Distributed Storage Systems

Remote Procedure Calls (RPC)
Network Filesystem (NFS)

Agenda

Distributed file systems (a first view)

Today’s topics

- Remote Procedure Calls (RPC)
  - Basics
- Introduction to Networked File System (NFS)
- Python examples: Werkzeug, JSON-RPC

Agenda

Distributed file systems (a first view)

Today's topics

- Remote Procedure Calls (RPC)
  - Basics
- Introduction to Networked File System (NFS)
- Python examples: Werkzeug, JSON-RPC

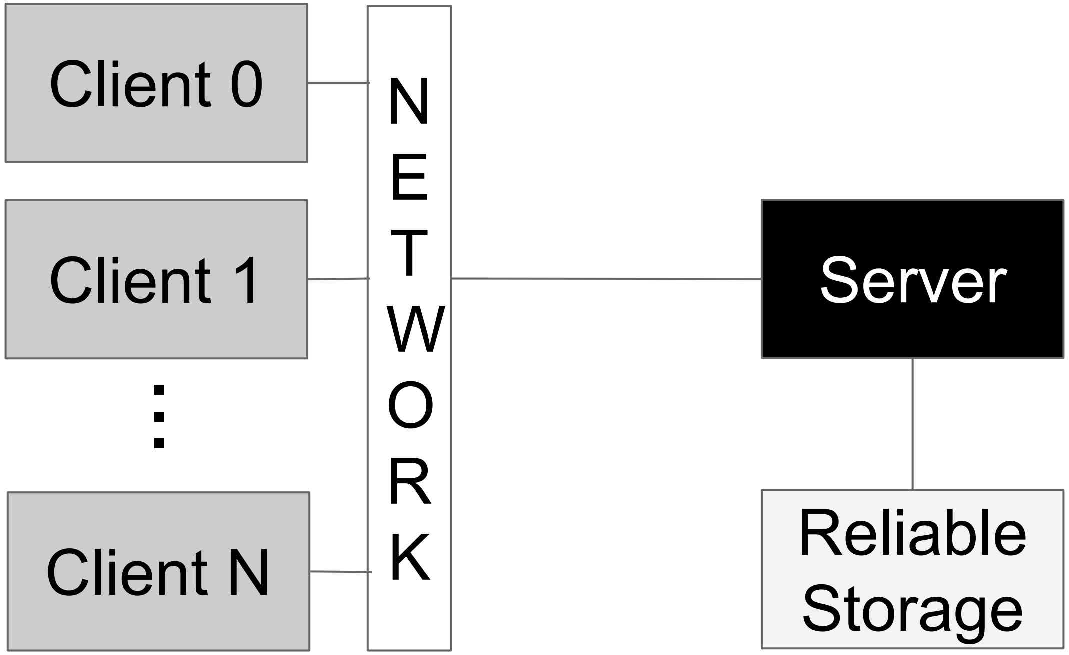
A bit of a chicken and egg problem if you have not heard of RPC

Goals

This week's Learning Goals

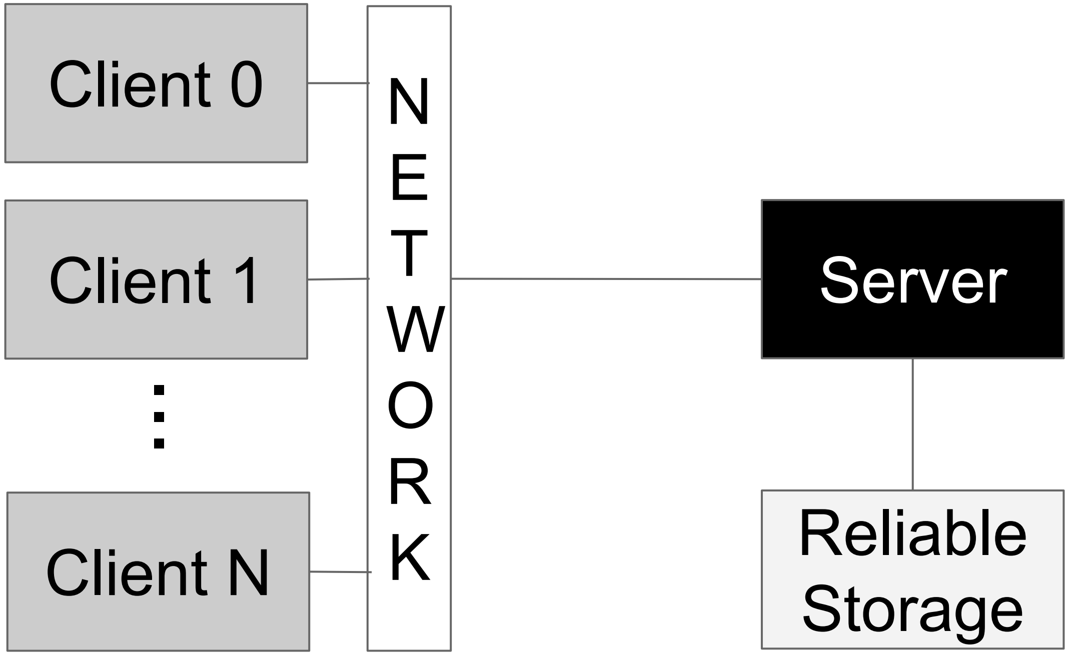
- Understand basics of RPC
- Understand NFS, which is RPC based
- Be able to create a small client application to create a file in a remote machine (server app) and store data to it using RPC

A Basic Distributed File System



- The server has the disks
- Clients send messages across a network to access their directories and files on those disks

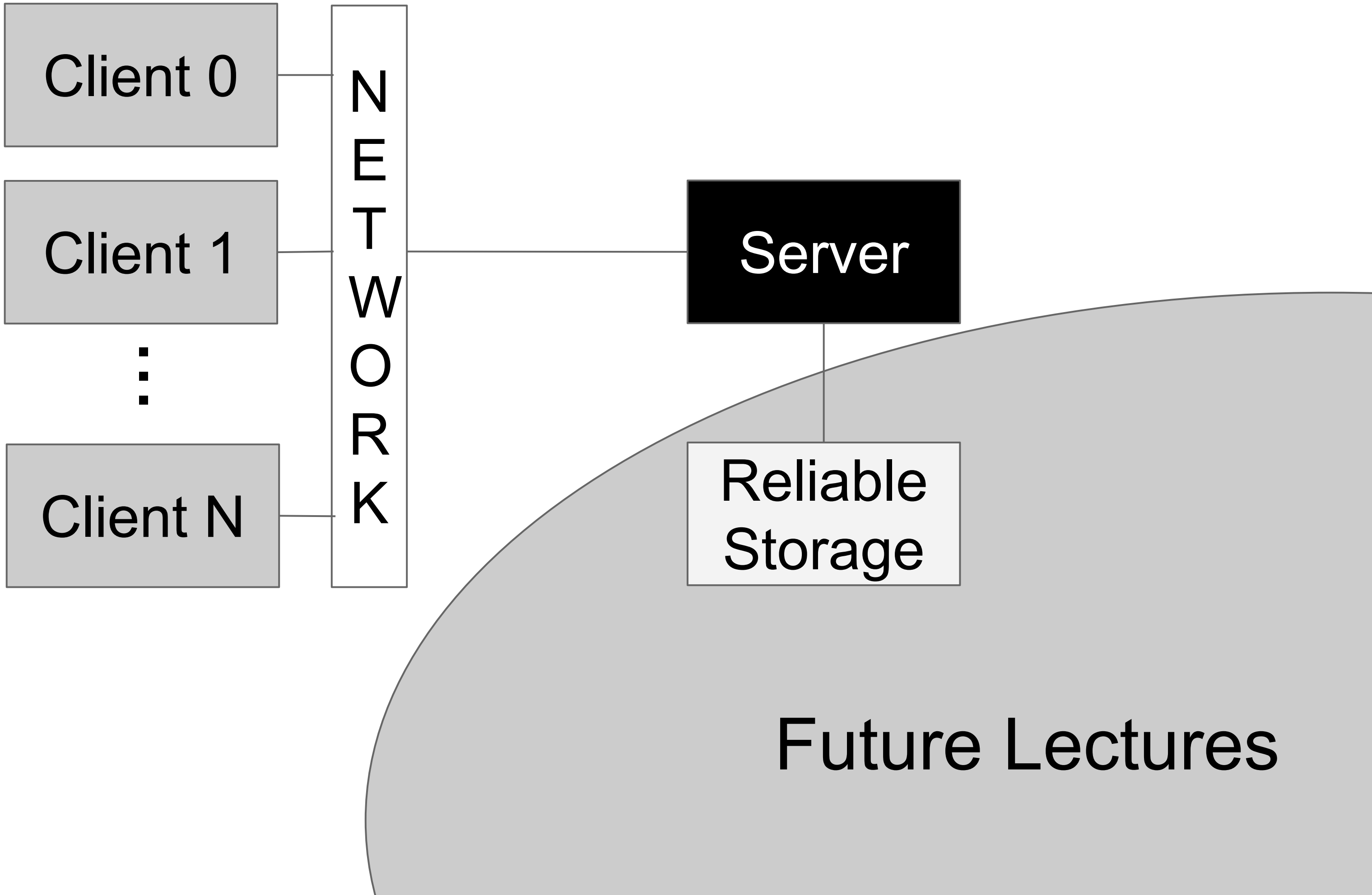
A Basic Distributed File System



Benefits:

- This setup allows for easy sharing of data across clients
- Centralized administration, e.g., to back up files.
- Security

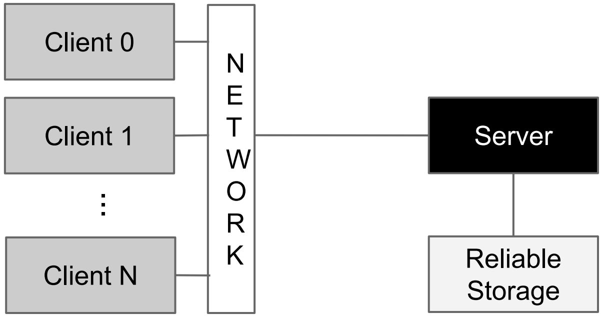
A Basic Distributed File System



A Basic Distributed File System

- There are two important pieces of software:
  - The client-side file system
  - The file server

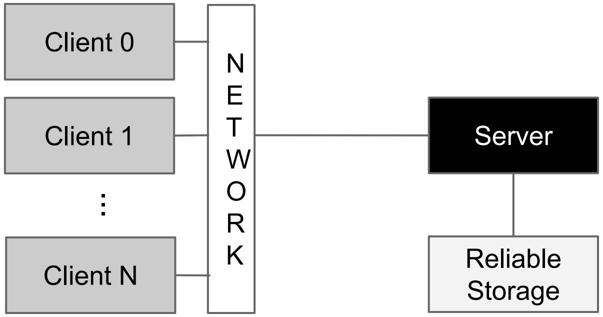
- A client application issues system calls to the client-side file system:
  - open(), close(), read(), write(), mkdir(), …
  - For example, a **read()**



A Basic Distributed File System

- The client-side file system sends a message to the server-side file system (file server) to **read** a particular block

- The file server will then read the block from disk (or its own in-memory cache), and send a message back to the client with the requested data



NFS

- One of the earliest
- Developed by Sun as an **open protocol**
  - Different groups could develop their own implementations and compete
- There has been an evolution of the protocol
  - NFSv2 (1989 - classical, but obsolete)
  - NFSv3 (1995 - most popular)
  - NFSv4 (2003 - has been slowly replacing v3)
  - NFSv4.1 (2010 - early adopters)

NFS

- One of the earliest
- Developed by Sun as an open protocol
  - Different groups could develop their own implementations and compete
- There has been an evolution of the protocol
  - **NFSv2 (1989 - classical, but obsolete)**
  - NFSv3 (1995 - most popular)
  - NFSv4 (2003 - has been slowly replacing v3)
  - NFSv4.1 (2010 - early adopters)

We will use NFSv2 as a case example

NFS Design Goals

- OS independence & interoperability
- Simple crash recovery for clients and servers
- Transparent access (client programs do not know files are remote)
- Maintain local file system semantics
- Reasonable performance

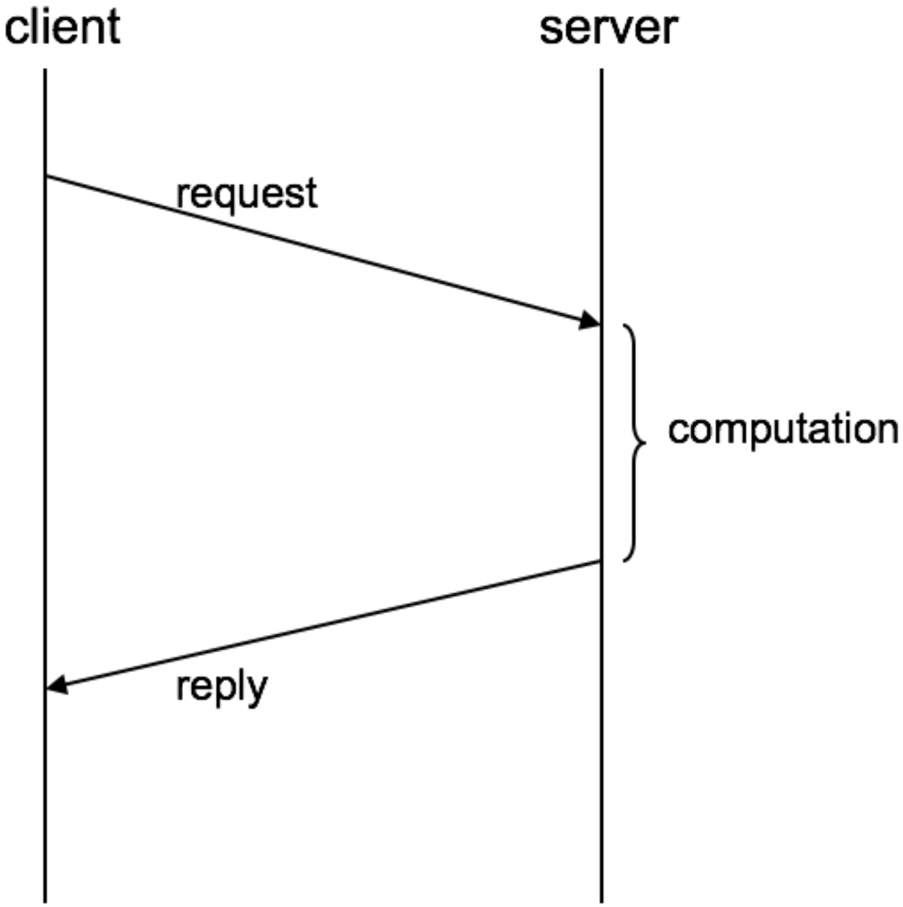
Slide adapted from Avishay Traeger - An Introduction to NFS

RPC (Remote Procedure Call)

- NFS is defined as a set of RPCs – their arguments, results, and effects
- RPCs are synchronous
- The use of RPCs makes the protocol easier to understand

Slide adapted from Avishay Traeger - An Introduction to NFS

RPC



- Request/Reply mechanism
- Procedure call – disjoint address space

Slide adapted from Swati Agarwal

Why RPC?

- Function Oriented Protocols (e.g., Telnet, FTP) cannot perform “execute function Y with arguments X1, X2 on machine Z”

- Formalize a protocol that allows for a simple solution to this question (J. E. White in 1976)

Slide adapted from Swati Agarwal

Challenges

- Binding
- Communication protocol
- Dealing with failures – network / server crash
- Addressable arguments
- Integration with existing systems
- Data Integrity and security

Slide adapted from Swati Agarwal

Performance

- Mainly RPC overhead – not due to local call
- For small packets, RPC overhead dominates
- For large packets, transmission time dominates
- High data rate achieved by interleaving parallel remote calls from multiple processes
- Exporting / Importing cost unmeasured

Slide adapted from Swati Agarwal

Stateless Protocol (NFSv2)

- The server does not keep state for RPCs
- Each RPC contains the necessary information to complete the call
- This makes crash recovery easy
  - Server crash: server does no crash recovery, clients resubmit requests
  - Client crash: no crash recovery for client or server

“Stateless” Protocol (NFSv2)

- The server does not keep state for RPCs
- Each RPC contains the necessary information to complete the call
- This makes crash recovery easy
  - Server crash: server does no crash recovery, clients resubmit requests
  - Client crash: no crash recovery for client or server
- Nice in theory, but:
  - Not really stateless: File locking adds state, provided by separate protocol & daemon
  - Server keeps an RPC reply cache to handle some duplicate RPCs

Slide adapted from Avishay Traeger - An Introduction to NFS

“Stateless” Protocol (NFSv2)

- How can we define the network protocol to enable stateless operation?
  - Clearly, stateful calls like open() can’t be a part of the discussion (as it would require the server to track open files)
  - However, the client application will want to call open(), read(), write(), close() and other standard API calls to access files and directories
- Key are file handles

File Handle

- File handle (fh, fhandle) is a structure common in NFS
- Provided by the server and used by the client to reference a file
- New fhandles returned by LOOKUP, CREATE, MKDIR, …
- The fhandle for the root of the file system is obtained by the client when it **mounts** the file system

NFS Protocol - File Operations
(not complete)

NFSPROC_GETATTR
expects: file handle
returns: attributes

NFSPROC_SETATTR
expects: file handle, attributes
returns: nothing

NFSPROC_LOOKUP
expects: directory file handle, name of file/directory to look up
returns: file handle

NFSPROC_READ
expects: file handle, offset, count
returns: data, attributes

NFSPROC_WRITE
expects: file handle, offset, count, data
returns: attributes

NFSPROC_CREATE
expects: directory file handle, name of file, attributes
returns: nothing

NFSPROC_REMOVE
expects: directory file handle, name of file to be removed
returns: nothing

NFS Protocol - File Operations
(not complete)

```
NFSPROC_GETATTR
expects: file handle
returns: attributes
```

Return file attributes

```
NFSPROC_SETATTR
expects: file handle, attributes
returns: nothing
```

Set file attributes

```
NFSPROC_LOOKUP
expects: directory file handle, name of file/directory t
returns: file handle
```

Look file in directory

```
NFSPROC_READ
expects: file handle, offset, count
returns: data, attributes
```

Reads up to count **bytes with** offset **bytes from file**

```
NFSPROC_WRITE
expects: file handle, offset, count, data
returns: attributes
```

Writes count **bytes with** offset **bytes to file**

```
NFSPROC_CREATE
expects: directory file handle, name of file, attributes
returns: nothing
```

Creates new file name

```
NFSPROC_REMOVE
expects: directory file handle, name of file to be remov
returns: nothing
```

Removes file name

NFS Protocol - Directory Operations
(not complete)

NFSPROC_MKDIR
expects: directory file handle, name of directory, attributes
returns: file handle

NFSPROC_RMDIR
expects: directory file handle, name of directory to be removed
returns: nothing

NFSPROC_READDIR
expects: directory handle, count of bytes to read, cookie
returns: directory entries, cookie (to get more entries)

NFS Protocol - Directory Operations

(not complete)

```
NFSPROC_MKDIR
expects: directory file handle, name of direct
returns: file handle
```

Create new directory name **in a directory**

```
NFSPROC_RMDIR
expects: directory file handle, name of direct
returns: nothing
```

Remove directory

```
NFSPROC_READDIR
expects: directory handle, count of bytes to r
returns: directory entries, cookie (to get mor
```

Returns up to count **bytes of directory entries from the directory.**
A cookie **is used in subsequent readdir calls to start reading at a specific entry in the directory.**
A cookie **of zero returns entries starting with the first entry in the directory**

NFS Protocol - Miscellaneous
(not complete)

```
NFSPROC_NULL
expects: nothing
returns: nothing
```

Used for pinging the server

```
NFSPROC_STATFS
expects: file handle
returns: file system stats
```

Returns file system stats: block size,number of free blocks, ...

The MOUNT protocol

The MOUNT protocol takes a directory pathname and returns an fhandle if the client has permissions to mount the file system

- Separate protocol
- Easier to plug in new permission check methods
- Separates the OS-dependent aspects of the protocol
- Other OS implementations can change the MOUNT protocol without having to change the NFS protocol

Security

NFSv2 uses UNIX-style permission checks

- The client passes uid/gid info in RPCs, and the server performs permission checks as if the user was performing the operation locally

- Problem – the mapping from uid/gid to user must be the same on the client and server
  - Can be solved via Network Information Service (NIS)
- Another problem – should root on the client have root access to files on the server?
  - Server specifies policy

“Stateless” Protocol - revisited

Idempotency

“Property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application”

- Idempotency is a useful property when building reliable systems
- When an operation can be issued more than once, it is much easier to handle failure of the operation:
  - you can just retry it!

Benjamin Peirce

Idempotency and NFS

- Your first question should be - how come?
- What about write operations?

Benjamin Peirce

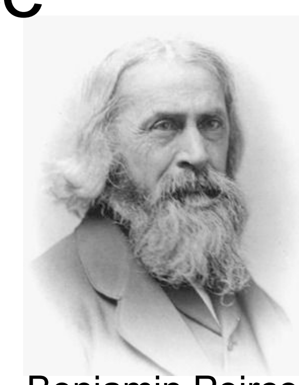
Idempotency and NFS

- Your first question should be - how come?
- What about write operations?

Definition of Write Operation:

Writes count **bytes with** offset **bytes to file**

This means that if you execute the same operation write a value to a memory location three times, it is the same as doing so once



Benjamin Peirce

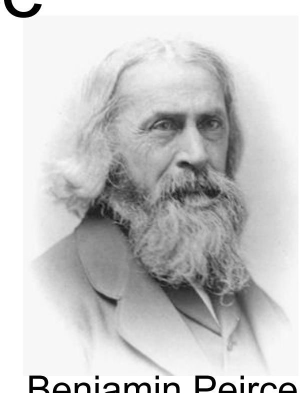
Idempotency and NFS

- Your first question should be - how come?
- What about write operations?

Definition of Write Operation:
Writes count **bytes with** offset **bytes to file**

This means that if you execute the same operation write a value to a memory location three times, it is the same as doing so once

Thus, "store value to memory" is an idempotent operation



Benjamin Peirce

Idempotency and NFS

- Your first question should be - how come?
- What about write operations?
- What about Mkdir?

Benjamin Peirce

Idempotency and NFS

- Your first question should be - how come?
- What about write operations?
- What about Mkdir?
    - Well, life is not perfect :-)

When you try to make a directory that already exists, you are informed that the mkdir request has failed

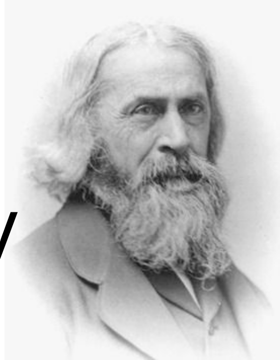
Benjamin Peirce

Idempotency and NFS

- Your first question should be - how come?
- What about write operations?
- What about Mkdir?
  - Well, life is not perfect :-)

In NFS, if the file server receives a MKDIR protocol message and executes it successfully but the reply is lost,

the client may repeat it and encounter that failure when in fact the operation at first succeeded and then only failed on the retry



Benjamin Peirce

Cache at Client-Side

Clients use caching and write buffering to improve performance - why?

Cache at Client-Side

Clients use caching and write buffering to improve performance - why?

Sending all read and write requests across the network can lead to a big performance problem:

“the network generally isn’t that fast, especially as compared to local memory or disk”

Cache at Client-Side

Clients use caching and write buffering to improve performance - why?

Sending all read and write requests across the network can lead to a big performance problem:

the network generally isn’t that fast, especially as compared to local memory or disk

Temporary buffer for file data and metadata read from server in client - but **also for writes**

Cache Consistency

Problem #1: Update visibility

If client C1 buffers writes in its cache, client C2 will see the old version

NFSv2 solution: Close-to-open consistency – Clients flush on close(), so other clients will see the latest version on open()

Slide adapted from Avishay Traeger - An Introduction to NFS

Cache Consistency

Problem #2: Stale cache

If C1 has a file cached, it will see old data even if the file is updated by C2

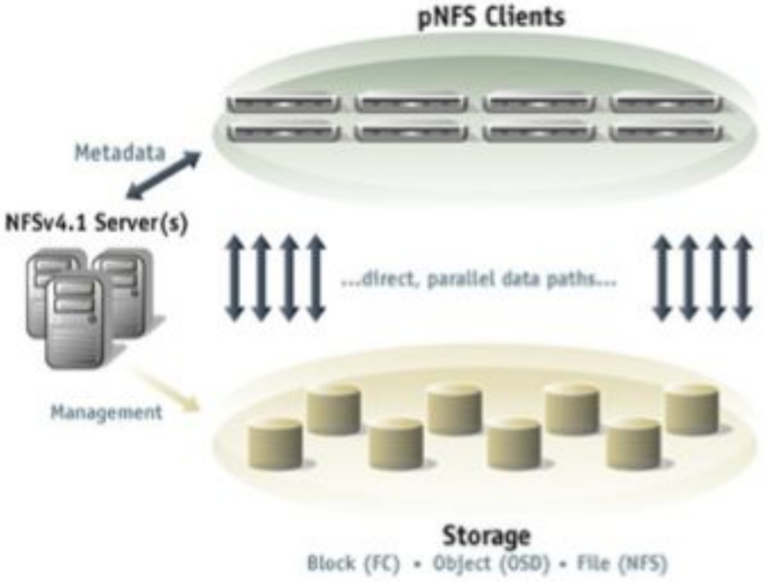
NFSv2 solution: Send a GETATTR and check the file's modification time to see if it has been updated.

Cache attributes for a few seconds to reduce the number of GETATTR calls

Slide adapted from Avishay Traeger - An Introduction to NFS

NFSv4.1
or Parallel NFS

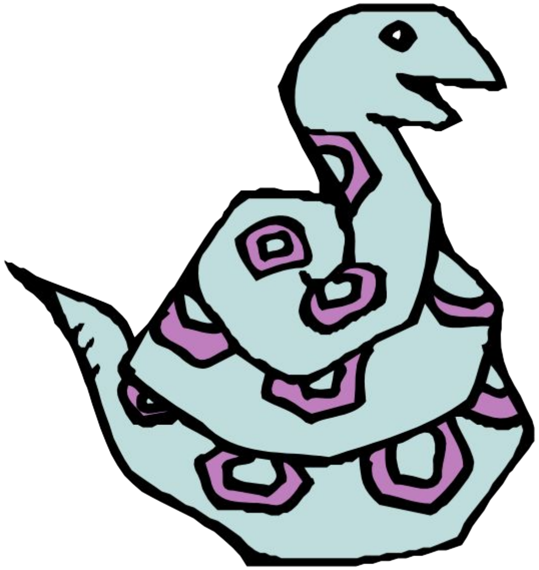
- Clients may now access storage devices directly and in parallel
- Eliminates the classic **NFS bottleneck of having only one server**
- The management protocol is NFSv4.1
- The data protocol can be
  - NFSv4.1 (Files)
  - OSD (objects), or
  - FC (blocks)


pNFS Clients
Metadata
NFSv4.1 Server(s)
...direct, parallel data paths...
Management
Storage
Block (FC) • Object (OSD) • File (NFS)

https://tools.ietf.org/html/rfc5661

Slide adapted from Avishay Traeger - An Introduction to NFS

RPC in Python



JSON RPC Basics

- What is JSON RPC?
  - Lightweight remote procedure call protocol
  - Check out: https://www.jsonrpc.org/
- What does it do?
  - Uses JSON as format to call methods at remote computer
  - JSON allows it to specify the method to use and its parameters
  - Process is carried out at server
  - Response from server is also in JSON format with the result of the method called

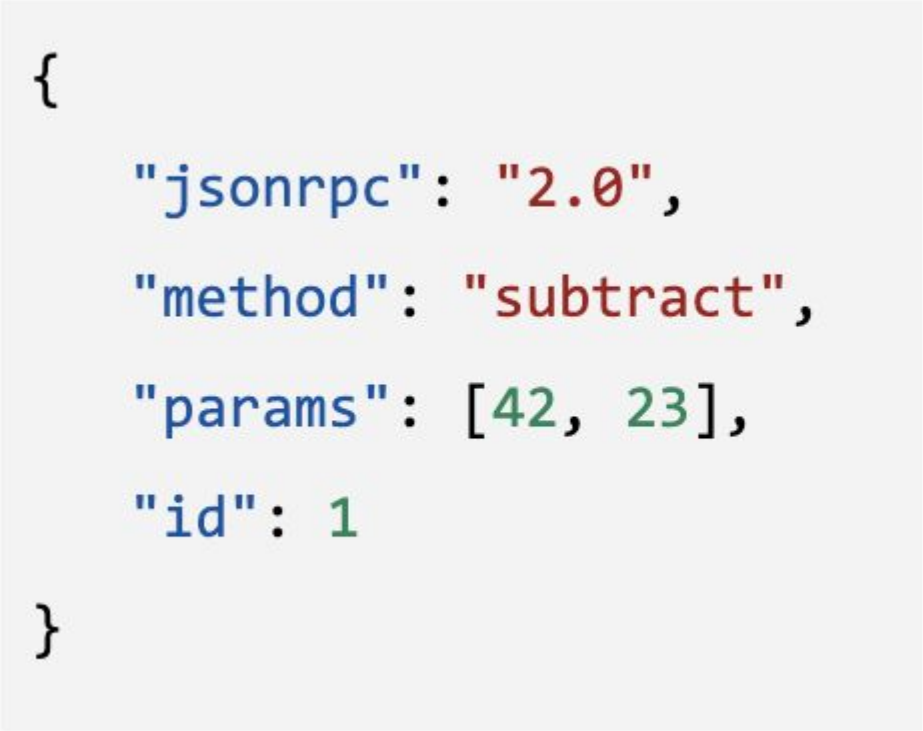
JSON RPC Basics

- What is JSON RPC?
  - Lightweight remote procedure call protocol
  - Check out: https://www.jsonrpc.org/
- What does it do?
  - Uses JSON as format to call methods at remote computer
  - JSON allows it to specify the method to use and its parameters
  - Process is carried out at server
  - Response from server is also in JSON format with the result of the method called

There are other options, e.g., XML RPC

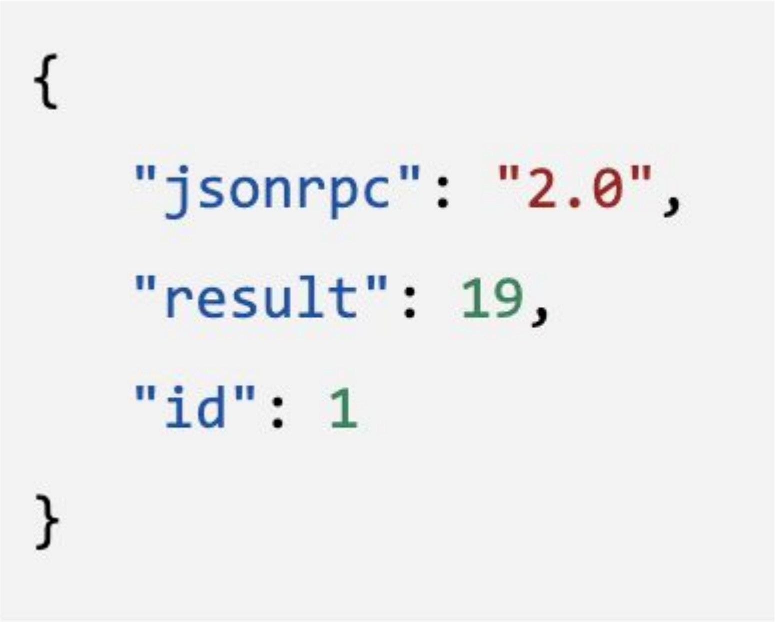
JSON RPC Basics

Request



```
{
   "jsonrpc": "2.0",
   "method": "subtract",
   "params": [42, 23],
   "id": 1
}
```

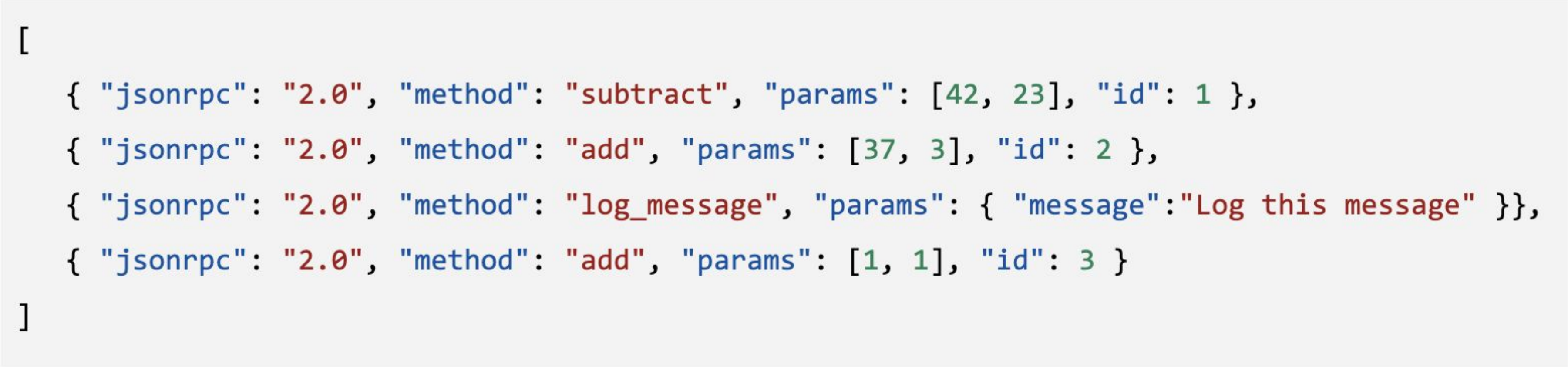
Response



```
{
   "jsonrpc": "2.0",
   "result": 19,
   "id": 1
}
```

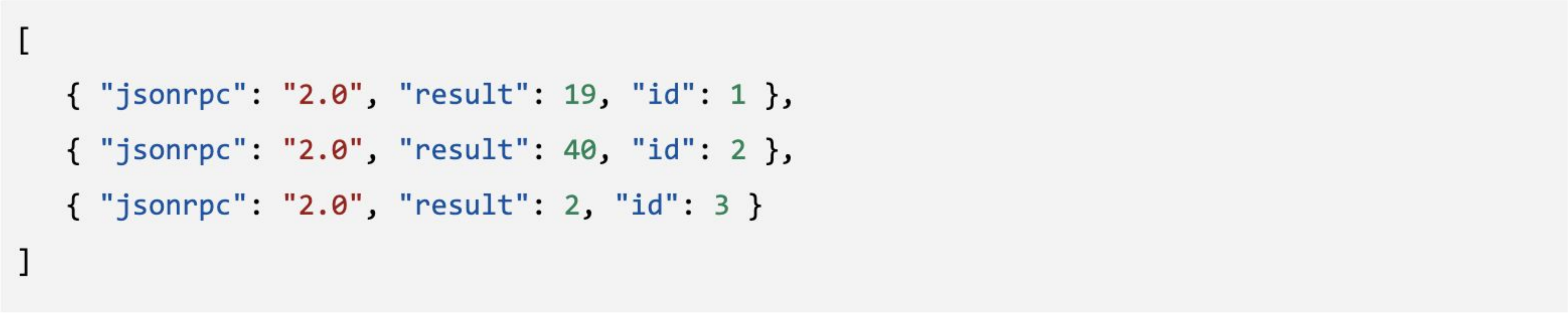
JSON RPC Basics

Batch request:



```
[
   { "jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1 },
   { "jsonrpc": "2.0", "method": "add", "params": [37, 3], "id": 2 },
   { "jsonrpc": "2.0", "method": "log_message", "params": { "message":"Log this message" }},
   { "jsonrpc": "2.0", "method": "add", "params": [1, 1], "id": 3 }
]
```

Batch response:

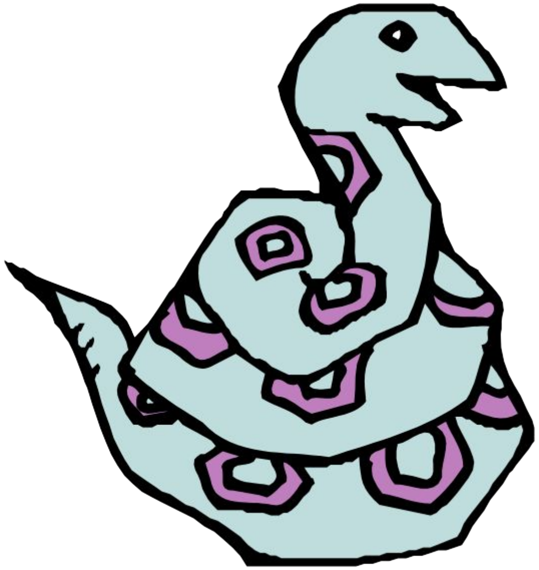


```
[
   { "jsonrpc": "2.0", "result": 19, "id": 1 },
   { "jsonrpc": "2.0", "result": 40, "id": 2 },
   { "jsonrpc": "2.0", "result": 2, "id": 3 }
]
```

JSON RPC Basics

- Advantages:
  - Simple
  - Human readable (easy to debug)
  - JSON object ↔ Python dictionary
  - Every major language/framework supports JSON
- Disadvantages
  - Messages are larger than a binary protocol
  - Binary data is not allowed in JSON - base64 encoding makes data 33% longer
- Recommended use:
  - Server-to-server communication
  - No file transfer
  - Python library: **tinyrpc**

REST API in Python



REST API

- What is a REST API?
  - A web service API with a special URL structure and well-defined HTTP response codes
  - Accepts and returns JSON or binary data
  - 4 main operations per resource (e.g. user, file, container, ...), different HTTP request types
    - GET: Retrieve information
    - POST: Create a new instance of the resource
    - PUT: Edit an existing instance
    - DELETE: Remove an existing instance
    - HEAD: retrieve metadata about an instance.

Sample REST API

User endpoints:

`GET /user`: List all users

`GET /user/[id]`: Return the details of one specific user

`POST /user`: Create a new user from the JSON object sent in the request body

`PUT /user/[id]`: Update the user who has the given ID, the changed attributes are in the request body as a JSON object

`DELETE /user/[id]`: Remove the given user

Container and object endpoints:

`GET /container`: List all containers

`GET /container/[container-id]`: List the contents (objects) of the given container

`GET /container/[container-id]/[object-id]`: Download the given object

`HEAD /container/[container-id]/[object-id]`: Get metadata of an object

…

REST API

- Response status code carries information
  - 2xx: Success
  - 4xx: Bad request
  - 5xx: Server error

- Recommended use
  - Public interface of your service
  - Preferred for browser-based clients
  - Document it with OpenAPI
  - Python frameworks: Flask (see lab) or Django