck because its resources need to be shared between multiple clients.

Single point of failure

A NFS server is both a single point of failure, and a bottleneck because its resources need to be shared between multiple clients.

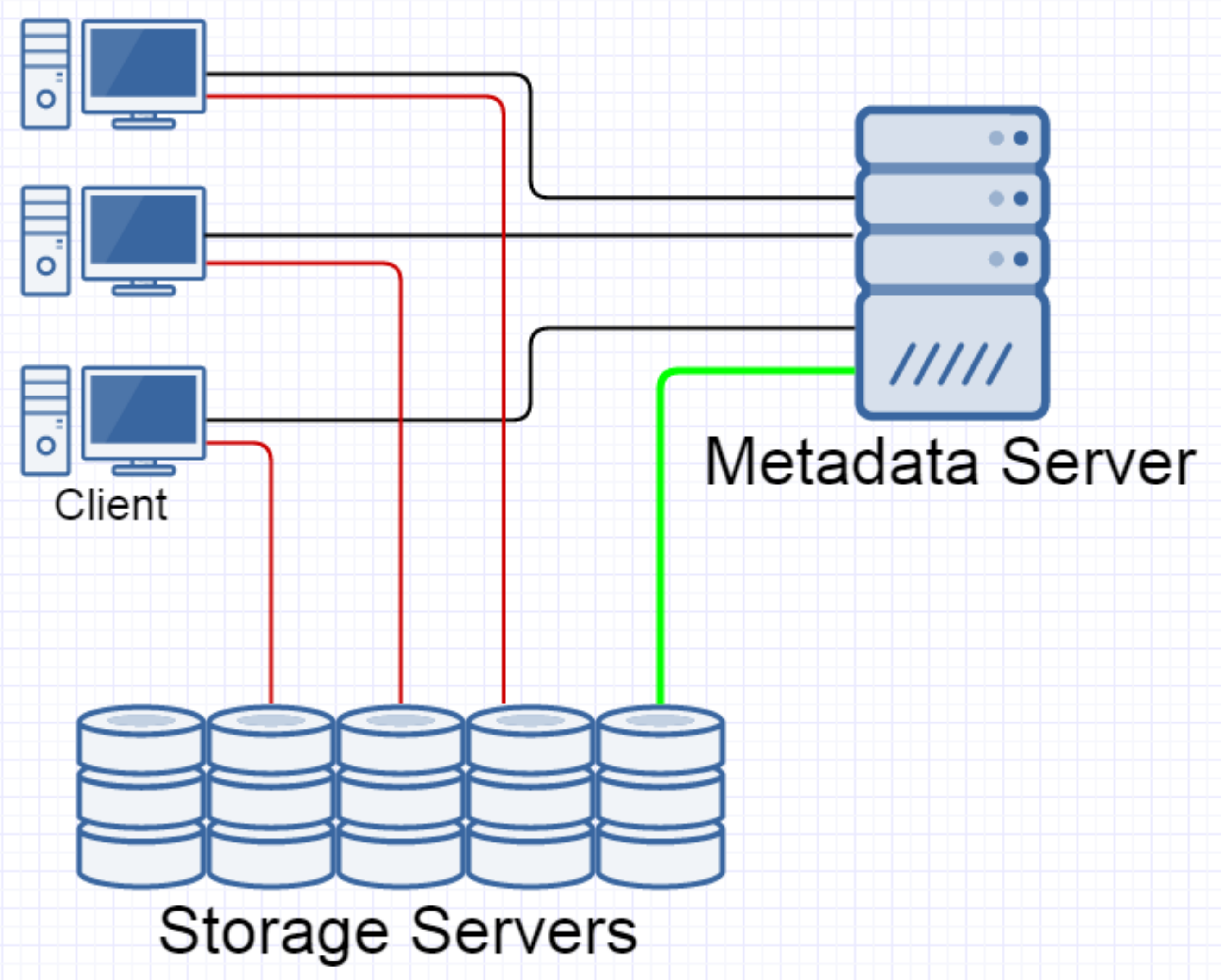
Parallel NFS в NFSv4.1:



Single point of failure

A NFS server is both a single point of failure, and a bottleneck because its resources need to be shared between multiple clients.

Parallel NFS в NFSv4.1:



It introduces interesting failure modes of its own:

- What happens if a client loses a connection to the metadata server, but retains connections to storage servers?

The Fallacies of Networked Computing

1. The network delivers messages reliably.
2. Messages are delivered instantly.
3. The bandwidth is unlimited.
4. The network is secure.

The Fallacies of Networked Computing

1. The network delivers messages reliably.
2. Messages are delivered instantly.
3. The bandwidth is unlimited.
4. The network is secure.

A function in the same process may succeed or fail. The caller does not care about that function crashing because in that case the caller is killed, too.

The network introduces a new failure mode: a function call did something, but the caller was not notified about that.

Network protocols must account for requests never being delivered and being delivered multiple times due to retries.

A broken connection is a norm, not an exception.

The Fallacies of Networked Computing

1. The network delivers messages reliably.
2. Messages are delivered instantly.
3. The bandwidth is unlimited.
4. The network is secure.

The speeds of function calls and RPC calls are qualitatively different.

Spilling some registers to the stack and jumping to a function is measured with 1 or 2 digits of nanoseconds. An RPC call in a 10Gbps ethernet network spends dozens of microseconds on the network RTT alone. As we move from local networks to the Internet, that becomes milliseconds.

The Fallacies of Networked Computing

Consider a naïve routine to copy a file:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r0 = write(fd_out, buf, r);
  ...
}
```

```
21-02-18 23:40:38.936 s#141270                                     0x4c44d78350, length = 16}
21-02-18 23:40:39.191 s#14127                                     4c44d78350
21-02-18 23:40:39.191 s#14127

21-02-18 23:40:39.757 s#14127                                      0x4c44d78360, length = 944}
21-02-18 23:40:39.757 s#14127                                     x4c44d78360
21-02-18 23:40:39.757 s#14127

21-02-18 23:40:40.242 s#14127                                      0x4c44d7e360, length = 16}
21-02-18 23:40:40.361 s#14127                                     4c44d7e360
21-02-18 23:40:40.361 s#141270.
```

It took approx. 1.4 seconds to download 976 bytes.

This is not a log of a transatlantic communication. Both the client and the server were in the UK.

The Fallacies of Networked Computing

1. The network delivers messages reliably.
2. Messages are delivered instantly.
3. The bandwidth is unlimited.
4. The network is secure.

Returning huge objects from a function call is cheap. It suffices to return 1 pointer.

A RPC call must transfer the whole response, and the speed of a 10Gbps network connection is much smaller than that of the RAM.

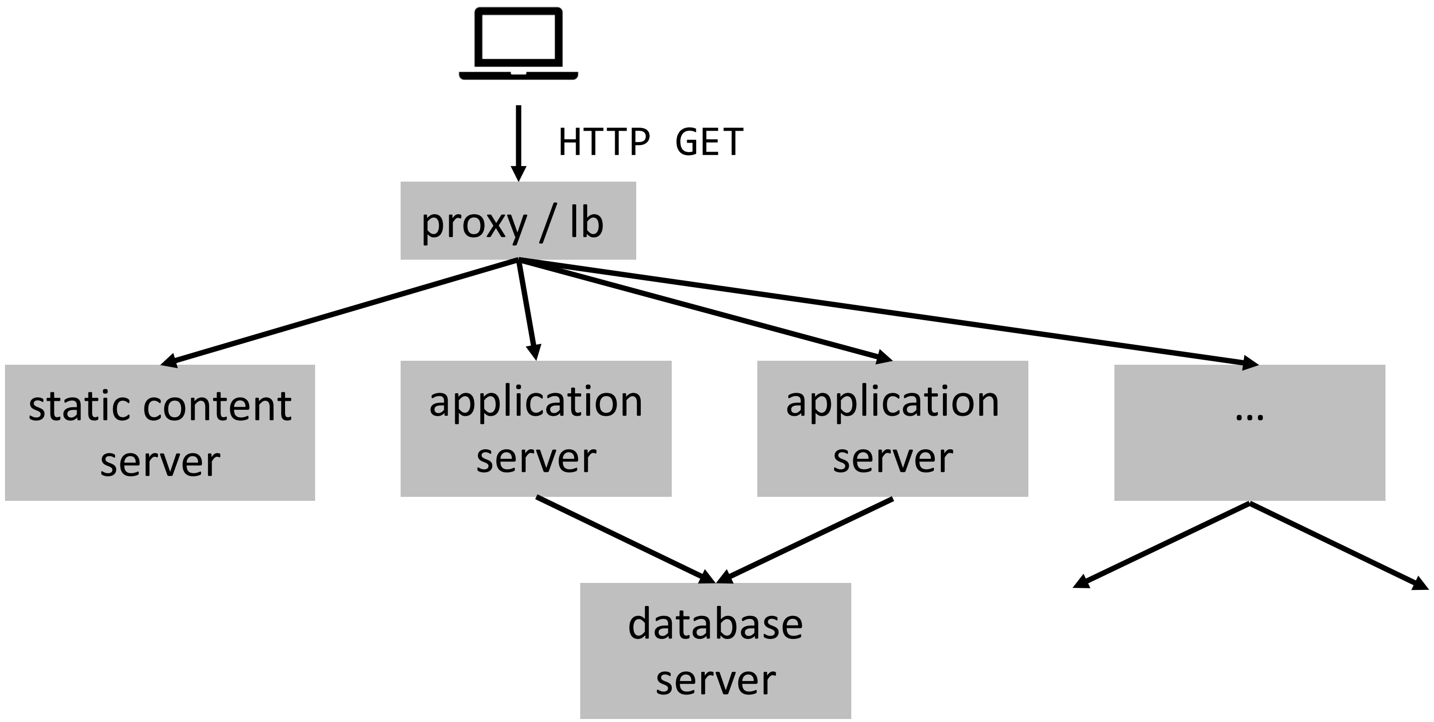
In many cases one must also factor in the CPU time needed to serialise and deserialise JSONs, protobufs, XMLs and other transport formats.

More networking-related issues

Typically, a server divides a user request into smaller sub-requests and sends them to servers that are deeper down the system:

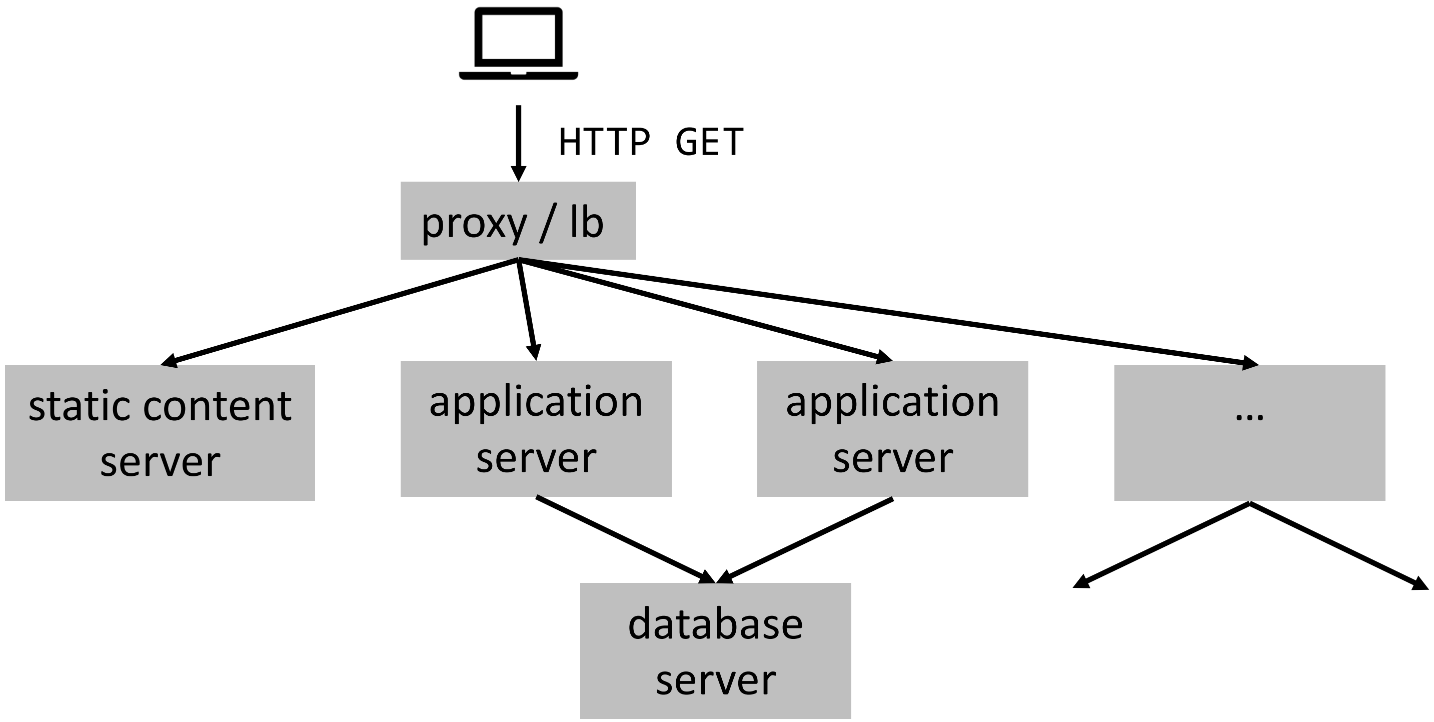
More networking-related issues

Typically, a server divides a user request into smaller sub-requests and sends them to servers that are deeper down the system:



More networking-related issues

Typically, a server divides a user request into smaller sub-requests and sends them to servers that are deeper down the system:

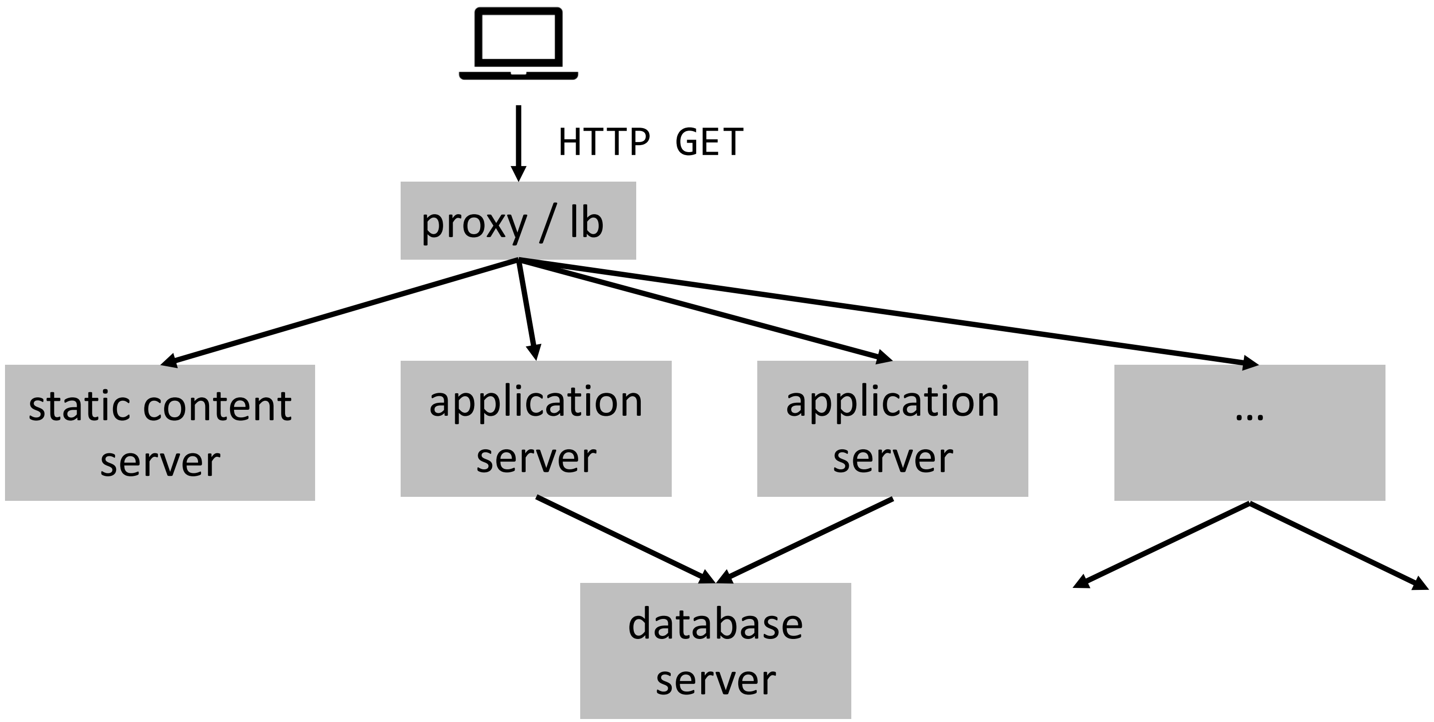


1. Suppose that every request spawns 10 sub-requests. Suppose also that the 99th pct. of the sub-request latency is 1ms, but the rest of sub-requests take 1s. What is the distribution of latencies of top-level requests?

   How to bound the tail latencies of top-level requests?

More networking-related issues

Typically, a server divides a user request into smaller sub-requests and sends them to servers that are deeper down the system:



1. Suppose that every request spawns 10 sub-requests. Suppose also that the 99th pct. of the sub-request latency is 1ms, but the rest of sub-requests take 1s. What is the distribution of latencies of top-level requests?

   How to bound the tail latencies of top-level requests?

2. How collect logs of all sub-requests together? How to analyse the execution trace of a top-level request? How to find reasons for requests getting blocked?

Logging in a distributed system and the birthdays problem

How to match logs from multiple services in a distributed system?

Logging in a distributed system and the birthdays problem

How to match logs from multiple services in a distributed system?

Let us assign a unique ID to each top-level request and propagate it into every sub-request. Now every log message related to a sub-request contains the ID of the top-level request. One can now scan logs of all services and filter those that contain a needed top-level request ID.

Logging in a distributed system and the birthdays problem

How to match logs from multiple services in a distributed system?

Let us assign a unique ID to each top-level request and propagate it into every sub-request. Now every log message related to a sub-request contains the ID of the top-level request. One can now scan logs of all services and filter those that contain a needed top-level request ID.

How can we generate globally unique IDs? We must do this without incurring any synchronisation between nodes of our system.

Logging in a distributed system and the birthdays problem

How to match logs from multiple services in a distributed system?

Let us assign a unique ID to each top-level request and propagate it into every sub-request. Now every log message related to a sub-request contains the ID of the top-level request. One can now scan logs of all services and filter those that contain a needed top-level request ID.

How can we generate globally unique IDs? We must do this without incurring any synchronisation between nodes of our system.

Can choose globally unique IDs randomly?

Logging in a distributed system and the birthdays problem

How to match logs from multiple services in a distributed system?

Let us assign a unique ID to each top-level request and propagate it into every sub-request. Now every log message related to a sub-request contains the ID of the top-level request. One can now scan logs of all services and filter those that contain a needed top-level request ID.

How can we generate globally unique IDs? We must do this without incurring any synchronisation between nodes of our system.

Can choose globally unique IDs randomly?

The birthdays problem: let us have n samples of a random integer value that is uniformly distributed in the interval [1, d]. What is the probability $p(n, d)$ of at least 2 samples being equal?

Logging in a distributed system and the birthdays problem

How to match logs from multiple services in a distributed system?

Let us assign a unique ID to each top-level request and propagate it into every sub-request. Now every log message related to a sub-request contains the ID of the top-level request. One can now scan logs of all services and filter those that contain a needed top-level request ID.

How can we generate globally unique IDs? We must do this without incurring any synchronisation between nodes of our system.

Can choose globally unique IDs randomly?

The birthdays problem: let us have n samples of a random integer value that is uniformly distributed in the interval [1, d]. What is the probability $p(n, d)$ of at least 2 samples being equal?

The answer is:

$$p(n,d) = f(x) = \begin{cases} 1 - \prod_{k=1}^{n-1} \left(1 - \frac{k}{d}\right), & n \le d \\ 1, & n > d \end{cases}$$

When $n \ll d$ we get

$$p(n,d) \approx 1 - \exp\left(-\frac{n(n-1)}{2d}\right)$$

Logging in a distributed system and the birthdays problem

How to match logs from multiple services in a distributed system?

Let us assign a unique ID to each top-level request and propagate it into every sub-request. Now every log message related to a sub-request contains the ID of the top-level request. One can now scan logs of all services and filter those that contain a needed top-level request ID.

How can we generate globally unique IDs? We must do this without incurring any synchronisation between nodes of our system.

Can choose globally unique IDs randomly?

If we generate 2^{32} random IDs that are 128 bits long then the probability to have a duplicate ID is

$$p(2^{32}, 2^{128}) \approx 1 - \exp\left(-\frac{2^{32} \cdot 2^{32}}{2 \cdot 2^{128}}\right) =$$

$$= 1 - \exp(-2^{-65}) \approx 2^{-65}$$

The birthdays problem: let us have n samples of a random integer value that is uniformly distributed in the interval [1, d]. What is the probability $p(n, d)$ of at least 2 samples being equal?

The answer is:

$$p(n,d) = f(x) = \begin{cases} 1 - \prod_{k=1}^{n-1}\left(1 - \frac{k}{d}\right), & n \leq d \\ 1, & n > d \end{cases}$$

When $n \ll d$ we get

$$p(n,d) \approx 1 - \exp\left(-\frac{n(n-1)}{2d}\right)$$

Logging in a distributed system and the birthdays problem

How to match logs from multiple services in a distributed system?

Let us assign a unique ID to each top-level request and propagate it into every sub-request. Now every log message related to a sub-request contains the ID of the top-level request. One can now scan logs of all services and filter those that contain a needed top-level request ID.

How can we generate globally unique IDs? We must do this without incurring any synchronisation between nodes of our system.

Can choose globally unique IDs randomly?

Another useful trick is to have IDs that have 6 bytes of the timestamp and 10 random bytes. They are equally unlikely to have collisions, but they are also sortable and include the timestamp.

Reminder: UUIDv4 vs. UUIDv7.

To do at home

1. Write a gRPC server (https://grpc.io) with a single request handler that generates 128 random bytes. Add random sleeps so that 99% of requests run very fast, and 1% of requests take 1s.
2. Write a gRPC server with a single request handler that does 16 sub-requests to different instances of a service from exercise #1 and concatenates their responses. Sub-requests must be processed in parallel.
3. Install Prometheus (https://prometheus.io). Instrument solutions of #1 and #2 to report request latencies to Prometheus. Install Grafana (https://grafana.com) and make a dashboard with plots of the 90th, 95th and 99th percentiles of latencies of service №2 and each of 16 instances of service #1.
4. Install Jaeger (https://www.jaegertracing.io). Instrument solutions of #1 and #2 with opentracing (https://opentracing.io) have them report request traces to Jaeger.