Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor thesis

Sandro Grizzo

Mapping process IDs to NFSv4 I/O metrics between computing and storage nodes through Linux kernel inquiry using eBPF

Faculty of Engineering and Computer Science
Department Computer Science

Sandro Grizzo

Mapping process IDs to NFSv4 I/O metrics between computing and storage nodes through Linux kernel inquiry using eBPF

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Informatik Technischer Systeme*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Sciences Hamburg

First Supervisor: Prof. Dr. Christian Lins (HAW)
Second Supervisor: Dr. Thomas Hartmann (DESY)
Consultant Supervisor: Tigran Mkrtchyan (DESY)

Submitted on: 06.01.2025

Sandro Grizzo

Thema der Arbeit

Zuordnung von Prozess-IDs zu NFSv4 I/O-Metriken
zwischen Rechen- und Speicherknoten
durch Nachforschungen im Linux-Kernel mittels eBPF

Stichworte

eBPF, Linux kernel, NFSv4+, dCache-Speicher, DESY

Kurzzusammenfassung

Die Verwaltung komplexer wissenschaftlicher Rechen- und Speicheranlagen, wie die am Forschungsinstitut DESY (Deutsches Elektonen-Synchrotron), stellt Systemadministratoren insbesondere bei der effektiven Diagnose und Lösung von Systemstörungen vor große Herausforderungen. In dieser Arbeit wird die Entwicklung benutzerdefinierter eBPF-Programme (extended Berkeley Packet Filter) untersucht, um den Einblick in den Betrieb des Linux-Kernels zu ermöglichen und den Administratoren Überwachungs- und Diagnosefunktionen zur Verfügung zu stellen. Die vorgeschlagenen Programme zielen darauf ab, die Systemadministration insbesondere im Hinblick auf die Handhabung von Problemen zu vereinfachen und zu beschleunigen. Dabei wird die Fähigkeit von eBPF genutzt, Echtzeiteinblicke in den Kernel mit minimalen Leistungseinbußen zur Verfügung zu stellen.

Sandro Grizzo

Title of Thesis

Mapping process IDs to NFSv4 I/O metrics
between computing and storage nodes
through Linux kernel inquiry using eBPF

Keywords

eBPF, Linux kernel, NFSv4+, dCache storage, DESY

Abstract

Managing complex scientific computing and storage facilities such as the ones at the research center DESY (Deutsches Elektonen-Synchrotron), presents significant challenges for system administrators, particularly in diagnosing and resolving issues effectively. This thesis explores the development of custom eBPF (extended Berkeley Packet Filter) programs to enhance visibility into Linux kernel operations and provide monitoring and diagnostic capabilities to administrators. By leveraging eBPF's ability to enable real-time insights into the kernel with minimal performance costs, the proposed programs in particular aim to simplify and accelerate system administration practices with regard to managing issues.

Contents

List of Figures

List of Tables

Listings

List Of Acronyms

ALICE A Large Ion Collider Experiment
ALPS Any Light Particle Search
API Application Programming Interface
ATLAS A Toroidal LHC ApparatuS
BCC BPF Compiler Collection
BPF see eBPF
BTF BPF Type Format
CERN European Organization for Nuclear Research
CO-RE Compile Once - Run Everywhere
CMS Compact Muon Solenoid
CPU Central Processing Unit
DCAP dCache Access Protocol
DESY Deutsches Elektronen-Synchrotron
DNS Domain Name System
DOT dCache Operation Team
DS Data Server
eBPF Extended Berkeley Packet Filter
EOS Exactly Once Semantics
FD File Descriptor
FLASH Freie-Elektronen-Laser in Hamburg
FTP File Transfer Protocol
GID Group Identifier
HEP High Energy Particle Physics
HPC High-Performance Computing
HTC High-Throughput Computing
ID Identifier
IETF Internet Engineering Task Force
JVM Java Virtual Machine
JSON JavaScript Object Notation
I/O Input/Output

IP Internet Protocol
LHC Large Hadron Collider
MDS Metadata Server
NAF National Analysis Facility
NFS Network Filesystem
NFSv4+ Network Filesystem Version 4 with + $\in \{0, 1, 2\}$
ONC Open Network Computing
OOP Object-Oriented Programming
PB Petabyte
PID Process Identifier
PNFSID Perfectly Normal Filesystem Identifier
pNFS Parallel NFS
RHEL Red Hat Enterprise Linux
RPC Remote Procedure Call
RFC Request For Comments
SMP Symmetric Multi-Processing
SSH Secure Shell
TGID Thread Group Identifier
TCP Transmission Control Protocol
UID User Identifier
UUID Universally Unique Identifier
VFS Virtual Filesystem Switch
VM Virtual Machine
WLCG Worldwide LHC Computing Grid
WN Worker Node
XDR eXternal Data Representation
XFEL X-Ray Free Electron Laser
XFS X Filesystem
XID RPC-related Transfer ID

«**Follow the white rabbit...**»

The Matrix, Wachowskis, 1999

1 Introduction

In modern scientific computing and storage facilities, the increasing complexity and scale of infrastructure pose significant challenges to system administrators. These professionals' task is to ensure the seamless operation of computational and storage systems that support a wide range of research activities. The management of such facilities often requires deep insight into system behavior to allow for a fast resolution of issues. However, obtaining these levels of insight is a non-trivial task.
The Linux kernel, as the basis of many large-scale computing environments, plays a crucial role in scientific system operations, too. Yet, despite its brilliantly engineered functionality, it often appears as an opaque space in many aspects, making it difficult to gain detailed insight into its internal workings. Unfortunately, this lack of transparency works to the disadvantage of those tasked with handling the related issues. It often leaves system administrators without the tools needed to fully comprehend or manage issues efficiently.
This thesis explores the development of custom eBPF (extended Berkeley Packet Filter) programs tailored to enhance system administrators' ability to monitor and diagnose system-related issues in these complex environments. Leveraging eBPF as a customizable technology offers a flexible way to enable real-time insights into the kernel's operations without significantly impeding system performance. Through the design and implementation of these custom-built programs, this study seeks to contribute to a more transparent system environment, supporting the operational efficiency of system administrators in their critical roles.

1.1 Experimental Physics and Research at DESY and abroad

The German research center DESY (Deutsches Elektronen-Synchrotron) has evolved into an ideal catalyst for fundamental research in the field of experimental physics in the past six decades. Besides its important role as driving force for the design, development and execution of on-site and off-site experimentation, it offers an extensive computational infrastructure for the analysis of data obtained through these experiments. [4] The main fields of experimental physics hosted at the DESY campus encompass the domains of high energy particle (HEP) physics, photon science and the development of accelerator technology. Photon science

experiments are carried out at Petra III, a synchrotron radiation source, together with the European XFEL and FLASH, which both produce ultra-short x-ray-laser flashes. [5]
The high energy particle sector at DESY is represented by an experiment named ALPS II. This research project deals with lightweight particles and their potential entanglement with the dark matter of the universe [6]. The main contributors to high energy particle research and experimentation outside of the DESY campus are settled at the renowned LHC (Large Hadron Collider) at CERN (Organisation Européenne pour la Recherche Nucléaire). This huge circular accelerator, located underneath the border area between Switzerland and France is home to experiments such as ATLAS, CMS, LHCb and ALICE. All of them, in one way or another, examine the inner structure of particles and study the smallest forces that keep matter consistent. [7]
A consequence of this vivid experimentation activity is a veritable «data deluge». The average amount of collision data recorded on disk by the LHC experiments in 2024 has exceeded the 900 petabyte (PB) mark already. And the trend points upward. [8] Figure 1.1[1] shows the demand for storage space at DESY over the course of the last thirteen years. Blue denotes the demand by the on-site HEP experiments, while orange shows the same for photon science.

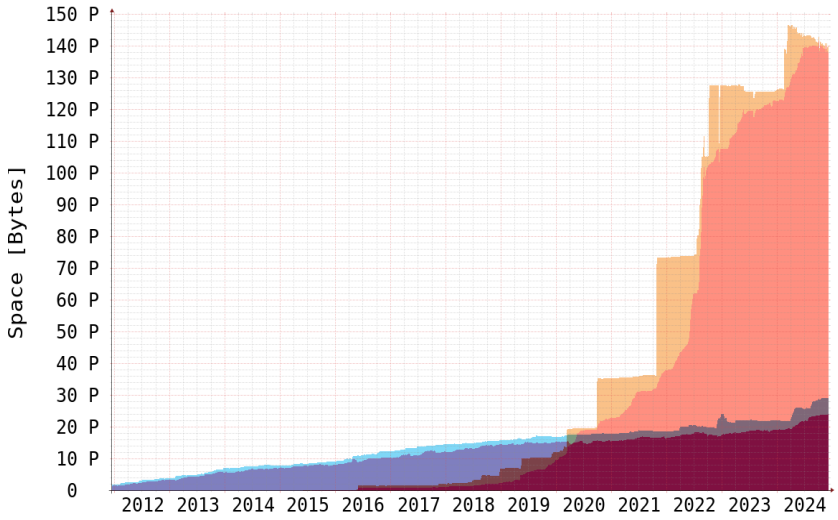

Figure 1.1: Increase in dCache storage space (in PB) for HEP (blue) and photon science (orange) at DESY over the last 13 years. Brighter colors show the absolute space, darker colors the effectively used space.

[1] Figure kindly provided by Thomas Hartmann, NAF and Grid system administrator at DESY.

1.2 The Scientific Computing Infrastructure at DESY

These large amounts of experimental data require infrastructure for their storage and archiving as well as computing facilities to analyze them. With its scientific computing infrastructure, the DESY research center provides storage and computational capacities to experiments conducted both on campus and abroad. For the global HEP-Community, DESY contributes to the Worldwide LHC Computing Grid (WLCG) as a Tier-2 grid computing center. The Maxwell cluster on DESY campus offers a High-Performance Computing (HPC) platform for the photon science data analysis. In addition, it allows GPU-accelerated computations for AI development and complex simulations [9]. One of the two infrastructural components relevant to this study is the *National Analysis Facility* or NAF. It serves as a general purpose batch system optimized for High-Throughput Computing (HTC) [10]. This facility is briefly introduced in Section 1.3. The entirety of the large-scale scientific computing infrastructure at DESY constitutes the Interdisciplinary Data and Analysis Facility (IDAF). [7]
The second infrastructural element considered in this study is the *dCache storage system*. At its core is an eponymous storage management software tailored specifically for the storage of scientific data. It allows for the deployment of a highly available and scalable distributed storage network.[11] The dCache storage system is described in more detail in Section 2.9.

1.3 The National Analysis Facility (NAF)

The NAF batch system, as the name states, is a computing facility available to scientists at DESY as well as to researchers from other institutes located in Germany. At the time of writing, the NAF comprises a total of 80 working group servers and about 200 worker nodes hosting 10390 computing cores [12] with a performance benchmark according to the HEP benchmark suite[2] HS23 [14] metrics of approximately 287 kHS23 [12]. All involved nodes run the Red Hat Linux distribution (RHEL 9.4) as their operating system.
Among other computing tasks, such as the development, testing and debugging of applications, the NAF allows scientists to interactively perform small-scale, fast-turnaround analyses of their experimental data [15]. The resource management of the worker nodes is managed by a batch and scheduling software called HTCondor [16]. The open source software[3] provides an appropriate match of available resources on the worker nodes to the hardware and

[2] «The HEP Benchmark Suite is a toolkit which orchestrates different benchmarks in one single application for characterizing the performance of individual and clustered heterogeneous hardware» [13].
[3] At the time of writing, HTCondor version 24.0.2 is used.

computation time requirements of the user's computation tasks. The latter are commonly referred to as user *jobs* by the administrators of the analysis facility. HTCondor allocates a job to a so-called *slot*. These slots define the computation time, the number of CPU cores, and the amount of memory resources tailored to match the job's requirements as closely as possible.

1.3.1 Bird's Eye View of a Job Workflow

For a user to submit an analysis job to the computing nodes of the NAF, they have to connect to a working group server (WGS) first. After successful authentication with the WGS the user starts job submission by defining their requirements in a submit file. Entering the appropriate shell commands will initiate a sequence of steps according to the user's specification of computation time and hardware requirements.
HTCondor will ...

- evaluate the submit file and queue the job,
- negotiate an appropriate slot for the job,
- schedule the job within the chosen slot on a worker node,
- make sure the time and space constraints are not exceeded by the job,
- return computation results to the user application and
- free the slot off the worker node after expiration of assigned computing time.

In order to perform the desired computation during the execution of the job, externally stored file data are required. In the case of the NAF, the file data are stored and managed by the dCache storage system. In order to access the remotely stored file data, a filesystem tree of the dCache storage is mounted onto the worker node's local filesystem via the network filesystem protocol version 4.1 (NFSv4.1). This is accomplished by the Linux built-in NFSv4+ client on the computing nodes of the NAF and a Java-based custom NFSv4.1 server implementation on the dCache storage end. The NFSv4.1 client handles the user job's file I/O requests such as open, read and write, by serializing them with the help of remote procedure calls (RPC) which are subsequently transmitted via the Transmission Control Protocol (TCP) across an Ethernet network.
At the other end of the network, a dCache NFSv4+ entry point receives the RPCs and deserializes them to NFSv4+ operations again. Once the appropriate dCache storage node that is capable of serving the file I/O request is determined, the transmission of the file data

is initiated. Files stored on dCache storage are immutable. Consequently, they can be written to or read from the storage nodes. But modified file data is never written back to the same file again[4]. The file data received by the NFSv4.1 client on the worker node of the NAF is made available to the user's job for computations and analyses subsequently.
Section 2 provides a more detailed discussion of the two core concepts briefly introduced here. The dCache storage system is covered in Section 2.9, while the NFSv4+ protocol is addressed in Section 2.8.

1.4 Current State Analysis

Considering the number and complexity of interactions between components of a computing and storage environment as the one presented above, the probability of any kind of issues to arise is high. The issues contained in the task space which administrators of these facilities are confronted with depend on multiple factors that are not always clearly distinct from one another. The fact that system administrators at DESY have no insight into the source code of jobs submitted by the scientific users of the facilities often leaves them guessing as to the solution for issues they perceive. Consequently, their ability to take appropriate measures is limited to the amount of information or metrics about the flawed system accessible to them. Several recently submitted theses address this lack of information on the administrator's side and explore potential solutions to increase the amount of valuable metrics to facilitate and accelerate solving issues. One of the submitted theses deals with dCache storage access patterns from user jobs that endanger the high availability of the storage system by overloading storage elements, consequently throttling transfer rates or even rendering the former inaccessible [18]. The author of the stated thesis proposes and implements a software utility based on machine learning algorithms aimed to allow predictions about potentially harmful access patterns. A second thesis proposes enhancing information readily available to administrators by providing *pre-mortem* messages from a dCache component involved in data transfer called *mover* (see chapter 2.9) [19]. Through these messages, such issues as the overload of storage elements mentioned earlier can be anticipated and prevented. Figure 1.2[5] illustrates the effect of diverging access patterns of user jobs onto dCache storage elements over time. Each colored rectangle represents a time span of three hours. The heatmap depicts high access rates in reddish colors, while low activity is colored green.

[4] This type of storage is commonly referred to as *WORM* - Write-Once, Read-Many storage [17]
[5] Kindly provided by Tigran Mkrtchyan, head of the dCache developer team at DESY.

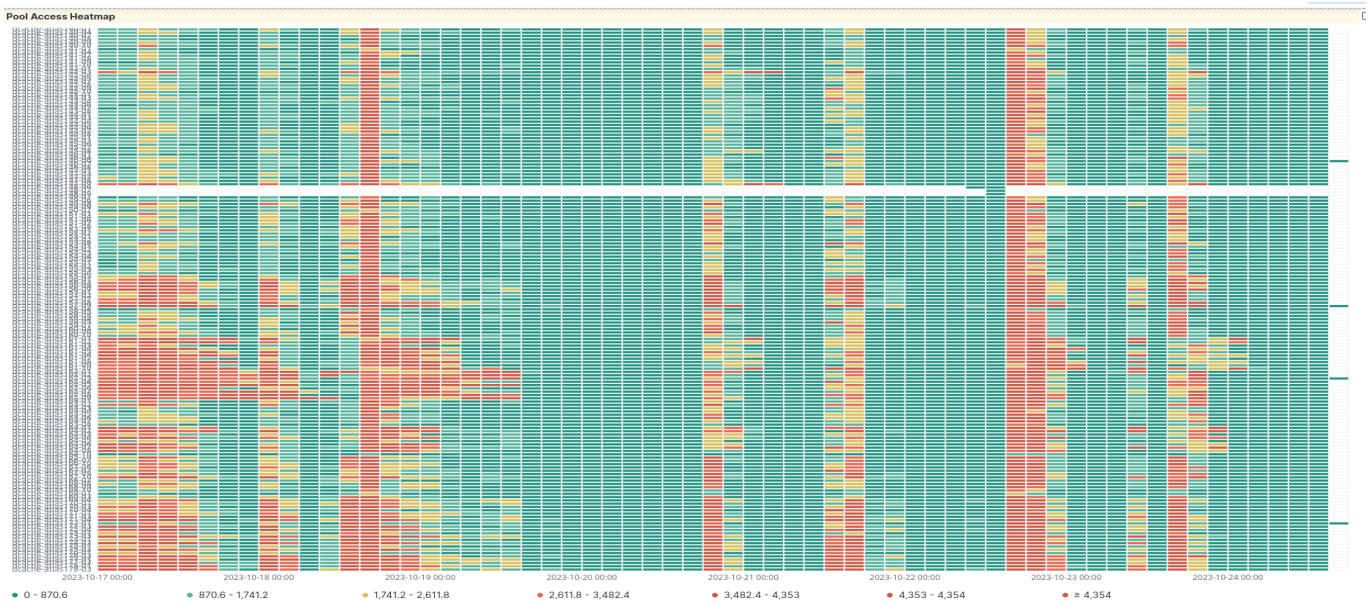

Figure 1.2: Heatmap illustrating access to dCache storage elements on the y-axis over time on the x-axis. Higher access rates are depicted in red, low activity is shown in green. Each rectangle represents a time span of three hours.

Another issue related to the NFSv4+ communication between computing nodes and storage nodes at DESY is the infinite NFSv4+ client loop. An unrecoverable loss of state information between a metadata server and a data server causes a NFSv4+ client to oscillate infinitely between the two servers due to contradictory information received from either. While the client loops, the user's application hangs waiting for the I/O request to complete. This, among many other issues, shows the urgent necessity to provide administrators with insight into the inner proceedings of their systems.

1.5 Research Objectives

This present thesis aims to pursue this goal by providing administrators with an x-ray view into one of the principal software components underlying the whole scientific computing infrastructure at DESY: The Linux operating system kernel. This is achieved by leveraging a technology called *eBPF* whose infrastructure is already built into the Linux kernel itself. The eBPF technology enables the monitoring of all kernel subsystems at runtime. It allows the collection of relevant metrics that can help administrators to aggregate valuable

information about their systems at runtime and derive effective strategies for the resolution of kernel-related issues [2]. This work focuses primarily on monitoring the kernel's built-in NFSv4+ client including its associated subsystems. It specifically explores the viability of obtaining metrics about the amount of data transferred via the NFSv4+ protocol and the feasibility of attributing those amounts of data to a specific user's job or a sub-process contained therein. Furthermore, it investigates the potential time-savings in the administrator's procedure of attributing opened file-related metrics to user jobs with and without the help of eBPF. This comprises the implementation of three custom eBPF programs including the gradual verification of the accuracy of the obtained metrics.

1.6 Related Work

Apparently, as far as the author's inquiries are concerned, no research has been done on this site-specific subject to date. However, in the context of research on leveraging eBPF to track down system specific *performance* issues such as network bottlenecks [20], memory management issues [21], lock and scheduling related problems [22], as well as security issues [23, 24], many articles and even a few books [2, 25] have been published by researchers as well as expert system administrators. One recent publication [26], which could be considered closely related to the research objectives of this present thesis, addresses the subject of real-time NFS performance metrics extraction with a custom-built eBPF program called *Tracklops*. The said publication, however, looks at the subject for the sake of mitigating load-balancing issues which is not directly the concern of this present work.

2 Core Concepts

This section examines the core concepts underlying the development and application of the proposed custom eBPF programs. The purpose of this detailed exploration is to outline the level of understanding required to identify and gather all relevant information from the involved systems and technologies, such as the Linux kernel.
The ultimate goal of the eBPF programs is to monitor metrics from the running Linux kernel and make them accessible to system administrators. Identifying the appropriate approach for developing and applying an eBPF program requires a thorough understanding of the kernel subsystem to be monitored. Part of this identification process is reflected in the descriptions provided in the following sections.
All given references and paths to the Linux kernel source code are meant relative to the root directory of the kernel source tree and refer to version 5.14.

2.1 Processes and Threads

One fundamental concept in operating systems is that of a process. According to a commonly cited definition, a process is an instance of a program in execution [27, 28]. Every userland process has its own user address space allocated in memory that is not accessible by other userland processes[1]. Neither can this very process access address spaces of other processes, unless it utilizes inter-process communication mechanisms provided by the kernel exactly for that purpose. The address space of a userland process is subdivided into so-called segments, or mappings in memory. One of these segments contains one, out of two, of the process' stacks, called *user mode stack*, as opposed to its *kernel mode stack*, discussed further below in this section. In addition, the dynamically allocatable *heap* memory as well as segments for the machine code (*.text* segment) of the process and its global and static variables (*.data* segment) coexist in this user address space. Library code used by the process is typically also mapped between the stack and heap segment. Every process consists of at least one *thread* of execution, commonly referred to as *main* thread. A thread can be considered an independent execution path within a process. Although being independent, it shares all the

[1] The following information are cited from [29], if not mentioned otherwise.

system resources currently used by the process as well as the segments that are contained within the address space of the process, apart from its stack. Naturally, the extent of shared resource usage between a newly spawned thread and its parent thread can be customized with the help of *flags* passed as arguments to the `clone()` call, which is invoked whenever a new thread has to be spawned. Each thread has its own stack memory space that is not shared with other threads but is smaller in size than the main thread's stack space. The Linux default main thread's stack size can be queried from a command shell with the `ulimit` command, as shown in Listing 2.1.

```
$ ulimit -s
8192
```

Listing 2.1: The ulimit command with the `-s` switch for stack size retrieval

The `ulimit` command, invoked with the `-s` option, yields the value of 8 MiB (the values are in increments of 1024 bits). New threads spawned by the main thread during runtime, usually get a default stack size of 2 MiB on x86_64 architectures, according to the `man 3 pthread_create` Linux manual page entry.

In Linux, there are two levels of execution privileges. Each processor architecture executes specific instructions to switch between these two levels. One is the unprivileged user mode, which is active when userland program code is being executed on a CPU core. In this mode, the process is unable to access any kernel data structures. The very moment the user process requires access to resources or data that are unreachable from within the unprivileged user execution context, a *system call* (more on this in Section 2.4) has to be issued toward the kernel, triggering a switch from user mode to privileged kernel mode. In kernel mode, kernel code paths are executed safely and efficiently to service the user thread's requests. Once the request is completed, the user process is switched back to unprivileged user mode [27]. While executing in kernel mode, the user thread utilizes its kernel mode stack allocated to every active process within kernel memory. Consequently, every user thread, including the main thread, has access to two stacks, the one it owns in the user address space and the other residing in the kernel address space of memory. One exception from this proceeding are the so-called kernel threads. These threads are either created at boot time and have very specific tasks assigned to them, or they are created on demand, whenever needed. Kernel threads execute kernel code exclusively and therefore have access to a stack in kernel memory only. They are completely unaware of the userland address space. In contrast to the size of the user stack, the kernel stack spans only 4 pages on 64-bit architectures, with a typical page

size of 4 KiB. The page size can be easily verified by invoking the `getconf` command on a Linux command shell as shown in Listing 2.2.

```
$ getconf PAGE_SIZE
4096
```

Listing 2.2: The getconf command yielding the utilized memory page size.

Awareness of the stack sizes is relevant for understanding the constraints enforced by the verifier component in the context of the in-kernel eBPF probe development described in Section 3.2.

2.2 The Process Descriptor

One of the main purposes of an operating system is to enable user applications to run safely and fast. Thus, in the context of *active* applications and processes, it is often stated that the Linux «kernel itself is not a process but a process manager» [27]. In order to keep track of all processes running on the system, be it threads in user applications or kernel threads, the kernel maintains a *process descriptor*, sometimes also referred to as *process control block* (PCB). It is allocated within a kernel slab cache in the form of a data structure called `struct task_struct`, defined in `/include/linux/sched.h`. The slab cache is the domain of the slab allocator which, instead of performing frequent time-consuming allocations and de-allocations, maintains memory areas of fixed sizes and reuses them for repeated memory allocation requests of the same type [27]. The size of the rather large task data structure is almost 8 KiB. This value can be obtained by querying the slab's task_struct cache from a command shell as shown in Listing 2.3 [30].

```
# cat /sys/kernel/slab/task_struct/object_size
7368
```

Listing 2.3: Retrieving the size of a task structure in bytes in the slab's task_struct cache from sysfs.

In the following, the terms *process*, *thread* and *task* are used interchangeably.

As mentioned before, the `struct task_struct` is the kernel's representation of one active thread of execution. It gathers all available information about the thread's execution context. The data structure stores information about the process' state, the memory address space and its usage, CPU scheduling details such as priorities, currently opened files, credentials

and identifiers as well as relationships to other processes such as parent, siblings and children. In total, `struct task_struct` contains more than 270 fields. Filesystem and namespace information are also contained. Additionally, the kernel maintains a circular, doubly linked list of type `struct list_head` in a field called `tasks` containing pointers to the previous and the next active task structures, respectively. Naturally, the head of this list is the task first born during the system's boot process, the root of the task tree, called `init_task` [27].

2.2.1 The `current` Macro

For the sake of lookup, retrieval, and manipulation of information it is indispensable that the diverse kernel subsystem routines have access to the task structure of the currently running thread at any time. While iterating the `tasks` list for this purpose is too costly due to the potentially high number of tasks in the list and the application of costly locking mechanisms, the kernel developers have designed a different, more efficient approach. A macro called `current` is utilized that contains the pointer to the task structure of the currently executed thread. The definition of the `current` macro is architecture-dependent. And while architectures like `aarch64` or `powerpc` store the pointer in one of their abundant processor registers, where it can be read from very rapidly, the `x86_64` architecture lacks this abundance of registers. Thus, the mechanism used in `x86_64` architectures to retrieve the pointer to the `struct task_struct` containing the current execution context, starts by declaring a so-called *per-CPU variable*. [29]

2.2.2 Declaring a Per-CPU Variable for the `current_task`

These special variables are allocated in a per-CPU memory area of each processor core present in a symmetric multiprocessing (SMP) system. The advantage of using per-CPU variables is that every processor core has its own copy of the per-CPU variable, allowing a fast, lock-free yet exclusive access to its own instance of the variable [31]. In order to set the preconditions for the retrieval of a per-CPU variable's value, like the pointer to `struct task_struct`, from the per-CPU memory area, the per-CPU variable has to be declared first. Listing 2.4 shows the `DECLARE_PER_CPU` macro, which takes the type of the per-CPU variable to be declared and an identifier as arguments.

```
DECLARE_PER_CPU(struct task_struct*, current_task);
```

Listing 2.4: Declaration of the `current_task` per-CPU variable

The chain of macro expansions, all defined in `/include/linux/percpu-defs.h` is shown in Listing 2.5.

```
#define DECLARE_PER_CPU(type, name)
 DECLARE_PER_CPU_SECTION(type, name, "")
...
/* which in turn expands to */
#define DECLARE_PER_CPU_SECTION(type, name, sec)
 extern __PCPU_ATTRS(sec) __typeof__(type) name
...
/* which expands to */
#define __PCPU_ATTRS(sec)
 __percpu __attribute__((section(PER_CPU_BASE_SECTION sec)))
PER_CPU_ATTRIBUTES
...
/* with PER_CPU_BASE_SECTION finally expanding to */
#define PER_CPU_BASE_SECTION ".data..percpu"
```

Listing 2.5: Macro expansions showing the per-CPU variable's memory section assignment

The last line of Listing 2.5 reveals that after all macros are expanded, the `current_task` per-CPU variable is allocated in the `.data..percpu` section of the per-CPU memory area of each processor core. This occurs, as already established above, during the kernel initialization phase at system boot time [32]. With this precondition set, it is now possible to comprehend how the access path to the currently running thread's `task_struct` pointer is implemented by studying the kernel sources.

2.2.3 Retrieval of the Per-CPU Variable

Returning to the `/arch/x86/include/asm/current.h` file, in which, apart from the per-CPU variable declaration macro described above, the `current` macro is defined for machines with `x86_64` processor architectures. This macro definition is illustrated in Listing 2.6.
During compile time, `current` is replaced by the function call to `get_current()` by the C compiler's preprocessor everywhere in the kernel code. As expected, the `get_current()`

```
#define current get_current()
```

Listing 2.6: Definition of the `current` macro

function returns a pointer to the `struct task_struct` of the currently running thread. It is actually a wrapper for a macro named `this_cpu_read_stable(current_task)`. The latter is defined in `/arch/x86/include/asm/percpu.h` like the rest of the macro definitions and inline assembly code presented in the following with one exception. The macro called `__pcpu_size_call_return(stem, variable)`, which is also part of Listing 2.7, is defined in `/include/linux/percpu-defs.h` instead.

```
#define this_cpu_read_stable(pcp)
__pcpu_size_call_return(this_cpu_read_stable_, pcp)
...
/* which, assuming a variable size of 8 bytes returns: */
this_cpu_read_stable_8(pcp)
...
/* which itself is a macro that resolves to */
#define this_cpu_read_stable_8(pcp)
percpu_stable_op(8, "mov", pcp)
```

Listing 2.7: Macro expansions used in the access path to the per-CPU variable

At the very end of the macro expansion chain, as shown in Listing 2.7, another macro called `percpu_stable_op(8, "mov", pcp)` emerges. It contains the inline assembly code needed to copy the per-CPU variable from the per-CPU memory area to a general-purpose register of the processor core for fast retrieval. This piece of kernel code, illustrated in its original form in Listing 2.8 needs a more detailed explanation. To a great extent, the difficulty to read it results from its generic nature. Note that Listing 2.9 shows the same definition as Listing 2.8. But, for the sake of clarity, all macros are expanded and the actual parameter values are passed to the parameter list. These parameters replace every variable listed in the function body that references one of them. Now the code reveals the instructions that are used to find the address of the `current_task` per-CPU variable at runtime. The listing containing the macros used for expansions and replacements can be found in Appendix A.1.

```
1  #define percpu_stable_op(size, op, _var)
2  ({
3      __pcpu_type_##size pfo_val__;
4      asm(__pcpu_op2_##size(op, __percpu_arg(P[var]), "%[val]")
5          : [val] __pcpu_reg_##size("=", pfo_val__)
6          : [var] "p" (&(_var)));
7      (typeof(_var))(unsigned long) pfo_val__;
8  })
```

Listing 2.8: The function macro definition of `percpu_stable_op()`

The code of Listing 2.9 begins with the declaration of a variable of type unsigned integer, 64-bit in size, called `pfo_val__`. In line 4 the `asm` keyword announces a forthcoming block of inline assembly code to the *gcc* compiler.

```
1  #define percpu_stable_op(8, "mov", current_task)
2  ({
3      u64 pfo_val__;
4      asm("movq %%gs:%P[var] , %[val]")
5          : [val] "=r" (pfo_val__)
6          : [var] "p" (&(current_task)));
7      (typeof(current_task))(unsigned long) pfo_val__;
8  })
```

Listing 2.9: Same definition as in listing 2.8 but with all macros and parameters expanded and replaced.

According to the documentation [33] of the GNU Compiler Collection, the general AT&T syntax of the assembly code in lines 4 through 6 can be formalized as:

$$\begin{aligned} &\text{" } < instruction{+}wordsize >< src >:< offset >, < dst > \text{ "} \\ &\qquad :< output\ operand\ constraints > \\ &\qquad :< input\ operand\ constraints > \end{aligned}$$

With this in mind, the assembly instructions can be interpreted as the following sequence of actions:
Copy the contents of the memory address[2], to which the contents of the *gs* register plus the offset to the memory address of the `current_task` per-CPU variable points to, into

[2] Of a **q**uadword, hence the *q* appended to the *mov* mnemonic. In contrast to a **b**yte (8-bit), **w**ord (16-bit) and **l**ong (32-bit) wordsizes, the quadword on x86_64 architectures is defined as being 64-bit in size [34].

a general purpose register of the processor and store it in the `pfo_val__` variable. Finally, cast the `pfo_val__` variable to be of type `struct task_struct*` pointer.
The documentation notes found in `/arch/x86/include/asm/stackprotector.h` inform about the following *gs* register related context:

> «[...] The same segment (to which the contents of the *gs* register points to)[3] is shared by percpu area and stack canary.
> On x86_64, [...] %gs (64-bit) points to the base of percpu area. [...]»

The mentioned `stack canary`[4] is a method to check for and protect against stack overflows by issuing a warning in the case a stack canary is overwritten by buggy or malicious code. However, with the chain of events described above, the kernel is able to retrieve the `struct task_struct` of the currently running task whenever needed. This is achieved by expanding the `current` macro down to assembly instructions which allow the memory address of the task structure to be read and copied from per-CPU memory rapidly.

[3] Added for clarity.
[4] The term *canary* is said to be deriving from the days of coal mining. Canaries were used as alarms in case poisonous gases leaked. The moment the canaries stopped singing, it was time to leave [35].

2.3 Process Identifiers

As seen in Section 2.2 the kernel maintains a `struct task_struct` for each active thread of execution. The task structure has two fields of type `pid_t`. One is called `pid`, the other one is called `tgid`. The PID or *process identifier* field contains a numeric value that uniquely identifies every single active thread. The TGID which stands for *thread group identifier* in turn uniquely identifies the thread group each active thread with a given PID belongs to. In a single threaded process, the only thread that exists is the main thread. Its TGID is the same as the its PID. The main thread, being the first thread of the process, is automatically assigned the role of the so-called *thread group leader*. In a multi-threaded process this still applies for the main thread. The other threads share the main thread's TGID but have a distinct PID assigned to them. Listing 2.10, for once, does not seem helpful in clarifying the circumstances of the identifiers in a multi-threaded process. The reason is that since the POSIX standard demands all threads in a multi-threaded process to have the *same* PID, their actual PID is stored in the `tgid` member of each thread's `struct task_struct` and the shared TGID is stored in the corresponding `pid` field. Consequently, the `ps` command which lists all processes and threads on the Linux command shell, shows the actual TGID under the column named PID and the individual PIDs of each thread under the column named LWP which stands for *Lightweight Process*. [27, 29]

```
# ps  -LA | grep 773
   PID     LWP     TTY         TIME CMD
   773     773     ?       00:30:25 NetworkManager
   773     816     ?       00:02:21 gmain
   773     817     ?       00:00:00 gdbus
```

Listing 2.10: The `ps` command listing the thread group #773. The NetworkManager thread is the main thread and thread group leader. LWP stands for Lightweight Process and denotes the actual thread's PID.

Things start getting more complex in situations where multiple so-called *namespaces* are required. This is the case in containerized environments where every container has its own set of identifiers that overlap with those of other namespaces. But since containers are isolated entities within a given root namespace there is no danger of confusion between the identifiers of the different namespaces. Still, other issues can arise which unfortunately go beyond the scope of this present thesis and will be not further considered. The Linux kernel, however, is well capable of handling multiple namespaces. [36]

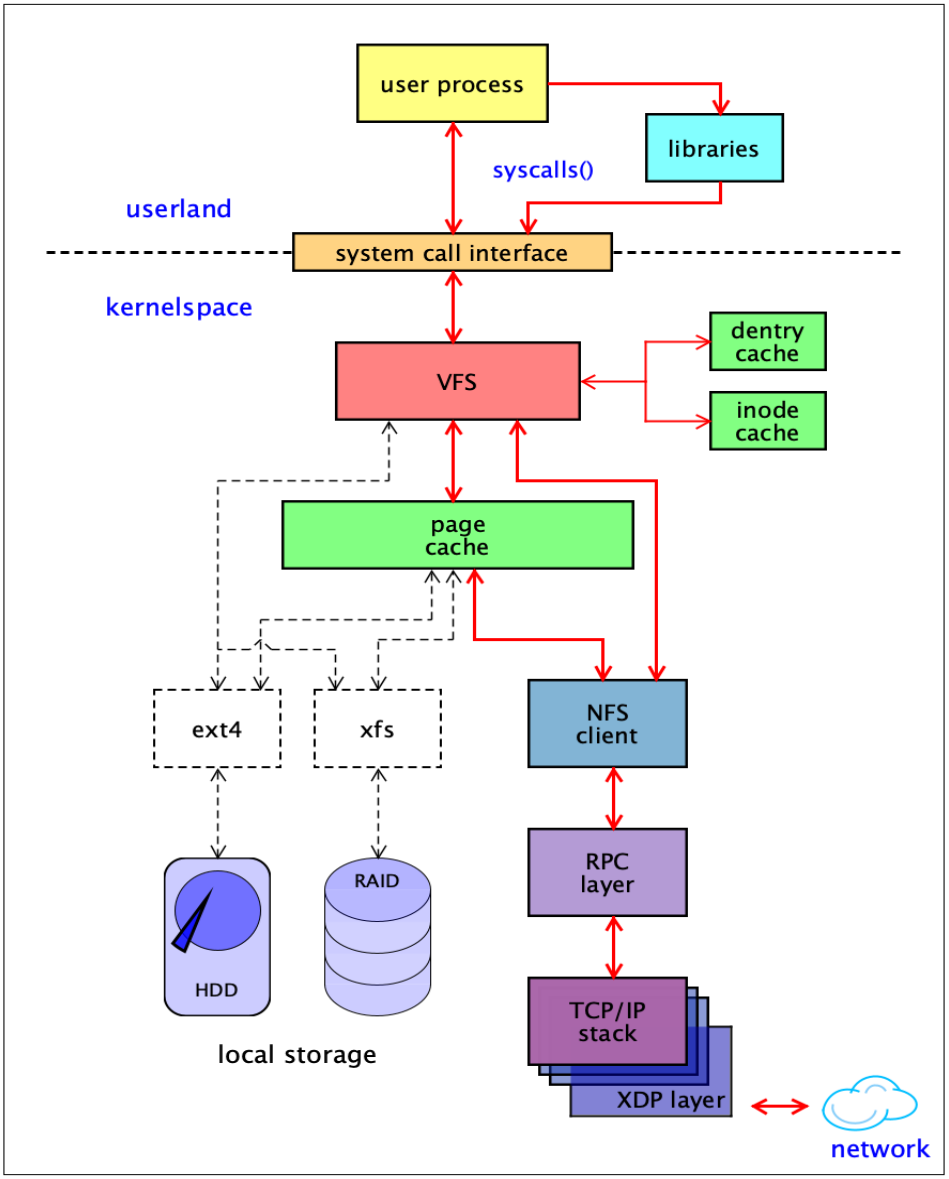

Figure 2.1: Main components involved in servicing an NFS file I/O operation on the client side

Interfacing Userland and Kernelspace

In order to understand the operating principles of eBPF with respect to file I/O, it is crucial to examine how the kernel deals with file I/O requests coming in from userland. File I/O operations include all routines operating on files and manipulating their contents. Fundamental operations perform tasks like creating a new file, opening an existing file, either stored on disk, in memory or remotely, retrieve or assign file attributes or so-called metadata, read from and write to a file and, of course, close the file or remove it. And since the old paradigm that in Unix «*Everything* Is A File» [36] still holds true, file I/O operations apply to numerous operating system objects such as regular files, pipes, directories, sockets and device files, also known as special files. [28] Flawless file I/O is a crucial aspect in all digitalized domains, including the one of experimental physics, where files hold the precious experimental outcomes. Therefore, the mechanisms and requirements of file I/O handling from userland to kernelspace are described in the next sections. Therein, special focus lies on the interplay of the kernel subsystems involved in file I/O with the built-in network filesystem client which is extensively used for file transfers between computing worker nodes and dCache storage nodes in the HTC environments at DESY.

2.4 The System Call Interface

Considering the fact that userland applications commonly do not have the privileges to interact with hardware components directly, they have to call into the kernel to request the system resources needed to perform the computation they were written for. The mechanism to perform this task is by issuing a *system call* (also called *syscall* in kernel developer parlance) into the kernel. In most cases the syscall issued by the application is a wrapper around the actual userland syscall code. Thus it does not directly call into the kernel, but, as shown in Figure 2.1, takes a detour into a standard library such as e.g. *libc*. [28] The first boundary landmark the library syscall encounters on its way into the kernel is the system call interface. This interface permutes the userland syscall to a generic kernel syscall that the underlying kernel layers, such as the virtual filesystem switch (more in Section 2.5), will understand. In accordance with the definition of an interface being a collection of function declarations in contrast to their concrete implementation, the function permutation that the syscall interface performs, helps to avoid mingling userland with kernel function code, whilst maintaining flexibility in the kernel's syscall implementations [28]. In this context, the utilization of an interface at the boundary between userland and kernelspace contributes to maintain a kernel

developer's paradigm called «Do Not Break User Space!» [37, 38]. Evidently, the goal is to be able to make frequent and numerous improvements to kernel code in an ever-accelerating release process without affecting the operability of userland programs.
It is worth mentioning the similarity of the interface usage approach in the Linux kernel to common design patterns in object-oriented programming languages as discussed in Neil Brown's article about object-oriented (OOP) design patterns in the Linux kernel [39]. In OOP an interface contains function prototypes that an object of the implementing class must be able to call to validly represent its declared type [40]. The kernel displays this concept in its virtual filesystem layer (see also Section 2.5). This layer acts as a uniform interface, which contains file I/O operations as function prototypes that every interfacing filesystem must implement to be considered interoperable. Erich Gamma et al. expressed this in their design principle called «program to an interface, not an implementation» [41], which by implication means that the interfaces must *not* be changed.

> «Since changing interfaces breaks clients you should consider them as immutable once you've published them.[...] Once you depend on interfaces only, you're decoupled from the implementation. That means the implementation can vary, and that's a healthy dependency relationship.[...]» [42]

More similarities to OOP design patterns will be pointed out in the further course of the description.

2.5 The Virtual Filesystem Switch (VFS)

Once the syscall has made the transition into the kernel, it hits the *virtual filesystem switch* layer. This layer is often simply referred to as *virtual filesystem*. Although unreflected in its name, the term virtual filesystem points to the fact that it acts as an interface to numerous filesystem implementations [43]. It interacts with them through a common set of functions. Without, however, being a *real* filesystem itself, but rather a *virtual* one. Looking at it from another angle, it can also be seen as a decision node or a demultiplexer, receiving one syscall as input and deciding on a set of function arguments, which of the many specific filesystem implementations (outputs of the demultiplexer) to delegate the call to. The latter view is emphasized by the term *virtual filesystem switch*, which simultaneously avoids a name clash with filesystems like procfs, sysfs, devfs and many other virtual (or pseudo) filesystems, that solely exist in memory. Both ways of looking at the VFS are equally valid. Additionally, there is a second crucial aspect to the purpose of the VFS layer. File I/O operations like reads and

writes of file data stored on storage devices like hard disks or tape machines, especially those to remote ones, are slow and costly for obvious reasons. As an intermediate layer designed to facilitate and accelerate the access to the requested file data, the VFS therefore also queries different *caches* for a potentially faster retrieval of recently used file metadata and contents (see also Figure 2.1). In case of cache hits, that is the presence of the requested data in the cache, the call chain towards the corresponding filesystem is discontinued, as long as the file I/O request can be serviced out of the cache directly. The caches, upon which the VFS heavily relies, are namely the page cache, the inode cache and the directory entry cache, or shortly *dcache*. Not to be confused with the distributed storage system management software *dCache* mentioned earlier. The inode cache and the dcache both store VFS objects that are part of the kernel's common file model, which will briefly be described in the following section.

2.6 The Common File Model

In the same way as the VFS requires the underlying filesystems to implement a common set of file I/O routines to ensure interoperability, it also enforces the exposure of a number of filesystem objects. This has to be fulfilled in such a way, that both, VFS and filesystems, can agree on these representations as being structured according to the specification of the kernel's common file model. As W. Mauerer states in his book about the architecture of the Linux kernel, this common file model is a «structure model consisting of all components that mirror a [...] filesystem» [36]. The common file model comprises the following components:

- filesystem type objects
- superblock objects
- dentry objects (common abbr. for *directory entry* objects)
- inode objects (common abbr. for *index node* objects)
- file objects

All of these objects are software constructs that, just like the *classes* found in object-oriented design, define fields that hold state information about the object and routines that define which actions an object is able to perform [27].

2.6.1 Filesystem Type Objects

Although most literature about the Linux kernel does not list it as a member of the group of common file model objects, the filesystem type object plays a crucial role in registration and identification of all filesystem types that the kernel maintains. The VFS actually enforces filesystem implementations to define a data structure called `struct file_system_type` declared as a prototype in `/include/linux/fs.h`. As the name states, the filesystem type object holds information about the *type* of a specific filesystem that the kernel is already aware of. That is either because the module containing the filesystem has been loaded into kernel memory or it has been already mounted.

```
1  struct file_system_type {
2    const char *name;
3    ...
4    int (*init_fs_context)(struct fs_context *);
5    const struct fs_parameter_spec *parameters;
6    ...
7    void (*kill_sb) (struct super_block *);
8    struct module *owner;
9    struct file_system_type * next;
10   struct hlist_head fs_supers;
11   ...
12 };
```

Listing 2.11: Excerpt from the `struct file_system_type` as declared in `/include/linux/fs.h`

Listing 2.11 shows an excerpt from the `struct file_system_type` revealing most of its contents. Line 2 lists the declaration of a field called `name` which is used as an identifier for the specific type of filesystem such as e.g. `nfs4`, `ext4` or `btrfs` whenever it is mounted. By means of this identifier and its length, it is found in the singly linked list of pointers to filesystem type structs. As will be shown in an example further below in this section, this list contains pointers to all filesystem type structs that have been already registered by the VFS. The `next` field of type `struct file_system_type` pointer in line 9 of Listing 2.11 points to the next filesystem type entry in this list. The corresponding routines performing tasks such as `register_filesystem()`, `find_filesystem()` and `get_fs_type()` are all defined in `/fs/filesystems.c` of the kernel sources.
Furthermore, in its function as a uniform interface for all filesystem types, the VFS declares function pointer prototypes within its filesystem type object. Lines 4 and 7 of Listing 2.11 show two function pointers declarations that will be assigned a memory address of a filesystem

specific function implementation in the course of the registration procedure. The `struct fs_parameter_spec` serves the purpose of storing the types of all potential parameters associated with a specific filesystem. For NFS, this long list of parameters is defined in `/fs/nfs/fs_context.c` as an array called `nfs_fs_parameters` of type `struct fs_parameter_spec`. The module owner in line 8 of Listing 2.11 holds a reference to a list of all modules known to the kernel, the module's state and the name of the module, respectively.
The `fs_supers` field in line 10 of Listing 2.11 holds a reference to the head of a list containing all superblocks (see Section 2.6.2) of filesystems belonging to one type only. This list is maintained by the kernel to keep track of the many filesystem instances of the same filesystem type that are mounted simultaneously. [36]

But how and at which point does the VFS become aware of the `nfs4` filesystem type? To understand this, the filesystem registration procedure has to be examined in more detail. Before the `nfsv4` module can be loaded into the kernel, three other modules have to be loaded first. Namely the `dns_resolver`, `sunrpc` and `nfs` module. The order in which they are listed here is the same order in which they are loaded into kernel memory as a cause of their interdependence. Listing 2.12 shows the verification of this statement by invoking the `modinfo` command on the Linux command shell.

```
$ modinfo nfsv4
...
depends:        nfs,sunrpc,dns_resolver
...
```

Listing 2.12: Invocation of `modinfo` command to verify module interdependence

Repeating the above procedure by invoking `modinfo` for each of the three modules that `nfsv4` depends on, finally reveals their loading sequence mentioned above. An examination of the kernel code shows that the `nfs` module performs the actual registration with the VFS as described in the following.

Whenever the `nfs` module needs to be loaded into kernel memory, the routine `module_init(init_nfs_fs)` is invoked. The routine `init_nfs_fs()` itself causes both, the `nfs` *and* the `nfs4` filesystem types to be registered. At the same time it invokes functions that initialize caches such as the page cache and the inode cache, already introduced in Section 2.5, reserved only for the NFS filesystem. The registration proceeds within the body of a routine called `register_nfs_fs()` defined in `/fs/nfs/super.c`. Focusing on the registration of the `nfs4` filesystem type only, from this point on, the call to the

generic function `register_filesystem()` is the next link of the call chain. This `register_filesystem()` routine is invoked with the memory address of a variable named `nfs4_fs_type` of type `struct file_system_type` passed as an argument (declared in `/fs/nfs/nfs4_fs.h`). Listing 2.13 shows how the `nfs4_fs_type` variable is filled with NFSv4 specific content. Part of this content is the hard-coded string `nfs4`, that is assigned to the filesystem type's `name` field in line 3. Moreover, memory addresses of NFS specific function implementations are assigned to the `init_fs_context` and `kill_sb` function pointers as expected by the VFS specification in `struct file_system_type` of Listing 2.11. The former will be invoked as the very first NFS specific code during the mount procedure as demonstrated in the example given in Appendix A Section A.3.1.

```
1  struct file_system_type nfs4_fs_type = {
2    .owner              = THIS_MODULE,
3    .name               = "nfs4",
4    .init_fs_context    = nfs_init_fs_context,
5    .parameters         = nfs_fs_parameters,
6    .kill_sb            = nfs_kill_super,
7    ...
8  };
```

Listing 2.13: The definition of `nfs4_fs_type` of type `struct file_system_type` in `/fs/nfs/fs_context.c` containing the first NFS specific routine (in line 4) invoked during the NFS mount process

Two steps are left to complete the registration procedure of the `nfs4` filesystem type with the VFS.
First, after sanity checking its argument the stated `register_filesystem(&nfs4_fs_type)` function invokes the `find_filesystem()` routine. The latter call does not exactly do what the name promises. It rather *tries* to find or *searches for* the filesystem type `name` in the list of filesystem types that the VFS already knows about. Thus, it either returns a valid pointer to an already registered filesystem type, *or* it returns NULL, indicating that no entry of that name was found in the list. Only the latter case induces the actual registration of the `struct file_system_type` named `nfs4_fs_type` by adding it to the list of registered filesystem types.

Secondly, at some point, the `module_init(init_nfs_v4)` routine, defined in `/fs/nfs/nfs4super.c` is called for the activation of the `nfs4` module which was not possible prior to the complete loading of the `nfs` module. As both filesystem types are already registered at this point, the `init_nfs_v4` routine initializes a DNS resolver, required for

resolving a hostname into a corresponding IP address, ensuring that DNS resolution functionality is available to NFSv4+ clients before they start mounting NFS shares. In addition, a DNS entry cache for faster name-address resolutions among other NFSv4.2-related cache allocations in kernel memory are performed.
In conclusion, this exercise of following the function call graph of the filesystem type registration has revealed the mechanisms that are utilized by the VFS to make filesystem types available and ready to be mounted in case userland syscalls request them. More specifically, what was demonstrated by looking closer at the filesystem registration process is the way the VFS takes care of assigning the correct filesystem function implementations to the prototypes provided and invoked by the VFS to delegate control to the specific filesystem in charge. This function pointer assignment technique is frequently encountered throughout the design of the various kernel subsystems. [29]

The level of detail with which these mechanisms are reproduced here reflects the level of knowledge necessary for developing effective eBPF programs. Only by following along and thereby understanding the interplay of the VFS, NFS and RPC layer it becomes possible to find the appropriate kernel routine to attach an eBPF probe to. An appropriate kernel routine, with respect to the intended query, is defined by the relevance of the metrics contained therein as well as its position in the sequence of the call graph (see Section 4.2.1 for more).

2.6.2 Superblock Objects

The object which is used by the kernel in order to describe the characteristics of a mounted filesystem is called *superblock*. It contains all the metadata and function prototypes associated with a specific filesystem instance. For a locally mounted, disk-based filesystem like the Linux native ext4, the *filesystem control block* represents a superblock on a physical storage device which is read during the mount process. All relevant metadata therein are copied into a structure in kernel memory called `struct super_block`. It stores information like the blocksize used by the storage device, the filesystem type, mount flags, the mountpoint dentry, a UUID, quota information, a pointer to the default superblock manipulation routines declared as function pointers, a pointer to the default dentry function pointers and, most importantly, a list containing all inodes associated with the superblock. The above enumeration is by far not exhaustive. [27, 28]

In the general context of the development of the eBPF programs it suffices to keep in mind that the `struct super_block` maintains metadata that describe the associated filesystem.

More specifically, it is crucial to remember that a `struct file_system_type` field is contained in each superblock structure which can be accessed from inside an eBPF probe for filtering purposes (see Section 4.2.1). With regard to NFSv4, a more detailed description of how the superblock is organized in the kernel is given in Appendix A.3.

2.6.3 Inode Objects

Similar to how superblocks store the metadata of filesystems, inodes store the metadata of files. Inodes store metadata such as the type of a file, its access permissions, the file size, last modification or access times as well as owner credentials in the form of a user identifier (UID) and a group identifier (GID). As a blueprint for an inode object the VFS layer declares a data structure called `struct inode`. Defined in `include/linux/fs.h`, this data structure contains fields holding file metadata as well as pointers to associated data structures. A unique 64-bit identification number, generally referred to as inode number, is contained in `struct inode` as well. Additionally, it contains a reference to the superblock of the filesystem it belongs to as well as a reference to the address space of the actual file data. Lastly, but not exhaustively, the VFS inode contains pointers to a data structure called `struct inode_operations` and interestingly also to one called `struct file_operations`. The latter is assigned to the inode when it is first allocated. The file operation structure is passed on to an associated `struct file` later in the inode's life-cycle (see also Section 2.6.5). Oddly, an inode contains everything there is to know about a file, except its name. Since file names are subject to change, their administration lies within the responsibility of the dentry object discussed in Section 2.6.4. [36] Figure 2.2 shows the interrelationships between `struct inode` and other data structures of the kernel's file and filesystem management for orientation.

As the VFS demands it, all supported filesystem types are required to have their very own compatible definition of an inode data structure. So does the network filesystem. The NFS representation of an inode is called `struct nfs_inode` defined in `/include/linux/nfs_fs.h`. It embeds a complete VFS `struct inode` in a field named `vfs_inode` and declares two NFS specific file identifiers, namely a 64-bit `fileid`, which corresponds to the VFS inode number, and secondly, a field `fh` of type `struct nfs_fh`, where *fh* abbreviates the term *filehandle* in this context. The `struct nfs_fh` is a small data structure defined in `/include/linux/nfs.h` that solely contains the filehandle as an array of type character and the filehandle size, which is defined as the macro `NFS_MAXFHSIZE`

and limited to 128. For an in-depth examination of the organization of inode structures, with respect to the NFS layer, refer to Appendix A.4.

2.6.4 Dentry Objects

Dentry and inode objects are tightly coupled constructs. As mentioned in Section 2.6.3, the inode object maintains every metadatum about a file, except its name, since that is subject to change and thus not unique. There can be many file names referring to one inode. Nonetheless, an inode lookup with a given path to a file must be efficiently practicable for the VFS layer. That is where directory entries or short dentries come into play. By providing the file name, dentries are a link between an filesystem object (file or directory) and its associated inode [36]. The Linux VFS layer declares a `struct dentry` in `/include/linux/dcache.h` that contains a reference to `struct super_block`, which in turn contains a reference to a `struct dentry` called `s_root`. As the name states, this dentry is the root directory of the filesystem administered by the superblock to which a given dentry belongs.[27]

Every given dentry holds references to its parent, to a list of its siblings and a list of its own children. With regard to a path lookup, there are two important pointers in `struct dentry`. One points to a `struct inode` and the other points to a `struct qstr` called `d_name` that stores the name of the path component this dentry is responsible for. A path component can be any string literal that follows specific naming conventions, of course. Thus, a path name is an assembly of string literals separated by forward slashes, plus, if present, the initial forward slash, which always stands for the root of the namespace tree. All of these path name components refer to directories, while the final component, if present, does not necessarily. The final component is the looked for resource, be it a directory or a file, and marks the end of the path lookup that the VFS performs to finally get hold of the associated inode. [44]

When the VFS is unfamiliar with the path components passed by a system call, it queries the underlying filesystem for more information and allocates the dentries and inodes in kernel memory *on the fly*. As mentioned earlier in Section 2.5, the memory regions allocated for this purpose are referred to as dcache and inode cache. Dentries actually only exist for performance reasons [45]. They increase the path lookup speed by reducing the necessity of querying the underlying filesystems that are always time consuming to service, regardless of the fact whether the storage is attached locally or remotely. As opposed to inodes, the

Linux sources of the NFSv4 module do not implement any dentry representation specific to the NFSv4 filesystem, but use exactly the same `struct dentry` the VFS exposes to all of its known filesystems. The dentries used by NFSv4 kernel code hold a reference to a field called `vfs_inode` of type `struct inode` embedded within the filesystem specific `struct nfs_inode` data structure, already known from Section 2.6.3. Appendix A.5 provides a more detailed view on the role of dentry object during the mount process of a NFS share.

2.6.5 File Objects

The last object of the common file model discussed here is the file object. A file object represents a file opened by a process [45]. Its VFS representation is the `struct file` declared in `/include/linux/fs.h`. It stores a reference to its associated inode and to a `struct path` that includes a dentry with the file's name. A field called `f_pos` stores the byte offset from the start of the file to the current position used for subsequent file operations. Owner credentials, a reference count and file mode are also part of the contents of the `struct file`. It does not include the file data itself or any metadata. Both are the domain of the associated inode, which holds references to the file data memory locations as well as to its metadata.

When a new file is opened by a process issuing an open syscall, a new `struct file` is allocated in a file cache with the help of the `alloc_empty_file()` routine defined in `fs/file_table.c`. After successful allocation in the kernel's file cache, toward the end of the function body of the `do_open()` routine defined in `fs/namei.c`, the `vfs_open()` function invokes a `do_dentry_open()` routine, both defined in `/fs/open.c`. This is the point, where the newborn file structure is filled with values, such as the associated inode to the `f_inode` field plus a set of function pointers to the `struct file_operations` field. The function pointers are provided by the associated inode, which holds a reference to those routines for this very moment.
The NFSv4+ specific implementation of the `open` file operation is a routine named `nfs4_file_open()`, defined in `/fs/nfs/nfs4file.c`. It is immediately invoked once the `struct file_operations` field has been assigned. It causes the allocation of a `struct nfs_open_contex` (`/fs/nfs/inode.c`), which contains fields related to the process of opening a file in NFSv4+, such as NFS lock contexts, NFSv4+ state information (see also Section 2.8 for details), file credentials, and the associated dentry. Once the `struct nfs_open_contex` is allocated, it is assigned to the `private_data` field of the file structure.

A following call to the `nfs_inode_attach_open_context()` routine causes the final embedding of the associated VFS inode into the `struct nfs_inode`, which creates the link between the regular VFS file object and the NFS inode. This association remains valid until the file is closed. In the latter case, once the file's reference count drops to zero the associated file structure is deallocated and all NFSv4+ related resources are released from kernel memory.

2.7 The File Description Table

In order for the kernel to keep track of the files that are currently opened by a given process, each process descriptor maintains a pointer to a data structure of type `struct files_struct` simply called `files`. The `struct files_struct` is declared in `/include/linux/fdtable.h`. To arrive at the actual array of open files, a field called `fdt` of type `struct fdtable` pointer has to be dereferenced. Another pointer named `fd` within `struct fdtable` points to the actual array of open files, also referred to as *open file description table*. The array is stored with a bitmap called `open_fds` containing the currently used array indices. As expected, the `fd` table is of type `struct file` double-pointer. This means, `fd` points to an array containing pointers to file structures. In other words, accessing the file description table at a given index returns a pointer to a `struct file` currently in use by the corresponding process. The indices to the file description table are of type integer and commonly referred to as file descriptors. Hence the name of the table. Figure 2.2 illustrates the interrelationships between data structures involved in the kernel's file and filesystem management including the open file description table.

When a userland process requests a file to be opened by issuing a corresponding syscall, e.g. `open() or socket()`, the kernel associates the smallest unused file descriptor found in the `open_fds` bitmap to a pointer to the just opened `struct file` and returns the file descriptor to the process as a handle to it. Each userland process has its own file description table as well as its own set of file descriptors. The latter will be subsequently used as an argument in every syscall the process issues to refer to the corresponding open file. Once the process completes its interaction with the opened file it issues a `close()` system call which causes the file descriptor to be erased from the `open_fds` bitmap. This declares the entry in the file description table as free and ready for the next open file association again. [27]

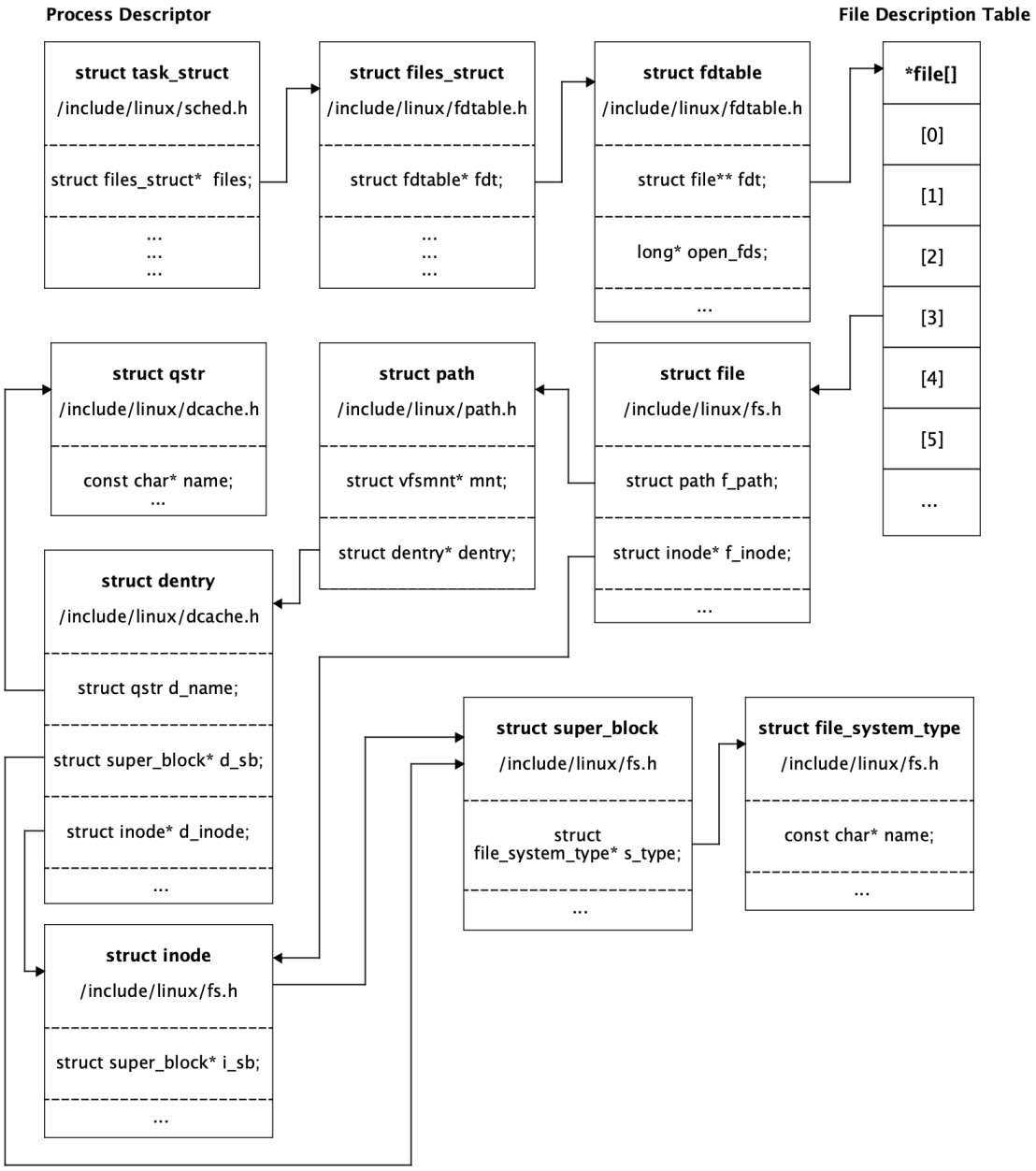

Figure 2.2: Overview of interrelationships between VFS objects and other data structures involved in the kernel's file and filesystem management (... indicate omissions)

2.8 The Network Filesystem Protocol Version 4+

Various mechanisms used by the Linux kernel to handle NFSv4+ requirements have been already introduced and illustrated with examples from the kernel source code in the previous sections. A short but more detailed introduction into critical features of the NFSv4+ protocol itself are presented in this section.

The development of the network filesystem predates the first release of the Linux kernel in 1991 by six years [46]. It was first proposed at Sun Microsystems back in 1985 [47]. Since its first appearance in the UNIX kernel [43], it has been ported to other operating systems in different versions. The latter of which has undergone a protocol redesign and extensive reimplementation, resulting in the current NFS version 4, with its minor version numbers 1 and 2. NFSv4 was standardized by the IETF in RFC 7530 in March 2015 [48] with supplementary protocol additions in RFC 8881 for NFSv4.1 [49] and in RFC 7862 for NFSv4.2 [50].

As the name states, NFS is a distributed filesystem which is shared across a network. It offers client access to files by mounting a remotely hosted filesystem onto a mount point of a local filesystem. While the NFS mount appears as a coalescent extension of the local filesystem tree, it hides all the network communication and file I/O details from the user. Since its first proposal [47] the NFS protocol standard utilizes the XDR standard as a machine independent representation of serialized data types and formats. The XDR standard, which is defined in the latest RFC 4506 [51] document, determines how data transmitted across a network is to be encoded such that a recipient is able to decode the same data without loss of meaning, independently from the hardware components and operating systems used on both machines. Additionally, the RPC mechanism, which is standardized through RFC 5531 [52], has been utilized by the NFS from the beginning until today. It has been updated by RFC 9289 introducing transport layer encryption of RPC requests in transit [53]. Being the carrier of the NFSv4+ payload[5], RPC allows a client to execute locally defined procedures on a remote machine and return the results thereof back to the client. Both, the XDR and the RPC mechanisms were also first developed by Sun Microsystems and are still referred to as *sunrpc* and *sunrpc/xdr* in the Linux kernel source code. The critical new features that the redesign of the NFS protocol introduced with version 4+ are described in the following subsections.

[5] See also Figure 2.1 for orientation

2.8.1 NFSv4+ Features

2.8.1.1 Unified Core Protocol

In contrast to previous versions, the NFSv4+ protocol standard is self-contained, meaning it is not dependent from ancillary protocols anymore. Those were needed to provide services necessary for mounting NFS shares, negotiating ports, handling locks or querying the status of the network communication participants with regard to network partitions or system crashes. In this sense, the presence of the previously used port manager for all these auxiliary services called `rpcbind` in Linux, the Lock Manager (NLM) protocol called `lockd`, the Network Status Monitor (NSM) RPC protocol called `rpc.statd` and a daemon called `mountd` necessary for client access permission handling during the mount process of NFS shares, have become obsolete by the 4+ version of the NFS protocol. [**?**] The effectuation of the mount process as well as locking state and mechanisms thereof have been incorporated into the core protocol instead. Locking state, specifically, «is maintained by the NFSv4+ server under a lease-based model» as stated in RFC 7530 [48].

2.8.1.2 Statefulness

Mentioning the term *state* in the previous paragraph already suggests that the NFSv4+ protocol is *stateful* in contrast to previous versions. In the context of locking, the server grants a lock in concert with a lease to the client and preserves the locking state even across network partitions or client machine reboots. If the client, which has been granted a lock for a lease period, does not renew the lease before its expiration, the server may release all states associated with the client's lease. But locking is not the only state related topic. State can also be expressed by identifiers such as the `clientid`, which uniquely identifies a client during a NFSv4.1+ session. It is generated by the NFSv4.1+ server. In one of the primary handshake operations called EXCHANGE_ID, which plays a crucial role in the initiation of a valid NFSv4.1+ session, the client transmits two identifiers in a `eia_clientowner` field of the payload. According to RFC 8881 [49] this field contains a client-owner-verifier (abbreviated as `co_verifier` in the following) and a client-owner-identifier (abbreviated as `co_ownerid`[6]), respectively.
In order to make this more tangible, the relevant NFSv4.1 handshakes during the mount of the

[6] Not to be confused with the `clientid` generated by the server for a given `co_ownerid`. After the primary handshake the `clientid` is used for subsequent client-server communication.

NFS share described in Section A.2 will exemplarily be looked into at a deeper level. Figure 2.3 shows the entries in the `eia_clientowner` field of the EXCHANGE_ID compound procedure sent by the NFSv4.1 client to the server.

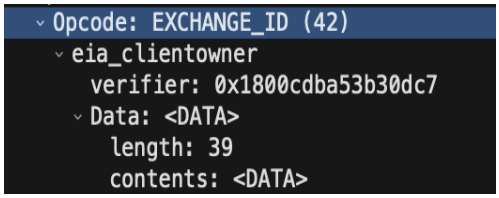

Figure 2.3: The `eia_clientowner` field contents of the client's EXCHANGE_ID call (all figures captured with `tshark` and dispayed with Wireshark Network Protocol Analyzer [1])

The verifier appears as a 64-bit numerical value (often boot time of the system [3]), while the `co_ownerid` is represented by a unique string composed of client-related information. Figure 2.4 displays the contents of the `Data` field, which is an alias for the `co_ownerid` field name used by Wireshark.

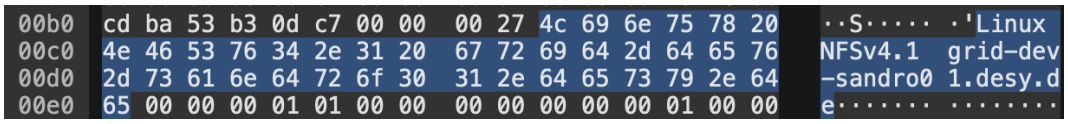

Figure 2.4: The contents of the `Data` alias `co_ownerid` field of the same client `EXCHANGE_ID` call as in Figure 2.3

In the given example, the NFSv4.1 client has chosen the string «Linux NFSv4.1» as a prefix to the hostname «grid-dev-sandro01.desy.de» of the client machine to be the unique `co_ownerid`. As a result of the `co_ownerid` value, the server constructs a unique `clientid` that is used to identify the NFSv4.1 client throughout the existence of a successfully established session. In contrast, the `co_verifier` is a verifier for a client incarnation, «allowing the NFSv4.1 server to distinguish successive incarnations of the same client after reboots» [49].

In its reply to the client's EXCHANGE_ID request, the server transmits the `clientid` as well as a unique `server_owner` identifier consisting of a numerical `so_minor_id` and an opaque `so_major_id` (shown in Figure 2.5 and Figure 2.6). The server acknowledges the successful receipt of the client's EXCHANGE_ID request with an NFS4_OK status message and returns the constructed `clientid` (see Figure 2.5).

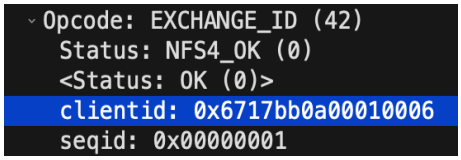

Figure 2.5: The `clientid` constructed by the server and returned in its EXCHANGE_ID reply

In the same compound EXCHANGE_ID procedure the server determines the `so_minor_id` to be 0 and the `so_major_id` to be the server's own socket address, namely «$131.169.223.60.8.1$» (see Figure 2.6).

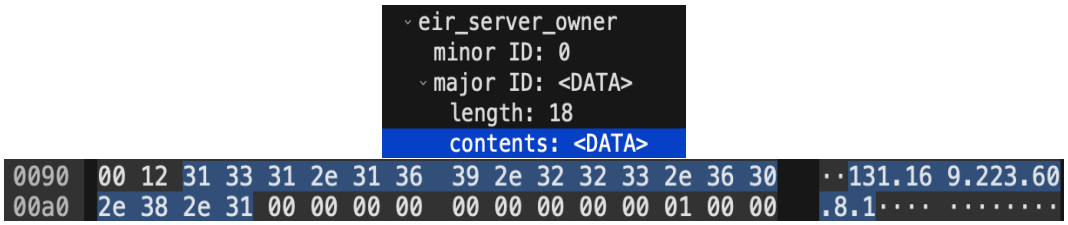

Figure 2.6: The `eir_server_owner` field contents of the server's `EXCHANGE_ID` reply

In IPv4, a socket address is conventionally notated as a tuple of the form:

$$< IPv4\,address >:< port\,number >$$

The corresponding port number of the *uaddr* IPv4 format notation used in the EXCHANGE_ID reply (defined in RFC 5665 [54]) can be decoded into conventional notation by performing an 8-bit left shift with the penultimate value of the socket address (here: 8) and adding the last value (here: 1) to it:

$$portnumber = (8 << 8) + 1 = 2048 + 1 = 2049$$

While port number 2049 is the official assignment for the NFS protocol registered by the IANA as reflected in [55], the socket address now resolves to «$131.169.223.60{:}2049$» in conventional IPv4 notation.
At this point of the network handshake proceedings, both NFSv4.1 server and client have exchanged unique identifiers that aim at preserving state even when in recovery mode after failures. Apart from the client and server owner IDs, there are other state enabling IDs such as the `sequenceid`, the `slotid`, the `fsid`, the `stateid` and the `sessionid`, only

to name a few. Describing them all would go beyond the scope of this thesis, but since the concept of state enabling NFSv4.1+ *sessions* encompassing the `sessionid` is a feature introduced in version 4 minor number 1, it will be briefly looked at in Section 2.8.1.3.

2.8.1.3 Sessions

According to RFC 8881 [49], the introduction of sessions to the NFSv4.1 protocol has paved the way for the support of the Exactly Once Semantics (EOS). The RFC 8881 document states that each compound procedure sent with a leading SEQUENCE operation «must be executed by the receiver exactly once». These leading SEQUENCE operations are exchanged between the NFSv4.1+ client and server in every operation following a successful CREATE_-SESSION handshake. This will be demonstrated by resuming the study of NFSv4.1 network communication, where it was left off in the previous section.
After completion of the EXCHANGE_ID handshake, the next operation is the request issued by the NFSv4.1+ client to establish a session with the server. This is accomplished with the help of the CREATE_SESSION operation, which is sent by the client providing its newly acquired `clientid` and the first sequence identifier called `seqid`. The latter was first sent by the server in its reply to EXCHANGE_ID and is depicted in Figure 2.5 below the value of the `clientid`. The `seqid` is incremented with every mutually exchanged and successfully completed compound procedure. The server's reply to the CREATE_SESSION request contains a session identifier called `sessionid` in the form of a 128-bit numerical value. The `sessionid` generated by the NFSv4.1 server in the example looks like the following:

session ID: 6717bb0a0001000600000000000000001

Taking a closer look at the `sessionid` reveals that with a NFSv4.1 server implementation as used by the dCache storage system, the most significant 64-bits match perfectly with the `clientid`. However, as already mentioned, from this point on, the two participants will exchange compound procedures with a leading SEQUENCE operation at the top level of their operation stack. Inside the SEQUENCE operation a 3-tuple of state-enabling values is contained. These are called `sessionid`, `seqid` and `slotid`, respectively. The `slotid` serves as an identifier for a so-called slot maintained in the slot table associated to a NFSv4.1+ session. A slot is an intermediate memory location for a request to be sent and a reply to be received. The replier will cache its reply in case it receives a request with a `seqid` equal to the one the replier already replied to. It will recognize the request as duplicate and re-send the cached reply. In this way, the session slot is used as a reply cache. Thus, by providing

an updated 3-tuple with every procedure, the participants can undoubtedly validate if a reply matches a request utilizing the `slotid` and can furthermore use the `seqid` «for a critical check to determine whether a request with a given `slotid` is a retransmit or a new never-before-seen request». EOS itself is made possible by using the reply cache for the 3-tuple values that can be kept in persistent storage. This provides EOS even through server failure and recovery. [49]

2.8.1.4 Compound Procedures and Callbacks

Prior to the redesign of the NFS protocol in version 4, each operation was sent in a distinct RPC request. Issuing the NFSv3 equivalent operations of an e.g. OPEN request followed by a GETATTR and a READ operation, concluded by a second GETATTR and CLOSE operation would have caused at least five RPC requests to be transmitted and their distinct replies to be received. Evidently, this verbosity increases latency due to the summation of round trip times caused by the transmission of the five distinct operations [49]. With the introduction of *compound procedures* an attempt was made to counter these shortcomings. Since NFS version 4 minor version 0, the protocol allows multiple operations to be packed and transmitted in a single RPC request. This does not only reduce the total number of RPC requests that have to be transmitted over the network, but also ensures that state enabling operations like SEQUENCE can be added to every compound procedure as a leading operation. This enables the association of the operations inside the compound to a specific session and the corresponding session slots. According to RFC 8881 [49] the recipient of a RPC, be it a request or a reply, must evaluate the sequence of operations inside a compound procedure *in order*. The evaluation ends abruptly at the first failure encountered in the processing chain and an appropriate NFSv4+ error flag is returned instead of a positive confirmation like NFS4_OK in the case an operation succeeds. The kernel code examples presented in the Appendix A in Section A.6 show how compound procedures are composed and prepared to be XDR encoded prior to their transmission across the network.

A second feature new to NFSv4+ is the utilization of *callback* operations. Due to its stateful nature, the version 4 protocol introduces the possibility that in addition to the standard fore channel on which the client sends RPC requests to the server and receives its replies, a backchannel can optionally be established. This enables the server to commence communication on its own initiative. As a result, the NFSv4+ server is able to revoke locks held by the client or recall delegated server responsibilities from the client at any time by transmitting a compound callback procedure containing operations like CB_SEQUENCE and CB_RECALL. [49]

2.8.1.5 Parallel NFS

NFS environments with high file data access rates that run on protocol versions prior to NFSv4.1 suffer from throughput losses due to the bottleneck caused by the NFS server. In such a scenario, the data transfer from the storage devices toward all requesting NFS clients has to be led through the server machine. To make matters worse, this happens on top of the administrative communication the server has to maintain with all clients [56]. The introduction of the *optional* parallel NFS (pNFS) feature to the NFSv4 minor version 1 protocol has led to significantly better file access performance as it allows «direct client access to the storage devices containing the requested file data» [49]. Figure 2.7 shows a typical pNFS setup as it is also encountered in the HTC environment at DESY. The main components of a setup that use the pNFS feature comprise a node with an operational NFSv4.1+ client on the one end. On the other, the NFSv4.1+ servers can play the role of a metadata server (MDS) or the role of a data server (DS). It is also possible that one physical machine takes both roles. The server will communicate its role to the client by setting the EXCHGID4_-FLAG_USE_PNFS_MDS or the EXCHGID4_FLAG_USE_PNFS_DS flag or both flags in the reply to the EXCHANGE_ID operation, respectively. The NFSv4.1+ client will negotiate all administrative handshakes regarding its file operation request with the metadata server. The actual READ and WRITE operation will be serviced directly from the data servers on which the file data is stored. This can be performed by accessing multiple data servers in parallel. Hence the name of the parallel NFS feature.

The two main goals of pNFS are, first, to separate the metadata access path from the actual file data access path, avoiding one server to become a bottleneck by having to service both of them. And secondly, the data transfer performance is further enhanced by leveraging direct parallel access to the file data storage devices. Simultaneously, a control path exists between the MDS and the DS allowing for the management of «state required by the storage devices to perform client access control» [49]. This means, the MDS can enforce restrictions regarding authentication and authorization to the DS as well. The control path protocol, however, is not part of the NFSv4+ protocol itself and can include further management functionality other than the above-mentioned client access control.[57]

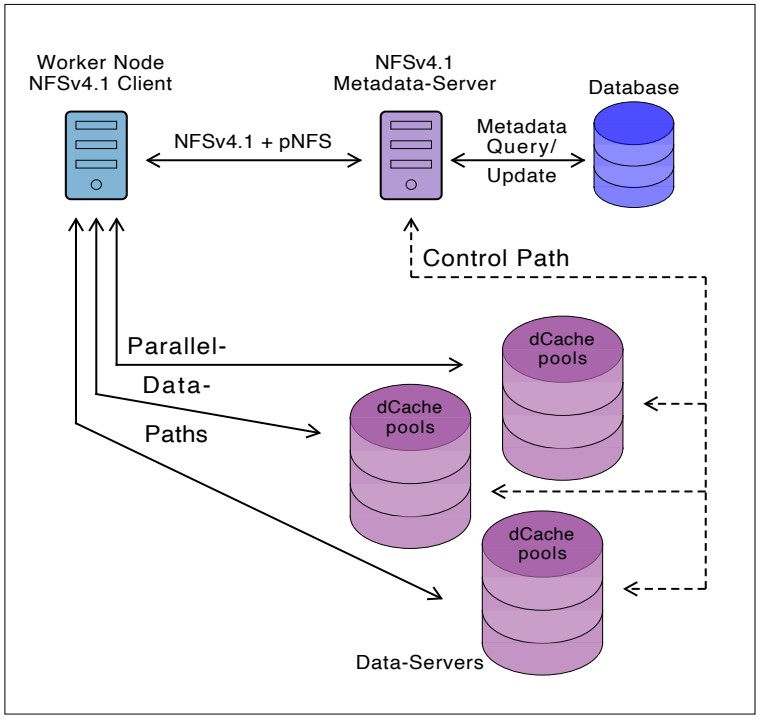

Figure 2.7: pNFS control and data flow diagram

In the HTC environment at DESY the MDS of the dCache storage relies on a *chimera* database to match an inode to a dCache-internal 36-byte long *chimera ID*[7] and to retrieve or update metadata about the associated file [59, 60].

The sequence diagram in Figure 2.8 illustrates all pNFS procedures exchanged between a NFSv4.1+ client and a MDS or a DS, respectively. It was drawn according to the sequence of NFSv4.1+ payload transfers captured on the wire with `tshark`. In this scenario, the client requests to open and read a specific file, which is stored in the directory tree of the mounted NFS share specified by the path the client provides. After the client and the MDS exchanged IDs, established a session, and negotiated state reclaims and security flavors successfully, the NFS filehandle of the root directory is provided by the MDS in its reply to PUTROOTFH. In order to obtain different sets of file attributes associated with the root filehandle, the client repeatedly transmits GETATTR operations. Starting from the root of the exported NFS share, the path to the file requested by the client reads as follows:

[7] Commonly referred to as *pnfsID* for historical reasons, since *pnfs* resolves to the name of chimera's predecessor called «Perfectly Normal Filesystem» [58]. However, for the sake of avoiding name clashes, in this work the less common name *chimera ID* will be used [59].

/pnfs/desy.de/dot/data_volatile/test_nfs42_8

In consequence, five repetitions of the ACCESS operation followed by a LOOKUP operation are needed to traverse all directory components until the final component (the filename in this case) is reached. Figure 2.8 depicts only the main operations contained in the compound procedures exchanged during this communication. This means that every shown LOOKUP operation is part of the compound procedure enveloping a SEQUENCE, PUTFH, LOOKUP, GETFH and a GETATTR operation. In this manner, the client obtains the NFS filehandle of every path component in concert with the associated metadata. With this information, the client is able to create a `struct nfs_inode` as described in Section 2.6.3 and store it in the inode cache for faster access if needed later on.

After having checked that permissions to access all components of the path are valid, the client transmits a request that contains an OPEN operation for the requested file and a pNFS specific operation called LAYOUTGET to the MDS. The concept of *layouts* is new to the NFSv4.1 protocol and solely used if the client and the MDS agreed on utilizing the pNFS feature initially. A layout is a set of parameters that defines the location of file data as well as how the file data is organized on one or multiple data servers. The client itself conveys a set of layout parameters to the MDS that it wishes to use when accessing the file data. These parameters include the *layout type*, which is supported by the client, an *I/O mode*, which indicates the type of I/O operation (read or write) the client wants to perform on the file. According to RFC 8881 a layout type can be a block, an object or a file layout type, which all contain different sets of parameters depending on the way file data are to be accessed on the DS [49]. The provision of a layout is associated with the maintenance of state as well. It therefore has to be verifiable through sequence IDs and a layout state ID, which are consequently present in every layout. Already granted layouts can be recalled by the MDS with the transmission of a CB_LAYOUTRECALL operation via the backchannel.

Returning to the sequence shown in Figure 2.8 the MDS confirms the client's parameters, adding a *device ID*, the filehandle of the requested file and a set of layout flags in its LAYOUTGET reply. The only layout flag set by the MDS in the captured sequence is FLAG_NO_IO_THRU_MDS. As the name suggests, it informs the client that it «should not send I/O operations to the metadata server» as stated in RFC 8435 [61], which is the document that defines an additional layout type called *flexfile layout*.
After agreeing on the layout to be used to access the file data on the DS, the client needs to be informed about the location of the file data. Thus, it requests this information by transmitting a GETDEVINFO operation to the MDS. This request contains the formerly received

device ID and the layout type to be used for access. The MDS replies by confirming the layout type and providing the socket address of the DS (in case file or flexfile layout types are used), to which the client has to connect to in order to access the file data. Network protocol type, NFS version, and read and write buffer sizes to be used with respect to the DS are transmitted from the MDS to the client as well.

The next steps the client performs in the communication sequence of Figure 2.8 are the establishment of a session with the DS, which it redirects to using the stated information from the GETDEVINFO reply received from the MDS. A subsequent transmission of a READ request toward the DS initializes the file data transfer. Once the file data transmission from the DS is completed, the client returns to communicate with the MDS. Since the provision of the layout, as mentioned above, is associated with state, a proper handover of the updated layout has to be performed by the client. This is achieved by transmitting a LAYOUTRETURN operation, in which the client indicates the completion of its request to the MDS. But since the MDS cannot be aware of any statistics about the transfers that took place between the client and the DS, the client populates the layout with a *I/O Stats* data structure containing metrics about the amount of bytes read, the read latency, the overall duration of the interaction and an error count. Additionally, the client returns information about the device ID and socket address of the DS it communicated with, the filehandle of the file read, and finally the layout state ID formerly granted with the layout itself. A subsequent CLOSE operation properly concludes the client-server interaction for the given file access and, by chance, this brief description of the pNFS feature incorporated in the NFSv4.1+ protocol.

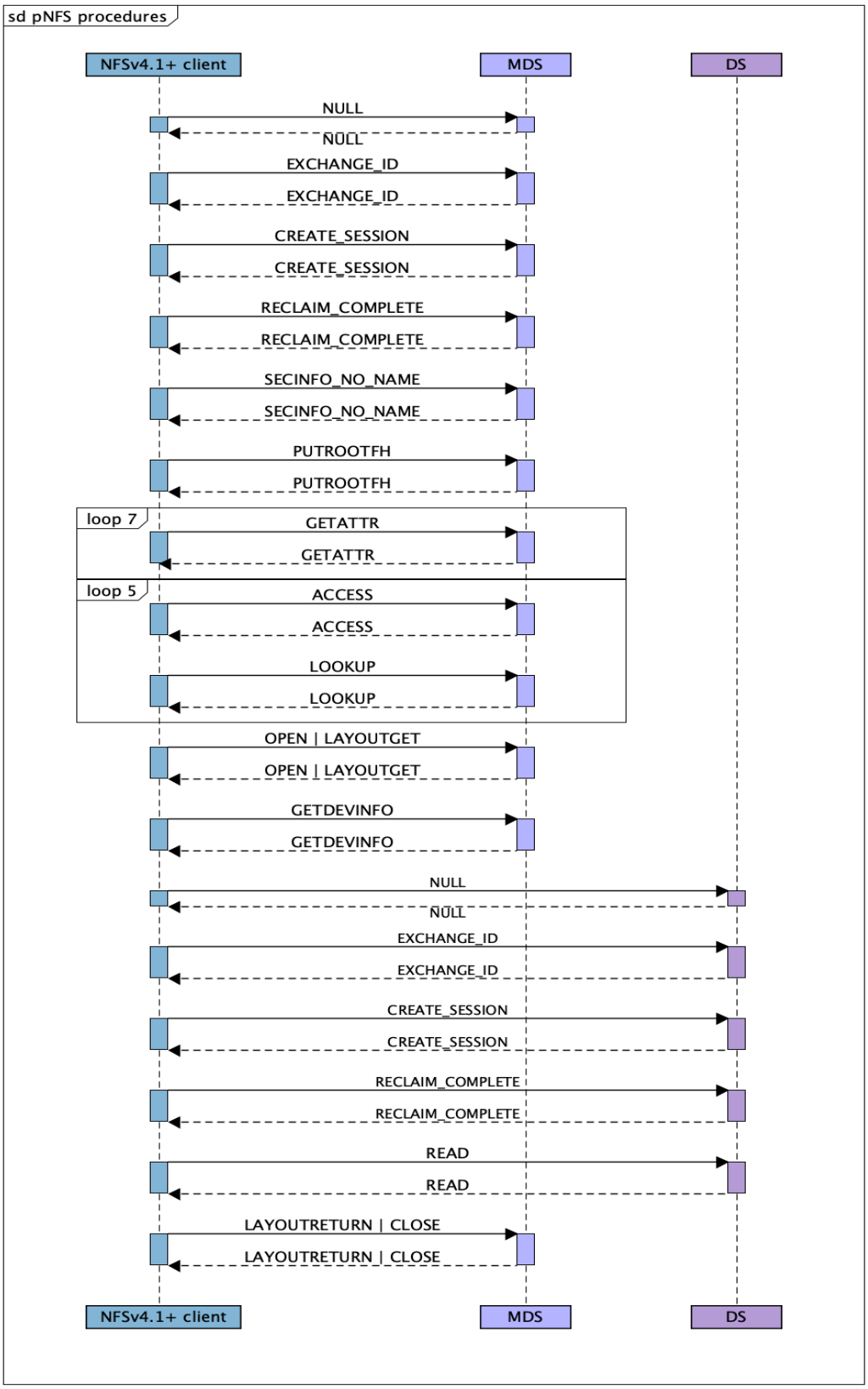

Figure 2.8: pNFS communication between NFSv4.1+ client, metadata server (MDS) and data server (DS) for a OPEN and READ file I/O operation

2.9 The dCache Storage System

When talking about the dCache storage system, it is reasonable to differentiate between the dCache storage managing *software* on the one hand and the assemblage of physical nodes running that software, forming the actual distributed *storage system* on the other. The development of the dCache software has undoubtedly been the consequence of a constantly growing demand for highly available mass storage space capable of handling data quantities at the petabyte scale. These huge amounts of scientific data are the outcome of numerous scientific experiments in the domain of high energy particle physics and photon science as mentioned initially in Section 1.1. Consequently, in the year 2000 the developers of the dCache storage idea expressed their vision in the following:

> «The goal of this project is to provide a system for storing and retrieving huge amounts of data, distributed among a large number of heterogeneous server nodes, under a single virtual filesystem tree with a variety of standard access methods. [62]»

Around the same time this vision was expressed, the development of the open source dCache software began as a joint effort of the two laboratories DESY and FNAL (Fermi National Accelerator Laboratory). The developing team was reinforced by developers from the Nordic Data Grid Facility (NDGF)[8], who joined the project in the year 2007 [63]. To a great extent (more than 95%) the dCache storage software is written in the Java programming language [64].

An important aspect of the dCache storage system is its ability to act as a data *disk cache* in front of a tertiary storage facility, e.g. magnetic tape. In fact, the flow of data typically begins with the ingress of newly gathered experiment data coming from one of the experimentation sites into dCache disk storage. From there, the freshly stored data is immediately flushed to tape media. At the time the group of scientists decides to analyze their data, it is staged back into the disk cache and *pinned* there for a determined amount of time. Once the file data is available on disk again, it can be requested by the scientist's jobs running on the compute nodes of the NAF or Grid via the NFSv4 or other supported access protocols. Since the staging of file data from tape media back to disk storage is significantly slower than the direct access from disk, dCache's caching mechanism allows for a faster access and transfer of experiment data to the worker nodes of the NAF or the Grid during periods of computation and analyses [65, 11].

[8] Nowadays named Nordic e-Infrastructure Collaboration (NeIC)

The architectural structure of the dCache storage software can be described as an agglomeration of microservices. Each distinct microservice is technically represented by a so-called *cell* in dCache nomenclature. They are contained in a *dCache domain*. Cells communicate by exchanging messages. This communication takes place even across domain boundaries using an address schema in the form of *cellName@domainName*. A single node can host one or more domains, since each domain technically represents a Java Virtual Machine (JVM). Distinct domains communicate with each other across cell tunnels by transmitting messages via TCP connections. The assemblage of all interconnected domains finally form a so-called *dCache instance*. An advantage implied by these architectural design choices is that a dCache instance can be easily scaled horizontally by simply adding new nodes, onto which domains of nodes under high load can be migrated. Otherwise, brand new domains can be created on those newly added nodes to balance the load as well. [60]

2.9.1 Main Components of a dCache Instance

The mandatory cell types that make up a minimal dCache instance are a *dCache door*, a *namespace provider*, a *storage pool*, a *poolmanager* and a coordination and discovery service. The latter service is needed for the auto-discovery of running cells and domains in the instance. It additionally performs the mapping of cells and domains to IP addresses of the physical nodes they are running on. As auto-discovery and coordination service dCache uses the *Apache Zookeeper* software, which provides configuration information and ensures coordination among the distributed domains [66].
The following three components of a dCache instance were already introduced in the context of the pNFS feature in Section 2.8.

2.9.2 An Entry Point to dCache

The dCache door is the entry point to a dCache instance, providing access to file data via «a variety of standard access methods», as stated at the end of the above mentioned quote about the initially desired dCache design features. Thus, a door converts protocol specific «instructions to a sequence of internal dCache message-based call sequences» [60]. The NFSv4.1+ protocol is only one of those data access methods used with dCache. Other supported protocols are Xrootd, WebDAV, FTP and DCAP, dCache's own access protocol. Figure 2.9[9] shows the share of transfers for each access protocol that occurred between the

[9] Figure kindly provided by T.Hartmann

NAF computing nodes and the dCache storage in August and September of 2024. While each door implements only one distinct access protocol, more doors for each supported protocol are typically present in a dCache instance for load balancing and high availability. For the NFSv4.1 access protocol the door service is the dCache representation of a pNFS metadata server.

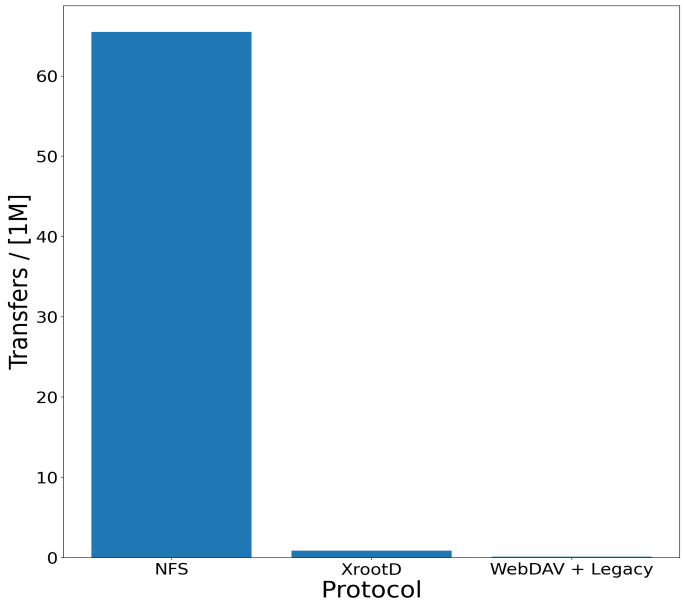

Figure 2.9: Share of per-protocol transfers between the NAF and the dCache storage system in August and September of 2024

2.9.3 The dCache Namespace Provider

The namespace provider used by dCache is the chimera database which maps inodes to chimera IDs and provides metadata about the requested files [59].

2.9.4 Storage Pools and the Poolmanager Service

A dCache storage pool or simply pool, is a logical storage location within the domain hosted on a data server. It is the actual container of the file data and is directly accessed by the clients (NFS or other). The dCache storage pools are administered by a service called poolmanager. This cell determines the way a file I/O request is handled. Depending on whether the file resides only on one pool, or on more pools as a replica, the poolmanager redirects the client

toward one specific pool or decides to choose one pool out of many according to a configurable load balancing policy. If the requested file does not reside on any pool but on tape media, the poolmanager triggers the file to be staged back to disk. Thus, the poolmanager and a sub-module called the *pool selection unit* are crucial components in the dCache control and data flow. [60]
There are two more dCache related aspects, which have to be mentioned before taking a closer look at which information about NFSv4+ transfers a dCache system administrator is able to retrieve from within the dCache system itself.

2.9.5 dCache Data Mover

The first aspect worth mentioning is that of a *mover*. A mover is the component within the dCache file data access flow that is prompted by the door to start servicing a file I/O request once an adequate pool has been selected by the pool selection unit. In turn the mover *pins* a file to the pool and thus saves it from eviction. Subsequently it guarantees the door to serve the file data to the client. In the case of NFSv4+, the mover is the actual thread pool that handles all read or write file I/O requests. The call from the door toward the mover is defined in the `NFSv41Door.java` file found following the path[10] inside the dCache codebase[11]. Therein, two private classes called `ReadTransfer` and `WriteTransfer` are contained, both defining a method called `selectDataServers()`, which in turn makes the very call to a method named `selectPoolAndStartMoverAsync()`. The important thing to note here is that the started mover does not silently stop by itself after completion of the file I/O request. But is rather «killed» by the door after the latter has been notified by a completion handler. Thus, once a mover is started, it lives as long as the file I/O request is not completed, independently from the time it takes to complete it or the fact whether the mover is actively transferring bytes or idling for any given reason. As a result, an accumulation of stale movers is possible if e.g. communication with the door fails to complete [67].
The second noteworthy aspect remaining is a file-related ID within dCache which already has been introduced superficially.

[10] dcache/modules/dcache-nfs/src/main/java/org/dcache/chimera/nfsv41/door/NFSv41Door.java
[11] Github Repository of the dCache codebase: https://github.com/dCache/dcache

2.9.6 dCache Internal File ID

The chimera ID is a 36-byte long string representation of a hexadecimal number. It uniquely identifies a given file in front of components inside the dCache storage system. It is constructed by performing multiple bit shifting operations on a randomly chosen UUID[12] interweaving the associated filesystem ID[13] (fsID). The result of the number-crunching algorithm found in the newID() and digits() methods in the InodeId.java file is stored as an array of bytes by performing a base 16 decoding of the string representation in a class of type *PnfsId*[14]. A typical chimera ID is shown in Listing 2.17.
Lastly, the dCache associated custom *nfs4j*[15] server and *oncrpc4j*[16] implementation are briefly introduced.

2.9.7 Java NFSv4+ Server and RPC Implementations Used by dCache

The NFSv4.1+ server used by the dCache storage software derives from a custom open source nfs4j implementation. The server itself is executed in concert with an equally open source custom implementation of the RPC protocol written in Java called oncrpc4j. The acronym ONC stands for Open Network Computing. In contrast to the Linux built-in NFS infrastructure, which is completely embedded within the kernel, the nfs4j server and the oncrpc4j RPC facility both run as userland applications in a JVM. Figure 2.10 shows the organization of the components involved in the handling of a NFS file I/O request on the dCache storage end, as opposed to the situation on the computing end shown in Figure 2.1. Just to be clear: Machines on both ends, the computing worker nodes end as well as the nodes on the dCache storage system end run Linux as an operating system.

[12] Found in dcache/modules/chimera/src/main/java/org/dcache/chimera/InodeId.java
[13] Currently hardcoded as zero.
[14] Found in dcache/modules/dcache-vehicles/src/main/java/diskCacheV111/util/PnfsId.java
[15] Java NFSv4+ server implementation: https://github.com/dCache/nfs4j
[16] Java ONCRPC implementation: https://github.com/dCache/oncrpc4j

2.9.8 The dCache admin interface

Before concluding the description of the main aspects related to the dCache storage system, a brief introduction to the so-called *dCache admin interface* is given in the following subsection. The dCache admin interface is a facility built into the dCache software that allows system administrators to interact with a running dCache instance via a custom command shell prompt. It is accessed by establishing a *secure shell* (ssh) connection to a dCache storage node.

```
1  [dcache-head-dot] (local)
2  sandro >
```

Listing 2.14: The command shell prompt after login to the admin interface

Once logged in, the system administrator has access to the admin interface command line prompt as shown in Listing 2.14. The identifier given within parentheses on the prompt always indicates the momentary *cellName@domainName* location the interface is connected to. The admin interface is not connected to any specific cell at this point, hence the `(local)` indication in the prompt of Listing 2.14. Any specific cell can be accessed with the `\c` command as shown in Listing 2.15. Here, the NFS door is accessed, which is reflected by the new *(nfs4-dcache-dot@dcache-dot_ nfs4wnDomain)* location prompt.

```
1  [dcache-head-dot] (local)
2  sandro > \c nfs4-dcache-dot
3  [dcache-head-dot] (nfs4-dcache-dot@dcache-dot_nfs4wnDomain)
4  sandro >
```

Listing 2.15: Accessing the dCache NFS door

The NFS door can be prompted to list all associated dCache pool nodes including their role (here: DS), their device ID and IPv4 and IPv6 socket addresses with the *show pools* command (Listing 2.16).

```
1  [dcache-head-dot] (nfs4-dcache-dot@dcache-dot_nfs4wnDomain)
2  sandro > show pools
3
4      dcache-pool01:
5      DS: 00000003000000000000000000000000,
6      InetAddress:
7      [/2001:001:200:3004:0:0:5:60:24001, /123.45.67.44:24001]
```

Listing 2.16: Listing available information on pools associated to the dCache NFS door

After having switched from the NFS door to the pool node mentioned above, a query is sent

to the location manager service (`\sl`) in order to list all the pools which the file with the given chimera ID is stored on. Listing 2.17 shows that the file resides only on a pool named *dcache-pool01*. The chimera ID is reprinted on the console together with metadata such as the information that the file is only cached on disk and might be persisted on tape storage (`<C` flag) [60] and its filesize.

```
1  [dcache-head-dot] (dcache-pool01@dcache-pool01Domain)
2  sandro > \sl 0000377415A929BF4C66B816684E12D10F7E rep ls $1
3
4      dcache-pool01:
5      0000377415A929BF4C66B816684E12D10F7E
6      <C----------L(0)[0]> 135
```

Listing 2.17: Querying the location of a file given the chimera ID

The following two Listings 2.18 and 2.19 simply show the interface's capability to transform a given path into a chimera ID and vice versa by querying the namespace provider service utilizing the `\sn` command and the `pf` (pathfinder) utility, respectively.

```
1  [dcache-head-dot] (dcache-pool01@dcache-pool01PoolDomain)
2  sandro > \sn pnfsidof /pnfs/desy.de/dot/data_volatile/
3  test_nfs42_10
4
5      0000117A270054204F1E9F068AF74039CC46
```

Listing 2.18: Asking the namespace provider sevice to transform a path into a chimera ID

```
1  [dcache-head-dot02] (dcache-pool01@dcache-pool01PoolDomain)
2  sandro > pf 0000377415A929BF4C66B816684E12D10F7E
3
4      /pnfs/desy.de/dot/data_volatile/test_nfs42_3
```

Listing 2.19: Asking the namespace provider sevice to transform a chimera ID into a path

For dCache system administrators, the possibility to list all active movers with the `mover ls` command is a precious source of information about potential transfer related shortcomings. It is even enhanced by adding the `-u` option allowing for a listing of the user and group ID associated with the user, whose application running on a computing worker node on the other end of the network originally issued the file I/O request (line 8 of Listing 2.20). Line 4 of the same listing starts with the mover ID, followed by the mover's status and the chimera ID of the file the mover is transferring data to or from. Interestingly, even the NFSv4+ state ID and the sequence ID are listed in the output. Both are metrics that can potentially be matched with their corresponding counterparts on the other end of the network for debugging

and failure tracing once gathered (with the help of eBPF programs) from inside the Linux kernel that embeds the NFSv4+ client. Apart from the client's IPv4 address, line 7 of Listing 2.20 shows a second metric of high importance to the dCache administrators. The parameter `LM` indicates the idle time since the mover has last delivered or received any transferred bytes to or from the NFS client. The assigned number should ideally be zero. The higher the `LM` value, the higher the probability of a stuck transfer for any given reason.

```
1 [dcache-head-dot] (dcache-pool01@dcache-pool01PoolDomain)
2 sandro > mover ls -u
3
4     67108996 : RUNNING : 0000A3FE7233DC7C40B481171F8C0154DD4C
5     IoMode=[WRITE, READ, CREATE]
6     h={NFSv4.1/pNFS,OS=[66a664830001008000000005, seq: 1],
7     cl=[123.45.223.116]} bytes=20480 time/sec=136 LM=63
8     <GidPrincipal[51108],..,UidPrincipal[1000],
9     Origin[123.45.223.116]>
```

Listing 2.20: Retrieving information on active mover instances

Fortunately, also the NFSv4.1 session IDs of the associated clients to any dCache storage node can be queried as shown in the next two Listings (2.21 and 2.22). From Listing 2.21 it is possible to match the port number of the client and the NFSv4.1 session ID to be the session maintained with the NFS door (alias MDS) itself, while Listing 2.22 shows the session associated with a dCache pool node (alias DS) from the same client but on a different port and, of course, with a different session ID.

```
1 [dcache-head-dot] (dcache-pool01@dcache-pool01PoolDomain)
2 sandro > nfs sessions
3
4   /123.45.223.116:860 : 66a5646d00000021000000000000001
5   slots (max/used): 15/0
```

Listing 2.21: Listing NFSv4.1 sessions associated to dCache storage nodes

```
1  [dcache-head-dot] (dcache-pool01@dcache-pool01PoolDomain)
2  sandro > \s nfs4-* show clients
3
4     nfs4-dcache-dot:
5     grid-dev-sandro01.desy.de/123.45.223.116:673:Linux NFSv4.1
6     66a55da40001007e0000000000000001 max slot: 15/0
```

Listing 2.22: Listing NFSv4.1 clients associated via a NFSv4.1 session

While the last Listing 2.23 of this sequence contains information about started transfers already known from previous listings, the new information on line 7 is a very helpful metric

for dCache administrators. First, it indicates whether the transfer is a write or a read transfer and secondly, it lists a metric in the form of *moverID@poolname* or simply *mover@pool*. In those cases, in which the poolname evaluates to null (as in 67108980@null), it becomes evident that the mover is not associated with a pool anymore and thus has no chance to complete the transfer it was originally started for. Consequently, the information obtained on line 7 of Listing 2.23 can be considered an important indicator of abnormal pool behavior or pool node downtimes for the alert administrative staff.

```
1 [dcache-head-dot] (nfs4-dcache-dot@dcache-dot_nfs4wnDomain)
2 sandro > \s nfs4-* show transfers
3
4     nfs4-dcache-dot:
5     2024-07-28T00:31:30.74+02:00 :
6     000012BC097370F64353B30BCFBE85D6E891 :
7     WriteTransfer 67108980@dcache-pool01,
8     OS=[66a55da40001007e00000016, seq: 1],
9     cl=[123.45.223.116], status=[idle], redirected=true
```

Listing 2.23: Retrieving information on NFSv4 transfers

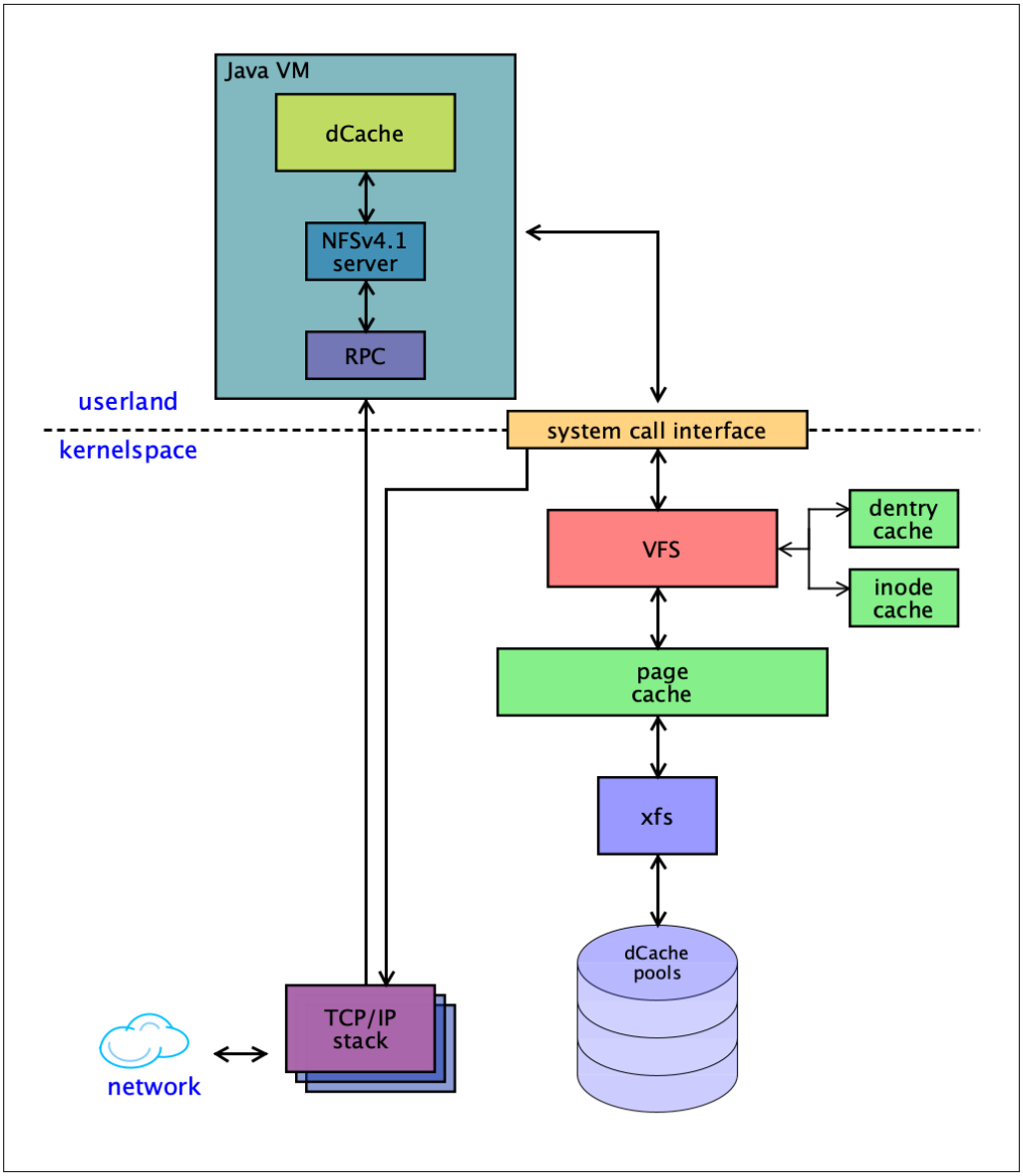

Figure 2.10: Main components involved in servicing a NFS file I/O operation on the dCache storage end

3 Linux Kernel Tracing and Probing

As the complexity of the Linux operating system has increased over the past decades, it has become more and more difficult to understand its inner workings [68]. Especially the observation of the system at runtime with mere command line utilities makes relevant system aspects appear to system administrators as if they were hidden in a black box. The development and integration of tracing and probing mechanisms into the Linux operating system and the kernel specifically, started at the beginning of the century [69].
To be absolutely clear about the semantics of the terms tracing and probing and what distinguishes them from one another, an apt definition with respect to the given context is provided in the following.

The Cambridge Essential American English Dictionary provides two suitable definitions for the term *tracing*. It describes the act of tracing as a method «to follow the movements, progress, or development of something». The purpose of which is to find «proof that someone or something was in a place» or not. The term *probing* is described as«to examine something with a tool, especially in order to find something that is hidden». The mentioned «tool» could also be referred to as a *probe* which in turn is defined as «a device that is put inside something to test or record information» (all definitions from [70]). With these definitions at hand, the transition into the domain of the Linux kernel can be made. That is, to describe why and what to trace and probe inside the kernel at all.

One reason is to simply learn about the proceedings of the kernel. Reading the kernel source code can already be considered a kind of tracing that promotes comprehension, yet neither a very quick nor a dynamical one. Undoubtedly, it is essential to understand how the different subsystems and layers of the kernel interact and which mechanisms allow for that interaction during runtime. Tracing can provide knowledge about the temporal dependencies of events and the flow of control on kernel code execution paths. Analogously to the above quote from the Cambridge dictionary, tracing provides certainty about the fact whether the control flow has come across a probepoint during execution or not.

Secondly, tracing enables the gathering of performance indicators or metrics. The aggregation of these metrics allows for insights into system parameters, such as network throughput, I/O latency, memory usage, CPU load, cache hits and misses and so forth. Thus, tracing

facilitates the evaluation of system parameters with respect to potential bottlenecks or other performance degrading issues. To the first two reasons for applying tracing methods to the running kernel, the third one adds the ability to perform debugging of potential performance bottlenecks or system failures due to code flaws or incompatibilities, among many other issues. Tracing-assisted debugging can be performed on a indication-based and less disruptive level than interactive debugging with, e.g. GDB (GNU Debugger). [71]

3.1 Tracing and Metrics Collection Utilities

The Linux operating system offers a broad spectrum of built-in tracing mechanisms and utilities. One of them was already introduced in Section 2.4. It is the userland syscall tracer called *strace*. Other tracing utilities like *ftrace* [72, 73], *tcpdump* [25] and *perf* [69, 25] have been an integral part of the Linux kernel's capabilities for spotting system specific issues since the noughties. Even if each of them aims at tracing different system aspects, they all can assist in collecting relevant metrics and identifying performance deficiencies in the operating system during runtime [69].
In addition, Linux system administrators have a wide palette of commands at hand that can be utilized to obtain system-specific metrics and statistics. These commands are issued from within a Linux command shell like the `bash`, while the output is either written to the terminal or can be redirected, e.g. into a file for later analysis.

3.1.1 The `lsof` Command

For an administrator, the inspection of the file description table of a specific process identified by its PID can be achieved by leveraging the powerful `lsof`[1] command. Listing 3.1 shows its invocation from the `bash` together with the option `-p` that allows to list the open files of a process specified by its PID.

[1] The command's shorthand name expands to: ***l**ist **o**pen **f**iles* (see `man lsof`)

```
# lsof -p 1088623
   COMMAND     PID USER  FD  TYPE DEVICE  SIZE/OFF  NODE
   vim     1088623 root  4u  REG   0,45   12288     21829222

   NAME
   /pnfs/desy.de/dot/volatile/.test_nfs42.swp
   (dcache-dot-door01.desy.de:/)
```

Listing 3.1: Excerpt output of the invocation of the `lsof` command showing the remote file *.test_nfs42.swp* opened by the local program *vim*

The `lsof` command output excerpt can be read as follows:
A command (or program) called *vim* that has the process ID 1088623, was issued by a user called *root* and has a file of type ***regular*** opened. The index (or file descriptor, FD) in the file description table of process 1088623 is number *4u*, where the appended *u* denotes the read and write access mode[2]. The device numbers, the current size or offset of the file and the inode number precede the path listing to the actual file named *.test_nfs42.swp*. Since it is a file opened within an NFS share, `lsof` even displays the hostname and mountpoint information of the NFS server from which the file originates.

3.1.2 The `/proc` Filesystem

System administrators can also utilize the `/proc` pseudo filesystem, which is mounted at system boot time, to retrieve process-related information otherwise hidden in the kernel. Listing 3.2 shows a similar output content excerpt as above, namely a glance into the file description table of the process with PID 1088623.

```
# ls /proc/1088623/fd
   lrwx------. 1 root root 64 Oct 21 16:58
   4 -> /pnfs/desy.de/dot/volatile/.test_nfs42_12.swp
```

Listing 3.2: Excerpt listing output of the `/proc/1088623/fd` directory revealing the remote file *.test_nfs42.swp* opened by the user process with PID 1088623

The invocation of the `ls` command with the `/proc/<PID>/fd` directory as argument shows a symbolic link (or symlink) called *4* that points (->) to the associated path of the file opened by a process with the given <PID>. The symlink *4* matches the value of the actual file descriptor number 4 used by this process. But there is more information about

[2] See Linux manual entry: `man lsof`

the file descriptor number *4* in the `/proc/1088623/fdinfo` directory. Running the `cat` command on the `/proc/1088623/fdinfo/4` file reveals the following output:

```
# cat /proc/1088623/fdinfo/4
   pos:  12288
   flags:  02500002
   mnt_id: 423
   ino:  21829222
```

Listing 3.3: Output of the content of the `/proc/1088623/fdinfo/4` file revealing the offset (pos) plus file flags and the mount ID/inode number combination which renders the inode number unique even across multiple mounted filesystems [3]

In Listing 3.3 even more metadata for the file referenced by file descriptor number *4* are displayed. The current file offset shown here as pos (for *position*), the file flags and both, the mount ID and the inode number of the associated inode, which in combination render the inode number unique across multiple mounted filesystems [3] are the content of the file.

These small examples disclose only a tiny fraction of the possibilities that exist to obtain relevant and otherwise hidden information from the kernel by querying the directories of the `/proc` filesystem. Unfortunately, in certain cases a direct query can be rather uninformative due to the fact that the `/proc` filesystem stores raw numeric values as kernel metrics, often without any valuable formatting or description applied to it. Listing 3.4 gives an example of the content of the file `nfs` that is meant to expose RPC and NFS related server and client statistics.

```
# cat /proc/net/rpc/nfs
   net 0 0 0 0
   rpc 1764145 6 1764167
   proc4 69 8 8 1482113 2 37...(only 6 values out of 65+ are shown)
```

Listing 3.4: Excerpt output of the content of the `/proc/net/rpc/nfs` file revealing only numerical values without informative descriptions

In order to translate the above output into a descriptive and humanly readable form, commands such as `nfsstat` and `nfsiostat` have been developed. They apply descriptive labels and a decent formatting to the output values as shown in parts in Listing 3.5.

```
# nfsstat
   Client rpc stats:
   calls      retrans    authrefrsh
   1763984    6          1764006

   Client nfs v4:
   null         read         write           commit       open
   8    0%      8     0%     1482113  84%    2    0%      37    0%
   ...
```

Listing 3.5: Excerpt output of the invocation of the `nfsstat` command which applies informative labels to the numerical values from Listing 3.4

In the above Listing (3.5) RPC client statistics such as the call count, the number of retransmissions, and the refresh of the cached credential information are listed. Additionally, NFSv4 client statistics about NFSv4 operation counts, such as the number of times a NULL call was transmitted, and the same for READ, WRITE, COMMIT and OPEN operations are shown exemplarily. The command called `nfsiostat` which translates values retrieved from the `/proc/self/mountstats` file into a form that can be evaluated, exposes more I/O related statistics. It comprises values about the throughput of the READ and WRITE operation, that is, the number of kilobytes read or written per second, as well as general NFS related statistics such as the number of operations per second, the number of kilobytes read or written per each operation, the cumulative average RPC queuing and round-trip times, and the number of operations (READ or WRITE) that completed with an error.
It is most important to understand that most utilities including the two described above aggregate I/O metrics on a per NFS client or per NFS server only. They do not allow the association of transfer metrics with individual users or user processes. System administrators are therefore limited in their scope when trying to interpret the obtained metrics with regard to issue handling.

3.1.3 The `rpcinfo`, `rpcctl` and `rpcdebug` Utilities

More RPC-related commands comprise the `rpcinfo`[3] command which allows to probe a given hostname and display all available rpc programs running on the remote machine. An invocation of `rpcinfo` as shown in Listing 3.6 with the `-a` option set followed by the server IP address and port number (for uadddr notation see Section 2.8.1.2) returns a sudden response if the server's RPC service for the specified program number and version is available and listening.

[3] See Linux manual entry: `man 8 rpcinfo`

```
# rpcinfo -a 131.169.223.60.8.1 -T tcp 100003 4
     program 100003 version 4 ready and waiting
```

Listing 3.6: Output of the invocation of the `rpcinfo` command with the -a option followed by the server IP address and port number plus the transport to be used and the specification of the RPC program number and version

The RPC program number (100003)[4] in the example of Listing 3.6 corresponds to NFS and the version is set to 4.

Another source of RPC connection metrics is the `rpcctl`[5] utility. It can help to identify connection flaws and gives the system administrators the possibility to actively manipulate connection parameters on the RPC and xprt[6] layer deep inside the kernel. Hence the name *rpcctl == rpc control* of the program. The command can show RPC client connection metrics as well as transport-related ones. Excerpts of the output of a `rpcctl client show` and `rpcctl xprt show` invocation are shown in Listing 3.7.

```
# rpcctl client show
   clnt-1: switch-1, xprts 1, active 1, queue 0
         xprt-1: tcp, 131.169.223.60 [main]
# rpcctl xprt show
   xprt-1: tcp, 131.169.223.60, port 2049, state <CONNECTED,BOUND>,
  main
         Source: 131.169.223.116, port 771, Requests: 2
         Congestion: cur 0, win 256, Slots: min 2, max 65536
         Queues: binding 0, sending 0, pending 0, backlog 0, tasks 0
```

Listing 3.7: Output of two invocations of the `rpcctl` command showing RPC client and transport (xprt) related metrics

The above output of the `rpcctl client show` invocation displays valuable metrics about the connection state of the RPC client, such as which xprt number it uses to which host, the transport protocol, and how many connections are currently active on the very RPC client. The metrics exposed by the `rpcctl xprt show` invocation of the xprt used by client 1 (clnt-1) are even more complete. The socket address, transport protocol and state of the connection are revealed as well as congestion information such as window sizes and metrics about the kernel's RPC workqueue which can carry information about potential issues related to the transmission of the RPC calls.

[4] Defined as macro `NFS_PROGRAM` in `/include/uapi/linux/nfs.h`
[5] See Linux manual entry: `man 8 rpcctl`
[6] Kernel developer's shorthand for the term *transport*

The last of the three utilities mentioned in the subsection's title, named `rpcdebug`[7], utilizes `dprintk` statements distributed across the source code of RPC- and NFS-related kernel modules. Activating the functionality by setting a debug flag with the `rpcdebug` utility «causes the kernel to emit messages to the system log in response to NFS activity» as the manual entry states. According to the same `rpcdebug` manual pages, the file that is actually read by the command is either one of the `/proc/sys/sunrpc/{rpc,nfs,nfsd,nlm}_debug` files, depending on which module is chosen. Apart from NFS client and RPC modules it evidently also allows to print debug messages from the Linux built-in NFS server, called `nfsd` in Linux kernel parlance as well as the Network Locking Manager facility, `nlm`, which is not used in the NFS protocol version 4+ anymore (see also Section 2.8.1.1). Listing 3.8 shows the setting of all available debug flags (with the `-s` option) for the RPC module (`-m rpc`) and lists them afterward. In addition, the system log output of a specific debug message is listed subsequently, invoked by the `dmesg -W` command. The `-W` option causes the system log output to wait for new messages and display them once they arrive.

```
# rpcdebug -m rpc -s all
   rpc     xprt call debug nfs auth bind sched trans svcsock svcdsp
  misc cache
# dmesg -W
   ...
   [4895451.213299] RPC:       xs_tcp_send_request(372) = 0
   [4895451.306931] RPC:       xs_tcp_send_request(452) = 0
   ...
```

Listing 3.8: Setting all available debug flags (with the `-s` option) for the RPC module (`-m rpc`) using the `rpcdebug` utility followed by an invocation of the `dmesg` command showing excerpts of the corresponding debug messages

The `dmesg` output shown in Listing 3.8 derives from a kernel function called `xprt_sock_sendmsg()` that is invoked within the body of the `xs_tcp_send_request()` routine. They are defined in `/net/sunrpc/socklib.c` and `/net/sunrpc/xprtsock.c`, respectively and are part of the kernel's TCP network stack. The actual `dprintk` statement in the sources is shown in Listing 3.9.

[7] See Linux manual entry: `man 8 rpcdebug`

```
static int xs_tcp_send_request(struct rpc_rqst *req)
{ ...
    status = xprt_sock_sendmsg(...);
    dprintk("RPC:       xs_tcp_send_request(%u) = %d\n",
    xdr->len - transport->xmit.offset, status);
... }
```

Listing 3.9: Excerpt from the kernel sources in the `/net/sunrpc/xprtsock.c` file revealing the responsilble `dprintk` statement and its parameters.

It reveals information on the displayed metrics. Obviously, the numerical value enclosed in parentheses is the size of the XDR encoded RPC and NFS payload in bytes. The returned zero value indicates the successful execution status of the transmission.

3.1.4 The `tshark`/`wireshark` Network Packet Tracer Utility

An absolutely powerful and indispensable tool, extensively utilized in every stage of this work's research and experimentation, is the `tshark`[8]/`wireshark` [1] utility. Since `tshark` is the command-line-based version of the packet tracer that supports the same options as `wireshark`, it can be utilized whenever an interactive user interface is not available. In turn, `wireshark` offers user-friendly graphical user interface, making the capturing, displaying of dissection results, filtering, searching, importing and exporting of capture files and colorization of packets an easily manageable task [1]. It allows to capture network packets and display their dissection results, revealing all header and data information contained therein. The utility can assist in studying the inner workings of a network protocol (like NFSv4+) by tracing the communication between participating network components that exchange protocol-specific messages. Once having acquired the ability to interpret the protocol-specific packet content, the metrics provided by the `wireshark` utility help administrators to detect and ultimately solve potential communication flaws between network components. Fortunately, the capture file format produced by the Linux built-in `tcpdump` utility as well as the files produced by `tshark` can be opened and displayed in the `wireshark` utility for subsequent in-depth analysis conveniently. Unlike the above-mentioned tracing utilities, `wireshark` is an externally developed community effort and has to be downloaded and installed prior to its utilization. Appendix A shows a complete NFSv4.1 payload dissection of a Linux NFSv4.1 client request from a NAF's worker node and the reply of the NFSv4.1 server from the dCache storage end in Listings A.11 and A.12.

[8] See Linux manual page entry: `man tshark`

3.1.5 `ftrace` - Linux Kernel Function Tracer

The `ftrace` facility was developed and is still maintained by kernel developer Steven Rostedt [72]. It became part of the Linux mainline kernel in the year 2008 [73]. It has evolved in functionality from a mere function-tracing utility toward enabling latency observations in scheduling flows as well as event tracing [72]. In the context of this present thesis the function tracer functionality and specifically the ability to record and display almost complete function call graphs have been of major interest. The results that can be obtained by its application are used to display the sequence of kernel function calls during the mount process of an NFS share in Appendix A starting from Section A.3.1.

Since the `ftrace` infrastructure is embedded within the kernel, it is immediately available and operational after system boot. All relevant files and directories of the function tracer can be found under the `/sys/kernel/tracing/` pseudo filesystem tree. The directory holds all configuration files as well as those used for direct interaction and control of the tracer itself. Although a front-end for `ftrace` called `trace-cmd` is provided to facilitate the handling of the control files by creating one-liners that include all necessary instructions, it has not been utilized during experimentation and verification phases of this thesis. Instead, the tracer was configured and controlled by *echoing* values into the configuration and control files (see Section 4.3).

A list of kernel functions that can be traced is found in the `available_filter_functions` file. Kernel modules of services like the NFS client which are not loaded at boot time have to be loaded beforehand (or started as a *systemd* service) in order to appear in the `available_filter_functions` file.
The `ftrace` utility offers numerous options for configuration. An interesting one to mention in the function graph tracer configuration is the parameter `max_graph_depth` which allows to choose the number of functions the tracer will descend into and include in the trace output. A second option enhances the focus on relevant function calls by preventing e.g. interrupts to be traced. This is achieved by writing function names to be excluded from the trace output into the `set_graph_notrace` file.

3.1.6 Event Tracing

It was mentioned before that `ftrace` possesses event tracing capabilities. What defines an event in the context of kernel tracing? According to [74] an event is the encounter of the execution control flow at runtime and a probepoint. The latter denotes a single location within the sequence of instructions that belong and correspond to a function's code block. The probepoint can be located anywhere within the instruction sequence. Each time the probepoint is *hit* by execution control flow an event is triggered. The consequence of the triggered event has to be provided by the developer interested in observing the function's execution context. By attaching or *hooking* a probe to a probepoint, the developer can inject custom code into the probed function's instruction sequence. Once an event is triggered at the probepoint, the injected instructions are executed. After completion of the probe's instruction sequence, the execution of the next regular instruction inside the probed kernel function is resumed. This mechanism allows for the collection of relevant runtime information that can be analyzed and used for performance measurements, debugging or simply to gain insights into the inner workings of the probed function as described at the beginning of this section. Since the underlying mechanisms of event tracing are shared among tracing utilities, such as `perf`, `ftrace`, `eBPF` and many others [74], the concepts of static as well as dynamic event tracing will be briefly introduced in the following two sections.

3.1.6.1 Tracepoint-Based Event Tracing

Tracepoints in Linux are located at determined places in the kernel source code. They can be recognized by the `trace_`-prefix preceding their function name. Listing 3.10 gives an example of multiple tracepoint locations within the body of the `xprt_request_transmit()` kernel function defined in `/net/sunrpc/xprt.c`. Tracepoints are inserted into the source code by kernel developers following a determined coding procedure involving a tracepoint definition in a header file. Additionally, they need to define a tracepoint statement providing a unique identifier for the tracepoint as described in the kernel documentation [75]. The documentation also states that each dormant tracepoint (that does not have a probe attached to it) does not cause any performance overhead except for a negligible time penalty for checking a condition and a similarly small space penalty. Since the inserted tracepoints are placed at fixed locations in the source code determined at the kernel code development stage, they are commonly referred to as *static* probepoints.

```
1  static int
2  xprt_request_transmit(struct rpc_rqst *req, struct rpc_task
                                                    *snd_task)
3  { ...
4    / * Tracepoint 1 */
5    trace_rpc_xdr_sendto(task, &req->rq_snd_buf);
6    ...
7    status = xprt->ops->send_request(req);
8    if (status != 0) {
9      req->rq_ntrans--;
10     / * Tracepoint 2 */
11     trace_xprt_transmit(req, status);
12     return status;
13   }
14   if (is_retrans) {
15     task->tk_client->cl_stats->rpcretrans++;
16     / * Tracepoint 3 */
17     trace_xprt_retransmit(req);
18   }
19     ...
20 }
```

Listing 3.10: Excerpt from the `xprt_request_transmit()` kernel function reveals three embedded tracepoints (comments added for clarity)

3.1.6.2 Kprobe-Based Event Tracing

Kernel Probes or kprobes, as they are commonly abbreviated to, are the dynamic counterpart to tracepoints. As they are not predefined, the probepoints to which a probe can be hooked can be virtually any instruction in the kernel code [76]. The kprobe mechanism works by registering a kprobe for a probed instruction. As opposed to static tracepoints, the user of the kprobe determines which instruction is to be probed. By registering a kprobe, which includes hashing it into a list of registered probes, the original instruction is copied into a memory space where it can be single-stepped *out-of-line*. It is then replaced by a breakpoint instruction. Once the execution control flow hits the breakpoint, the kprobe infrastructure code is notified and the hashed list of kprobes is searched for a registered kprobe corresponding to the breakpoint. If a match is found, control is passed to a user-defined kprobe *pre-handler* that allows the collection of relevant metrics *before* the original instruction is single-stepped. After the completion of the pre-handler the copied instruction is single-stepped and a likewise user-defined kprobe *post-handler* is executed that allows to record information *after* the probed instruction. Once the post-handler completes, execution resumes normally at the

instruction after the probed one. The kprobe pre- and post-handler code is provided by the user. Due to the general lack of code verification, erroneously written handlers can cause kernel crashes or other less disruptive issues. [77]

3.2 eBPF (Extended Berkeley Packet Filter)

At the present time, the eBPF acronym has merely become the label for a technology, whose abilities have gone far beyond the filtering of network packets. And since the acronym is solely used when having to differentiate it from *classic* BPF (cBPF) [78], it has become a custom to simply call it BPF (as will be done in the following). But what is BPF and how does it differ from the tracing utilities mentioned so far?
As Brendan Gregg describes in his book about *BPF Performance Tools*, BPF is an in-kernel «general-purpose execution engine» that can be utilized to «create advanced performance analysis tools» [2]. Since this definition includes tracing, BPF can be also considered a tracing technology. In fact, it is used to trace and probe the Linux operating system kernel. It allows developers and system administrators to run custom programs safely within kernelspace without having to change the kernel source code, load kernel modules or recompile the kernel. Unlike the other tracing utilities so far mentioned, BPF programs not only allow administrators to query the kernel for relevant system metrics at runtime, but also enable the application of security [79] and networking policies [80] e.g. based on packet metrics. Similar to other tracing utilities, BPF programs help with debugging runtime issues of kernel subsystems and modules as well as analyzing system behavior. However, due to the way BPF programs are written, the possibilities of tracing have become more customizable. BPF programs can be manufactured and shaped according to the specific needs of system administrators. For example, the state of virtually every instruction executed in the kernel can be monitored by leveraging the kprobe infrastructure of the kernel in a *safe* way with only minimal performance overhead. While utilizing the same underlying kernel tracing mechanisms, custom BPF program code has to undergo a rigorous verification process prior to its inclusion into the kernel. This ensures the preclusion of harmful program behavior due to programming flaws or malicious code and consequently prevents crashes or corruption of the running kernel [2]. This opens up a fairly larger and safer spectrum of feasible options for monitoring what is really happening deep inside the Linux kernel at runtime.
The *e*BPF infrastructure including a `bpf()` system call was merged into the Linux kernel version 3.18 in 2014 [81, 82]. It surely has evolved ever since and is currently still undergoing vivid development. Expansions, improvements and new functionalities are added regularly [83].

Actually, the BPF infrastructure has become a kernel subsystem on its own [84]. Figure 3.1 depicts the control flow across all components involved in the BPF infrastructure including userland elements. Those will be described briefly in the following sections.

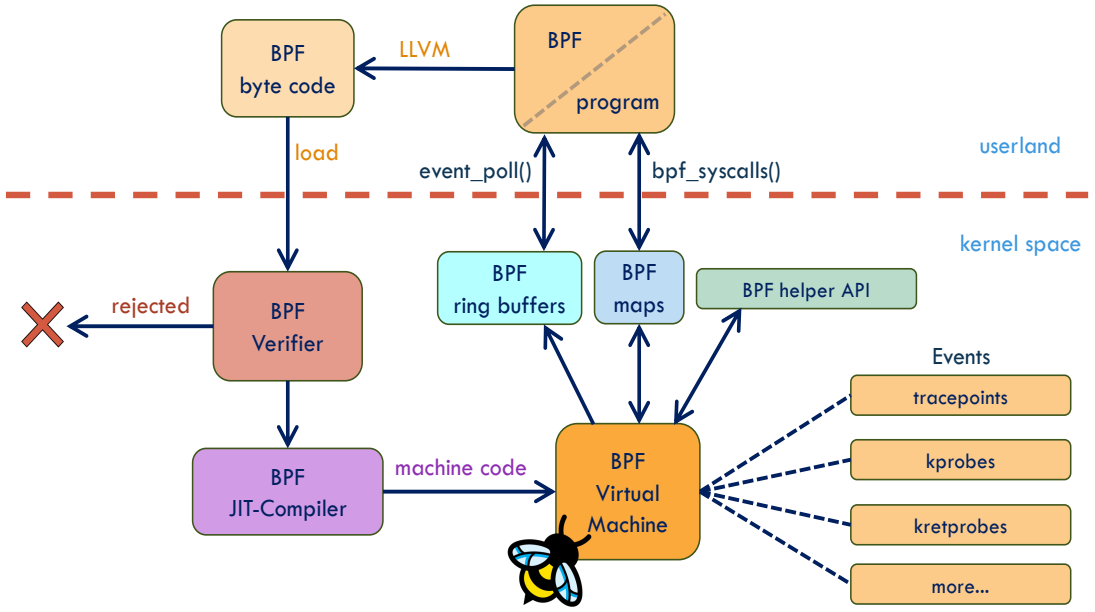

Figure 3.1: The eBPF infrastructure (drawn according to [2])

3.2.1 BPF Development Frameworks

In order to facilitate the development of custom BPF programs (according to the needs of system administrators) a userland library called *libbpf* is provided by the Kernel development community. It is a C language-based library that supports the preparation of compiled BPF object files for verification, loading and attachment to hooks inside the kernel as well as their removal[9]. Developers of BPF programs are able to utilize libbpf-provided APIs that allow for interaction with the running in-kernel BPF program part across storage data structures called *maps*. Additionally, libbpf supports the use of the BPF CO-RE (compile once - run everywhere) concept for improved portability of BPF programs across kernel versions, which will be touched upon in Section 3.2.5. In short, libbpf supports and helps with the effectuation of necessary tasks throughout the four phases of a BPF program's life-cycle, enabling the

[9]The opening, loading (into the kernel), attachment (to a hook) and destruction (or tear-down from the kernel) of an BPF object are commonly referred to as the four phases of the BPF object's *life-cycle* [85]

developer to focus on the BPF program's content. [85]
Other libraries and frameworks exist, e.g. the *ebpf-go* project that provides an environment for BPF program development in the GO programming language [86] or the *BCC*[10] project which offers a development toolchain for the creation of BPF programs with frontends in Python and the Lua programming language [87]. Since libbpf is a fully self-contained Linux library without external dependencies, and because it allows the whole development-cycle of BPF programs to be carried out using the C programming language, it has been chosen for the development of BPF programs in this present work. Thus, only the methods applying to the BPF program development with libbpf will be described.

3.2.2 BPF Program Structure

As the dotted line that cuts through the *BPF program*-box of Figure 3.1 implies, BPF programs consist of two distinct parts. A userland part and a kernel part. The name of the source code file containing the userland part conventionally ends with a *.c* file extension, while the kernel part source code file ends with *.bpf.c*. Optionally, a header file (with *.h* extension) containing `struct` declarations and `#define`s used in both parts can be provided as well. The userland code of a BPF program typically contains a `main()` function that comprises code which handles the four phases of the BPF object's life-cycle. Additionally, a *while*-loop is included in the `main()` function to perform event polling (using the `event_poll()` routine) on a ring buffer data structure serving as data vehicle from kernelspace BPF program part to the userland part (see Figure 3.1). Finally, it contains the definition of a *handle_event()*[11] routine which, as the name suggests, processes the event data collected by the in-kernel part of the BPF program, e.g. by simply printing it to the console or performing further computation before forwarding the results (e.g. in JSON format) to a visualization and analysis platform like e.g. *OpenSearch-Dashboards* [88].

The kernel part of a BPF program is responsible for gathering, temporarily storing and transferring the desired metrics from the probepoints to which the BPF program is hooked to the userland BPF program code. The gathering of metrics is achieved by utilizing the BPF helper API. The function calls this API offers, permit a safe retrieval of valuable information from the kernel hook's execution context. Examples of BPF helper functions will be listed in Section 3.2.4. In-kernel storage needed by the kernel-side BPF program part is provided by the above

[10] BPF Compiler Collection
[11] The given function identifier is not mandatory, but semantically it is one of the few correct ways of naming it.

mentioned *map* data structures and the transfer happens through a map-like data structure, also mentioned above, called *ring buffer* [89]. Both parts of the BPF program can be compiled with the Clang compiler and its LLVM[12] backend, which supports BPF as a compilation target directly. The kernel BPF program part is first compiled into *BPF-bytecode*. LLVM includes an optimizer that helps reduce the number of BPF-bytecode instructions emitted which is an important aspect with respect to the next stage to which the kernel BPF program bytecode has to proceed to: the BPF verifier. [2]

3.2.3 The BPF Verifier

The BPF bytecode verifier validates the integrity of the first stage compilation result of the user provided kernel-side BPF program code. The BPF bytecode thereby undergoes a rigorous verification process with respect to a set of rules enforcing compliance with predefined limits. To prevent the leakage of sensitive kernel data to userland the bounds of valid memory access are checked for every instruction. The BPF verifier also enforces a usable per BPF program stack size limit of only 512 bytes. As already mentioned in Section 2.1 the kernel mode stack is small and static in size and K. N. Billimoria, author of the book named *Linux Kernel Programming: A Comprehensive Guide to Kernel Internals [...]* warns the aspiring kernel developer to

> [...] be very careful to not overflow your kernel stack by performing stack-intensive work (such as using large local variables or recursion). [29]

This is prevented by the BPF verifiers strict stack limit verification.
Another aspect that is ensured by the verification process is the termination of the BPF program's control flow. Thus, a complete path verification is performed. Every single decision branch is tracked until an exit statement is met. This is especially important with respect to the number of possible iterations that can be performed in a loop which currently is limited to thirty-two. The BPF verifier imposes a stringent one million bytecode instruction limit for its verification process. Taking into consideration that this instruction count limit includes every single execution path the verifier has to track for every single iteration step in a loop, it becomes evident that the overall complexity of the kernel-side BPF program is rather limited. Eventually, should any of the above mentioned rules be violated or limits exceeded by the BPF program code, the whole program is rejected and the execution of the BPF program fails. [89]

[12] LLVM is not an acronym but the name of the project itself (see https://llvm.org/).

3.2.4 The In-Kernel BPF Virtual Machine

The core of the in-kernel BPF infrastructure is the BPF Virtual Machine (VM). It consists of a software implementation of ten general-purpose 64-bit (virtual) registers. The BPF bytecode instructions operate on these ten registers utilizing them for the maintenance of state at program runtime. The probed function's return value is stored in register *0*, while registers *1* through *5* hold the probed function's arguments, if there are any. Registers *6* through *9* hold callee-saved values that are preserved across function calls. The read-only register number *10* holds the current stack-frame pointer. [2, 89]
The verified BPF bytecode is subsequently translated to native machine code that can be executed on the corresponding processor architecture. This task is performed by a *Just-In-Time* (JIT) compiler component at runtime. [2]
As mentioned in Section 3.2.2 already, the kernel-side BPF program code calls into a BPF-helper API for the safer and easier retrieval of predefined metrics. These metrics include a pointer of type `struct task_struct` to the currently executed task which can be obtained by means of the `bpf_get_current_task()` helper function. Studying the kernel source code[13] reveals that the latter function simply returns a pointer to *current*, the per-CPU variable seen in Section 2.2.1. Other helper functions with rather self-explanatory names are shown in the non-exhaustive list[14] below.

- `bpf_get_current_pid_tgid()`
- `bpf_get_current_uid_gid()`
- `bpf_get_current_comm(void* buf, __u32 bufsize)`
  obtain the name of the executable for the current task and write it into a buffer with size `bufsize` pointed to by `buf`
- `bpf_get_smp_processor_id()`
- `bpf_ktime_get_ns()`
  obtain a kernel time stamp (time elapsed since boot time) in nanoseconds
- `bpf_probe_read_kernel(void *dst, u32 size, const void *unsafe_ptr)`
  access kernel memory in read mode safely within bounds

[13] See `/kernel/trace/bpf_trace.c`
[14] See Linux manual entry: `man 7 bpf-helpers`

It must be added here that not all helper functions are available in all BPF programs depending on the BPF program type that is used. BPF program types must be declared by the BPF program developer. They depend on the subsystem and the kernel function that is to be probed. A command line utility called `bpftool` that is installed by default in the Linux RHEL distribution can be invoked with the `feature` argument to obtain information about which BPF helper functions are currently supported by which BPF program type.[89] In fact, the `bpftool` utility is of great help to the BPF program developer as it allows to inspect the internals of the otherwise hidden BPF virtual machine.

3.2.5 BPF CO-RE and the BPF Type Format

Reading a value of a field from a kernel structure is a frequently performed operation within a BPF program. One of the major drawbacks in BPF programming is the fact that fields can change their offset position within a `struct` from one kernel version or kernel configuration to another. This shift in field position due to alteration of `struct` layouts is referred to as *relocation*. For obvious reasons, the relocation of fields is detrimental to the portability of BPF programs between different kernel versions and configurations. Fortunately, libbpf supports a concept called CO-RE (Compile Once - Run Everywhere) that makes reading a `struct` field relocatable. By supplying so-called *BPF Type Format* (BTF) information about the kernel's data structures and function layouts, libbpf is able to perform a relocation of shifted offset values on the target kernel. Again, the `bpftool` utility mentioned in Section 3.2.4 comes to the rescue with the extraction of BTF information from the source kernel at development time, rendering the provision of the BTF type information an easy task for the BPF program developer. In conclusion, instead of using the above-mentioned `bpf_probe_read_kernel()` helper function to access fields from kernel memory, a macro called `BPF_CORE_READ()` is used to perform the same task, but with correct offsets applied according to the BTF information of the target kernel. [85, 90]

4 Methodology

The following sections contain the description of methods, environments and tooling utilized to develop and verify the three custom BPF programs. The functionality of the BPF programs as well as two use cases for their application in the NAF-dCache environment at DESY will be presented.

4.1 The Test and Experimentation Environment

For the development and verification process of the custom BPF programs a test and experimentation environment is used. It consists of a virtual machine[1] (VM) running the Linux RHEL 9.4 operating system with a 5.14.0-427.22.1.el9_4.x86_64 kernel version. Inside this VM the BPF program code is developed using the command shell built-in *vim* text editor. Requirements and dependencies for the build process of the BPF programs comprise the following libraries and toolchains:

- libbpf (already installed as /usr/lib64/libbpf.so.1.4.0)
- libelf (already installed as `/usr/lib64/libelf-0.191.so`)
- clang-18.1.8-3.el9.src.rpm (package must be installed)

To complete the development setup an NFS share[2] of the dot-dCache instance is mounted onto the VM's local filesystem tree under the `/pnfs` root directory. Only the NFS version 4.1 is specified as mount option. (see Listing A.2 in Appendix A). The dot-dCache instance is a dCache cluster used by the scientific computing group members for testing and debugging and thus is not part of the productive dCache storage clusters. With the provision of an NFS share to the dot-dCache instance, a close-to-reality environment is available for the development and testing of the custom BPF programs, without having to interfere with, and potentially be detrimental to, productive machines and production flows.

[1] Kindly set up by Thomas Hartmann, system administrator in charge of the NAF and Grid computing facilities at DESY.

[2] Kindly provided by Christian Voss, head of the dCache operational team (hence the name *dot*) at DESY

4.2 Outline of the Custom BPF Programs

Three custom BPF programs are the outcome of the software development process. The three programs are briefly presented in the following. The focus lies on their main purpose and the specific nature of the implemented control flow.

4.2.1 BPF program 1: `nfs4_byte_picker`

The `nfs4_byte_picker` BPF program aims at the collection of NFSv4+ and RPC-related metrics. Its purpose is to enable a system administrator to aggregate the amount of bytes sent and the amount of bytes received by a single RPC request-reply round trip across the network. The special feature of the `nfs4_byte_picker` is to allow the association of the transferred bytes with a specific user process via its PID and TGID as well as with a specific user via its UID and GID metric. A more detailed explanation is necessary with respect to the obtained process identifiers. Two distinct points in time, at which the PIDs[3] of the executing threads are obtained, have to be differentiated. The first one belongs to the thread that executes the probed kernel hook and thus is active when the event fires. It is retrieved from the `pid` field of the currently executed `struct task_struct` returned by the `bpf_get_current_task()` helper function inside the BPF probe that is attached to the hook. The second point in time occurs earlier in the sequence of function calls involved here. Namely, during the execution of the `rpc_init_task()` routine defined in `/net/sunrpc/sched.c` which is carried out with every single transmission of an NFS request via the RPC layer. The purpose of the `rpc_init_task()` routine is to initialize the fields of a freshly allocated `struct rpc_task` with values gathered earlier on the NFS layer. One of those fields is the RPC task structure's `tk_owner` field. To the latter the `tgid` of the currently executing thread is assigned as shown in Listing 4.1 .

```
1  task->tk_owner = current->tgid;
```

Listing 4.1: Assignment of the `tk_owner` field of a new RPC task structure with the TGID of the current task inside the `rpc_init_task()` routine

An evaluation of entries in the `nfs4_byte_picker` output shows that the two process identifiers, namely the TGID of the thread executing the kernel hook and the TGID of the thread executing the initialization of a new `struct task_struct`, can differ from each

[3] The TGID is always included with each mention of the PID from this point on, if not declared otherwise.

other. This hints at the fact that the two routines are executed by two distinct threads. Chapter 5 evaluates the implications of this circumstance.
Additionally, the time span from when a request is sent to when the reply is received can be calculated from the obtained time stamps provided by the `nfs4_byte_picker`. More specifically, from when an event is triggered in a probed kernel function hook called `xs_tcp_send_request()` (defined in `/net/sunrpc/xprtsock.c`) that performs the handover of a request from the RPC layer to the TCP layer, to the moment an event is triggered in the probed `xprt_complete_rqst()` routine (defined in `/net/sunrpc/xprt.c`) which handles the incoming RPC reply.

4.2.2 BPF program 2: `nfs4_path_finderV`

The `nfs4_path_finderV` BPF program is designed to associate process and user identifiers with open file descriptors pointing to files stored in a mounted NFS share. More specifically, it first performs a lookup of used file descriptors in the `open_fds` bitmap, as described in Section 2.7 (see Figure 2.2 for orientation). For each file descriptor found in the bitmap an associated file structure pointer is dereferenced until the `i_mode` field of the associated inode becomes accessible and is read. The `i_mode` field contains the type of a file. If the file in question is of type *regular*, the inode's `i_sb` field that points to the associated `struct super_block` is further dereferenced until the `name` field of the associated `struct file_system_type` becomes accessible. The control flow of the BPF probe evaluates whether the obtained name of the filesystem type corresponds to `nfs4`. If the condition evaluates to false, the currently pursued file descriptor is ignored and the next one, if applicable, is dereferenced up to this point in the way described above. Instead, if the name of the filesystem type is equal to `nfs4`, the chain of pointers down to the name of the current path component is followed. The path component's name is stored in a ring buffer whose entries are polled by the `event_poll()` routine in the userland part of the `nfs4_path_finderV` BPF program (see Figure 3.1). When the root dentry ('/') of the path is finally encountered, the reverse-assemblage of the path components (starting from the root toward the final path component) is triggered and output after completion, together with other collected metrics.

4.2.3 BPF program 3: `socket_collector`

The third custom BPF program called `socket_collector` basically has the same control flow design as the `nfs4_path_finderV` program. The former differs from the latter in the evaluation of the filetype, which is accessed in the same way as described above. This time only files of type *socket* are further pursued, and additionally, all filesystem types are considered. The principal question the query performed by the `socket_collector` addresses is:

- Which files of type socket are opened by which process and by which user ?

4.3 Outcome Verification Methods

During the early development phase, a feature provided by the `ftrace` infrastructure called `trace_pipe` is utilized. It is located under the same `/sys/kernel/tracing/` pseudo filesystem path discussed earlier in Section 3.1.5. The `trace_pipe` file can be used to stream live output of the function tracer to standard output. The stream will block until new tracing events become available [72]. By using a BPF helper function called `bpf_printk()` inside BPF programs it is possible to write into the `trace_pipe` stream and view the values of collected metrics displayed directly on the console. This method can be used for the early stage of BPF development, when maps and ring buffers have not been implemented yet and the access to kernel structure fields is tested. Since the `bpf_printk()` helper negatively affects execution speed[4] it should not be used in production code. Nevertheless it is applicable for debugging purposes. In fact, all three custom BPF programs accept an `-d DEBUG` argument that activates the printing to the `trace_pipe` stream for this purpose.

[4] Refer to the Linux manual entry: `man 7 bpf-helpers`

```
kworker/u128:2-3663906 [012] bpf_printk: PID: 3663906
kworker/u128:2-3663906 [012] bpf_printk: UID: 0
kworker/u128:2-3663906 [012] bpf_printk: GID: 0
kworker/u128:2-3663906 [012] bpf_printk: rpc_task_owner_pid:
 3663971
kworker/u128:2-3663906 [012] bpf_printk: uid_cred: 1000
kworker/u128:2-3663906 [012] bpf_printk: gid_cred: 51108
----------------------------------------
dd-3663971 [005] bpf_printk: cl_nodename: grid-dev-sandro01.desy.de
dd-3663971 [005] bpf_printk: PID: 3663971
dd-3663971 [005] bpf_printk: UID: 1000
dd-3663971 [005] bpf_printk: GID: 51108
dd-3663971 [005] bpf_printk: rpc_task_owner_pid: 3663971
dd-3663971 [005] bpf_printk: uid_cred: 1000
dd-3663971 [005] bpf_printk: gid_cred: 51108
```

Listing 4.2: Testing BPF probe functionality at an early development stage using `trace_pipe` live stream (time stamps omitted for brevity)

Listing 4.2 shows metrics delivered from the kernel probe and printed to the `trace_pipe` stream. Only the information printed after each *bpf_printk:* statement originates from the kernel probe's `bpf_printk()` output. The command name, PID, [CPU] and time stamps (omitted here for brevity) at the beginning of each line are provided by the `ftrace` facility. These lines allow a first verification of the obtained metrics. It becomes evident from the lines in Listing 4.2 that a kernel thread called *kworker/u128:2* with PID 3663906 executes the probed kernel function on behalf of a user process called `dd` with PID 3663971 whose TGID is stored in the `tk_owner` field inside the corresponding `struct rpc_task` (refer to Section 4.2.1).

In later development stages, maps and ring buffers are implemented to facilitate the transfer of event records from kernelspace to the userland part of the BPF program. The collected values of each event are mapped to their corresponding metrics in key-value pairs and stored into files as JSON object entries, one for each fired event. An examplary JSON object entry from a result file with self-explanatory metric:value-mappings is shown in Figure 4.1.

```
{
    "host": "grid-dev-sandro01.desy.de",
    "cmd": "sshd",
    "time_stp": 2458737062111,
    "PID": 1381410,
    "TGID": 1381410,
    "UID": 0,
    "GID": 0,
    "fd": 4,
    "sock_type": "SOCK_STREAM",
    "sock_state": "TCP_ESTABLISHED",
    "sock_family": "AF_INET6",
    "sock_src_addr": "2001:638:700:10df:0:0:1:74",
    "src_port": 22,
    "sock_dst_addr": "2001:638:700:1005:0:0:1:74",
    "dst_port": 35962
}
```


Figure 4.1: JSON formatted entry from a `socket_collector` BPF program output

At this stage of development, each experiment performed with one of the custom BPF programs is accompanied by the simultaneous execution of four additional tracing methods. This allows the verification of the obtained metrics from the BPF program output by comparing the available metrics obtained with each executed tracing utility. The five tracing methods applied simultaneously (including the BPF program under test) comprise:

- `strace`
- `ftrace` (see Section 3.1.5)
- `tshark` (see Section 3.1.4)
- `rpcdebug` (see Section 3.1.3)
- BPF program under test

The `strace` utility is mainly used to verify the correct behavior of the userland part of the BPF program, with respect to the BPF system calls it issues.

A typical workflow with the `ftrace` facility aiming at the creation of function graphs as used in the verification process is listed below.

1. Choosing the appropriate tracer for the given purpose (`function_graph` in this case) by writing into the `current_tracer` file:
   `echo function_graph > current_tracer`
2. Selecting the kernel function that is to be traced and set as root node of the function call graph:
   `echo __x64_sys_mount > set_graph_function`
3. Turning on tracing by writing a *1* into the `tracing_on` file:
   `echo 1 > tracing_on`
4. Executing a command or run an application that will trigger the kernel function set in the `set_graph_function`. Here, exemplarily the `mount(8)` command is issued along with appropriate mount options:
   `mount -o vers=4.1 dcache-dot-door01.desy.de:/pnfs /pnfs`
5. Waiting until the mount process is completed and the command-line prompt returns.
6. Redirecting the recorded trace output into a file for later analysis, e.g.:
   `cat trace > /data/test_mount_nfs_241019_217/ftrace_graph_217`
7. Turning off function tracing:
   `echo 0 > tracing_on`
8. Disabling the tracer again by writing `nop` into the `current_tracer` file:
   `echo nop > current_tracer`

Throughout the testing phase, a `max_graph_depth` value of 0 is used which sets the depth to infinity (see Section 3.1.5). This setting allows for the inclusion of every single nested function invocation into the tracer's output, starting from the system call interface down to the lowest level of the network stack before leaving the kernel's control domain and back[5].
In conclusion, cross-comparison of all tracing records obtained through the simultaneous execution of the stated tracing utilities enables the verification of the accuracy of metrics reported by the custom BPF programs under test.

[5] The `__x64_sys_mount()` system call e.g. causes 32351 kernel functions to be called between its opening and closing curly bracket, after which it returns control to userland code again.

4.4 Real-World Use Cases

Two use cases will be presented in this section. The use cases correspond to real-world work flows in the operational proceedings performed by the system administrators of both, the NAF/Grid[6] computing facilities on the one hand and the dCache storage system on the other. Their purpose is to demonstrate the sequence of actions currently taken by system administrators in case an issue is detected or reported. No BPF program at all is involved in these use cases. They are compared to the scenario that includes the application of the `nfs4_path_finderV` eBPF program in Section 5. The two use cases describe the following real-world scenarios:

1. An issue is first detected by an NAF user and reported to an NAF administrator
2. An issue is first detected on the dCache storage end by a dCache administrator

Figure 4.2 illustrates the sequence of actions taken for both use cases. All of the following information is the result of the author's personal communication with the administrators at DESY.

In use case number one, the user detects an issue with one or more of their jobs and informs the NAF administrator about it. Not in all cases important job metrics such as the username, UID or the worker node (WN) the job is running on are included in the user's notification. This means that additional time is spent for further communication with the user. The use case assumes that the username of the user is known to the administrator. A HTCondor query is started to obtain the name of the affected WN on which a potential issue with the user's job is assumed. At this point, the administrator has no information about the scope of the issue. This also applies to other potentially affected users, who have not noticed issues yet, as well as to other WNs. Consequently, the administrator's query to find the affected WN and the correct PID of the affected job is time consuming and has an uncertain outcome. The NAF administrator quits the query due to the lack of information compared to the magnitude of possibilities and returns to communicate this to the user who filed the issue in this use case. In the best case, the user runs only one single job. This job is quickly identified, and the slot it runs in is cleared. The actual issue is not necessarily resolved by the latter action. Other users potentially experience similar issues with their jobs on that WN. In the worst case scenario the whole WN needs to be rebooted, clearing all jobs off that machine.

[6] For the sake of brevity only the term *NAF* will be used in the text from now on. The Grid computation infrastructure is always included, though, if not mentioned otherwise.

The second use case describes a scenario with a lower time requirement. As seen in Section 2.9 the dCache administrator has more metrics available through the dCache admin interface than the NAF administrator has in the case an issue is detected. The administrator of the dCache storage system provides the IP address and port of the WN, the UID and the path to the file the NFSv4+ client on the WN in question is accessing. This is valuable information for the NAF administrator to start a query on the known WN in order to find the PIDs of the user's jobs. With these PIDs at hand, if still present by then, a long-winded query via the `/proc/<PID>/fd` filesystem path is started aimed at finally finding the corresponding path. Since the path is known (from the dCache end) at that time, the corresponding PID is the one that belongs to the faulty job which is subsequently cleared off the WN's slot.

Lastly, in case the dCache administrator perceives completely idle data movers which never delivered a single byte (high *LM* value) to the NFSv4+ client, but the storage pools are in a flawless state, an NFSv4+ client-server issue can be assumed. According to the dCache system administrator there are typically three levels of escalation for the recovery of an NFSv4+ client or server. The levels increase in their effect on the system from less disruptive to most disruptive:

1. The dCache admin interface provides a `kill-client` command. Despite the name, the command only triggers the client to reconnect to one or more NFSv4+ servers it previously was connected to.
2. A restart of the NFS door (metadata server).
3. A restart of the WN machine on which the NFSv4+ client resides.

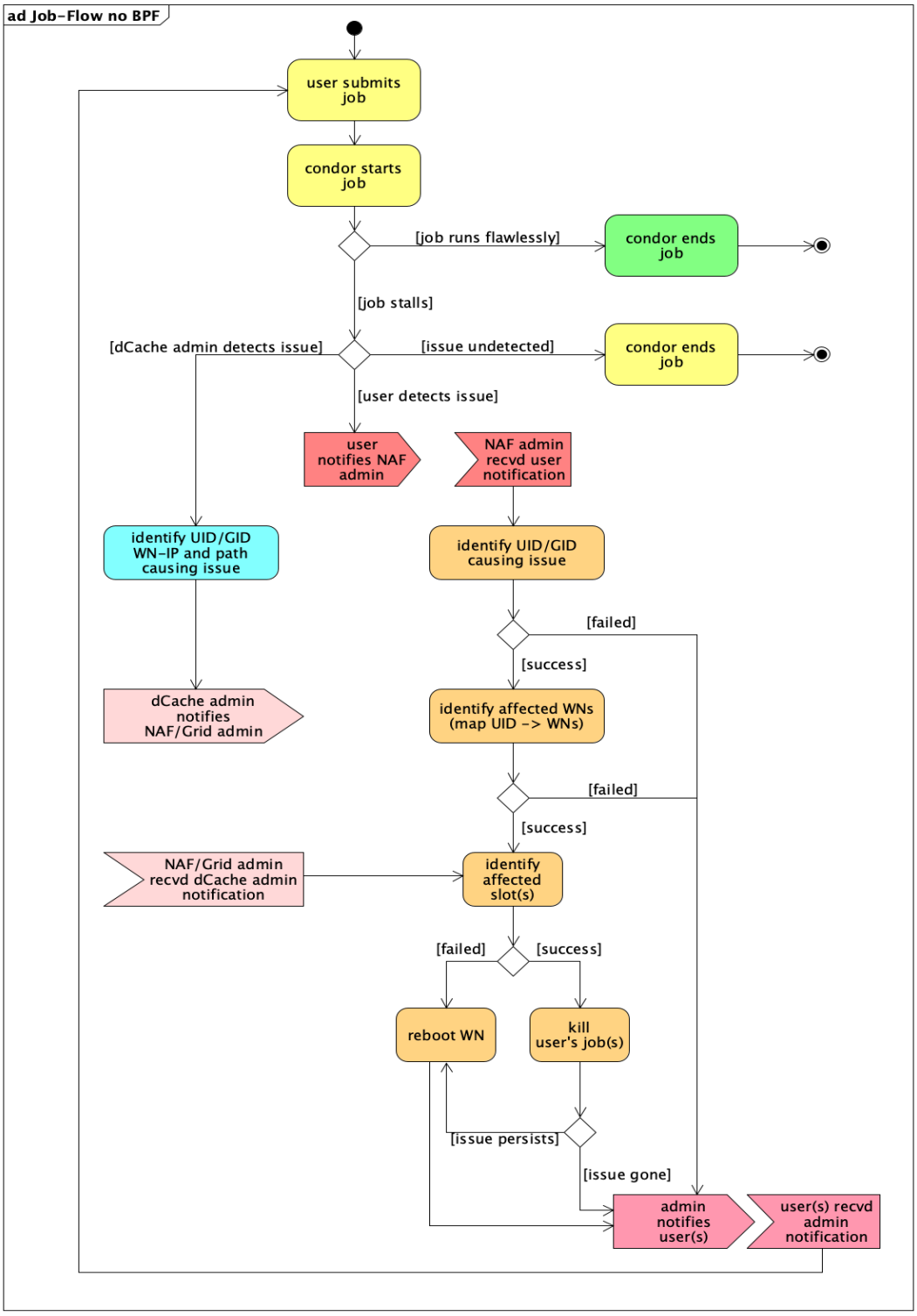

Figure 4.2: Activity diagram of a Job Workflow as described in use cases 1 and 2 (no custom BPF program involved)

5 Evaluation

This section presents the key findings of the present work with respect to the research objectives proposed in Section 1.5. An overview of the key findings is listed below.

- **Key Findings:**
    1. **Kernel Metrics Count:** A total of 33 different kernel metrics are made available to the system administrators at DESY through the application of the custom-built BPF programs
    2. **Time-Savings:** The time span from notification of an issue to the point where the system administrators obtain the relevant metrics for taking action is drastically reduced
    3. **User Process ID Tracking:** Experiments during the development process of the BPF programs reveal that the PID of a user process can be tracked down to the RPC layer of the kernel
    4. **NFSv4+ Bytes Per User PID:** As a consequence thereof, a BPF program is implemented that retrieves the amount of bytes sent and received with NFSv4+ on a per user process basis. This, to the best of the author's knowledge, did not exist so far

5.1 Kernel Metrics Made Available Through BPF

Table 5.1 lists the kernel metrics available to system administrators through the application of the three custom BPF programs. A total of 33 newly available metrics are now supplied. The comparison of available metrics on the computing end (NAF) with those on the storage end suggests a lack of balance between the two. This is certainly not accurate, since the new metrics are made available to every member of the administrative team. Apart from that, many of the new metrics, such as the RPC client ID or the open file descriptor value, either do not apply on both ends of the network or they are not of any interest for the pursuit of issues on either end.

Naturally, every single metric contributes to the resolution of an issue to a different degree. These shares in contribution are highly dependent on the issue in question. Applied to use case number one described in Section 4.4, where the administrator is left with the UID of the user only, the metric of the path to an open NFS share (in combination with the corresponding PID) contributes more to the process of localizing the WN and the exact slot than the knowledge of e.g. the ID of the RPC client involved.
The 33 metrics presented are the result of experimentation. They represent the outcome of research on the limit of what is feasible with the application of BPF technology. And they are considered only a starting point.

No.	Metric	NAF admin (BPF)	dCache admin (IF)
1	command name	+	-
2	time stamp	+	+
3	current CPU	+	-
4	PID of current task	+	-
5	TGID of current task	+	-
6	PID of RPC task owner	+	-
7	UID of current task	+	+
8	GID of current task	+	+
9	UID of RPC task owner	+	-
11	GID of RPC task owner	+	-
12	XID of RPC request/reply	+	-
13	RPC client ID	+	-
14	CGroup ID	+	-
15	access protocol name	+	+
16	access protocol number	+	-
17	access protocol version	+	+
18	transport protocol	+	-
19	open file descriptor	+	-
20	path string	+	+
21	NFSv4 server name	+	+
22	NFSv4 server port	+	+
23	NFSv4 server IP address	+	+
24	NFSv4 client name	+	+
25	NFSv4 bytes sent p.process	+	-
26	NFSv4 bytes received p.process	+	-
27	NFSv4+ client ID	-	+
28	NFSv4+ session ID	-	+
29	NFSv4+ state ID	-	+
30	NFSv4+ sequence ID	-	+
31	NFSv4+ transfer status	-	+
32	socket family	+	-
33	socket type	+	-
34	socket state	+	-
35	socket source IP address	+	-
36	socket source port	+	-
37	socket destination IP address	+	-
38	socket destination port	+	-

Table 5.1: Comparison of metrics obtained through BPF to available metrics through the dCache admin interface (IF).
LEGEND:
+ = available
- = not available yet/does not apply

5.2 Time-Savings During Issue Management

Currently, the only custom BPF program running in an NAF production context at DESY is the `nfs4_path_finderV`. While being executed on the worker nodes, its output file stream is fed into a data aggregation and visualization software called *Elastic Kibana* [91]. The `nfs4_path_finderV` program scans the file descriptors of every process scheduled on any of the multiple CPUs of a worker node. It reports the path, if applicable, to a file on an NFS share that the process has currently opened, together with its PID and TGID. Figure 5.1 shows the entirety of metrics the BPF program collects.


```
{
    "host": "grid-dev-sandro01.desy.de",
    "cmd": "vim",
    "timestp[us]":13856811940,
    "cpu": 13,
    "PID": 9502,
    "TGID": 9502,
    "UID": 1000,
    "GID": 51108,
    "cgroup_id": 9563,
    "fd": 3,
    "path":"/pnfs/desy.de/dot/volatile/test_nfs4_216"
}
```


Figure 5.1: A JSON formatted entry from the `nfs4_path_finderV` BPF program output

To measure the time the kernel needs to perform a listing of the `/proc/<PID>/fd` directory, the `ls` command is invoked together with the `time` utility on the Linux command shell as shown in Listing 5.1.

```
$ time ls /proc/943626/fd
  l-wx------. 1 root root 64 Dec 26 20:46 0 -> /var/log/sssd/
  sssd_sudo.log
  ... (+ 17 more)
  ...
  sys  0m0.011s
```

Listing 5.1: Timing the `ls /proc/<PID>/fd` command

The $0.011\,s$ that the command spends in kernel mode is the shortest time span out of 25 measurements with approximately 18 ± 1 file descriptor lookups. The measurements are taken on an idle machine with respect to user processes (other than the listing) and no open files

to NFS shares are involved. For comparison, table 5.2 shows the average time needed by the `nfs4_path_finderV` to perform the same task of scanning 18 file descriptors on a busy production NAF worker node. For each scanned file descriptor an entry as the one shown in Figure 5.1 is generated in the output stream. The time to perform the JSON-formatted entry in the userland part of the `nfs4_path_finderV` is not part of the measurement.

ls /proc/<PID>/fd	**nfs4_path_finderV**
$0.011\,s$	$0.000944\,s$

Table 5.2: Comparison of time needed for the scanning of 18 file descriptors in the kernel

More important for the system administrators' work than the result of the measurement showing that the BPF program is processed about eleven times faster is the fact that it constantly collects the metrics and pipes them to Elastic Kibana without any human interaction.
In the first use case scenario the NAF administrator gains a collection of job PIDs from the HTCondor query to identify the WNs that potentially expose the issue in question. Consequently, every single PID must be processed manually on the command shell or in a scripted manner to finally find a path to an NFS share. By the time this is executed to completion, the process causing the issue might be already removed or file descriptors might be unavailable. Under these conditions, the administrator is not able to perform any further queries and has to quit the issue tracking process.
In contrast to the above use case, now a scenario is possible in which the `nfs4_path_finderV` program is being executed in the WN's kernel, constantly piping metrics to Elastic Kibana. The first action the NAF administrator takes is to query Kibana for open paths to NFS shares given the UID at hand.

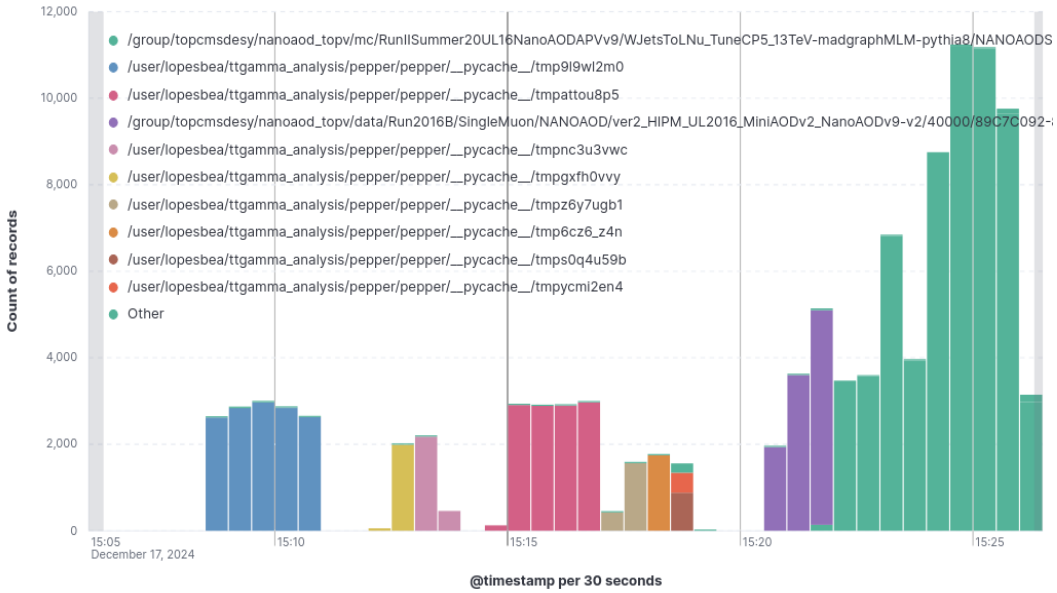

Figure 5.2: Kibana query result showing NFSv4+ paths opened by processes with PIDs in the range of 2505-2535 and their scheduling activity from user with UID 35XYZ on the batch1568 worker node

Figure 5.2 shows the visualization of a Kibana query. It shows how often processes selected by PIDs in the range of 2505 through 2535 were scheduled on CPU cores of the batch1568 worker node over time. The opened NFSv4+ paths are listed partly as well. They all belong to one user with a given UID. With the overview of activity patterns of processes over time, the administrator is able to recognize issue-causing patterns more easily. Patterns resembling no activity over longer periods hint at stalling I/O requests, while unusually high bursts of activity could hint at a forthcoming storage pool overload [18]. Especially when the same file on an NFS share is opened by multiple user's processes performing file I/O concurrently. Yet, since the information provided by the `nfs4_path_finderV` is already available, the Kibana query performed to find out which other PIDs have the same path to a file opened is a matter of seconds.

For comparison, time measurements of the first use case performed by the NAF system administrator reveal an average time requirement of approximately $480\,s$. This time value includes the HTCondor query to find out whether any jobs from a user with a given UID are still running and, if this is the case, on which WNs they are being executed. Also included in the measured time span is the execution of the `ls /proc/<PID>/fd` command on the command shell as described above. In contrast, the time required to perform a Kibana query yielding the same results takes an average time of approximately $61\,s$. This corresponds to a time-saving of approximately $87\,\%$.

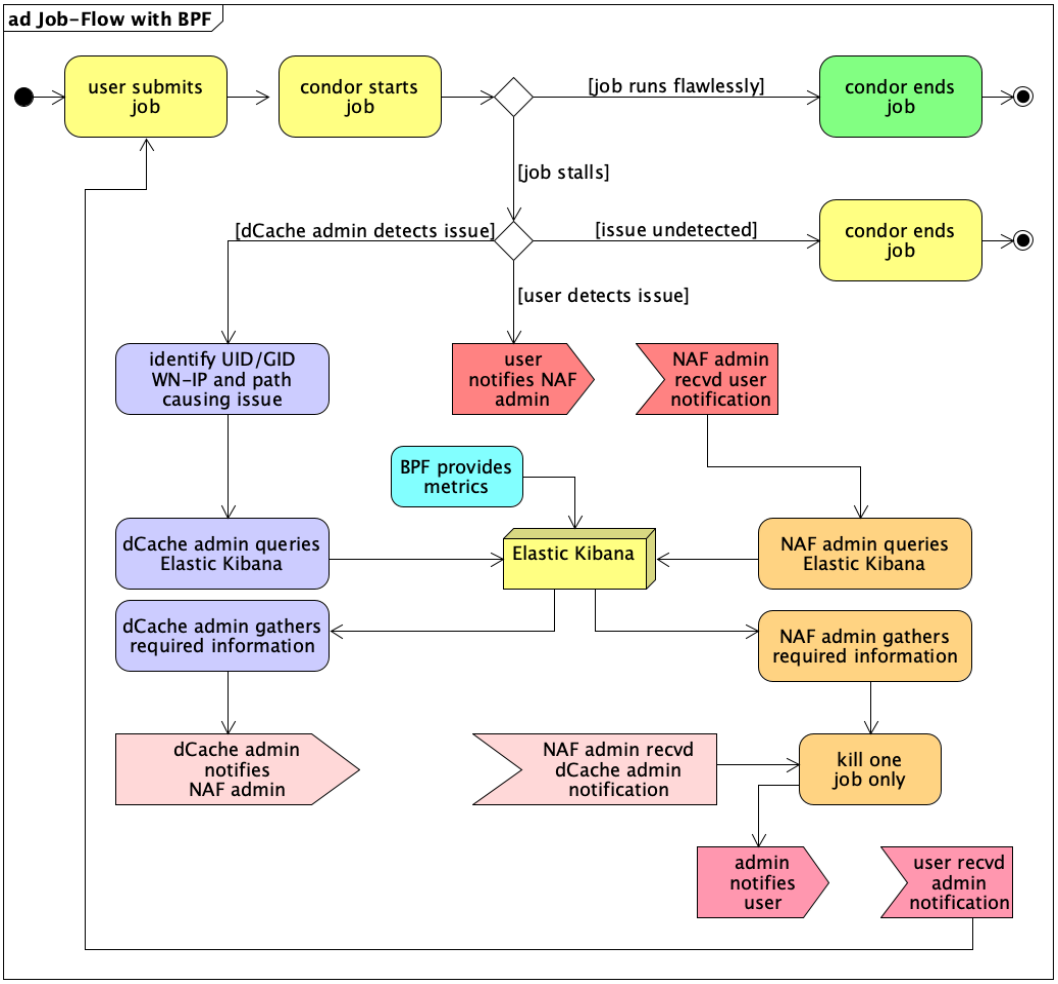

Figure 5.3: Activity diagram of a Job Workflow with custom BPF program involved

5.3 User Process ID Tracking

The discovery that the PID of the user process is traceable down to the kernel's NFS and RPC layer is considered a success. It paved the way for the development of the `nfs4_byte_picker` BPF program. There was a degree of uncertainty associated with this question during the research phase to this thesis. At that time, from the mere study of the kernel sources, it was not fully clear which task would assign its `tgid` field value to the `tk_owner` field of the initiated `struct rpc_task` (as explained in Section 4.2.1). The discovery emerged through experimentation during the development phase of the `nfs4_byte_picker` program and is verified by empirical evidence. This evidence derives from

the study of hundreds of experimentation records that contained output entries as the one depicted in Figure 5.4 by comparing the PID/TGID (of *current*) to the PID of the RPC task owner. Yet, there is the rare observation of a kernel thread being both, *current* and RPC task owner. Thus, the question about which circumstances determine whether the `rpc_init_task()` routine is invoked by the userland thread or by a kernel thread leaving its PID behind, is still left unanswered.

5.4 NFSv4+ Bytes Per User PID

Although the custom `nfs4_byte_picker` program is not running in production at DESY, its functionality has been visualized with the help of Kibana plots. As explained in 4.2.1 the `nfs4_byte_picker` program collects the bytes sent and received by the client via the NFSv4+ protocol. Additionally, and this can be regarded its real strength, it associates those transferred bytes to the PID of the originating user process. The full set of metrics derived from the `nfs4_byte_picker` program is shown in Figure 5.4.


```
{
    "host": "grid-dev-sandro01.desy.de",
    "cmd": "kworker/u128:0",
    "timestp_xmit_start[us]": 19510944953,
    "timestp_xmit_end[us]": 19510965117,
    "cpu": 7,
    "PID": 10742,
    "TGID": 10742,
    "UID": 0,
    "GID": 0,
    "cgroup_id": 1,
    "rpc_task_owner_pid": 12503,
    "rpc_task_owner_uid": 1000,
    "rpc_task_owner_gid": 51108,
    "xid_call": 118395688,
    "xid_rply": 118395688,
    "xprt_protocol": "TCP",
    "protocol_name": "nfs",
    "protocol_number": 100003,
    "protocol_version": 4,
    "server_name": "dcache-dot-door01.desy.de",
    "server_port": 2049,
    "server_ip_addr": "131.169.223.60",
    "client_name": "grid-dev-sandro01.desy.de",
    "rpc_client_id": 3,
    "bytes_sent": 388,
    "bytes_rcvd": 532
}
```


Figure 5.4: JSON formatted entry from the `nfs4_byte_picker` BPF program output

As a proof of concept the plot displayed in Figure 5.5 was generated. It shows the course of data transferred over time between a worker node and a NFSv4+ data server on a dCache storage node. The transfer comprises a request for large tarball files from the NFS share. Once transferred, the tarballs are extracted locally on the worker node. The extracted files are subsequently sent back to the storage. This task is executed by one single user process on the worker node. The shown plot is a result of an applied filter in Kibana. It is filtered by PID and hostname and set to display the sum of each of the two metrics from the `nfs4_byte_picker` program output, namely the *bytes_sent* and the *bytes_rcvd* metric, respectively. The process was paused at 11:39 a.m. and resumed at 11:43 a.m. The dips in the graphs of the shown plots reflect this interruption.

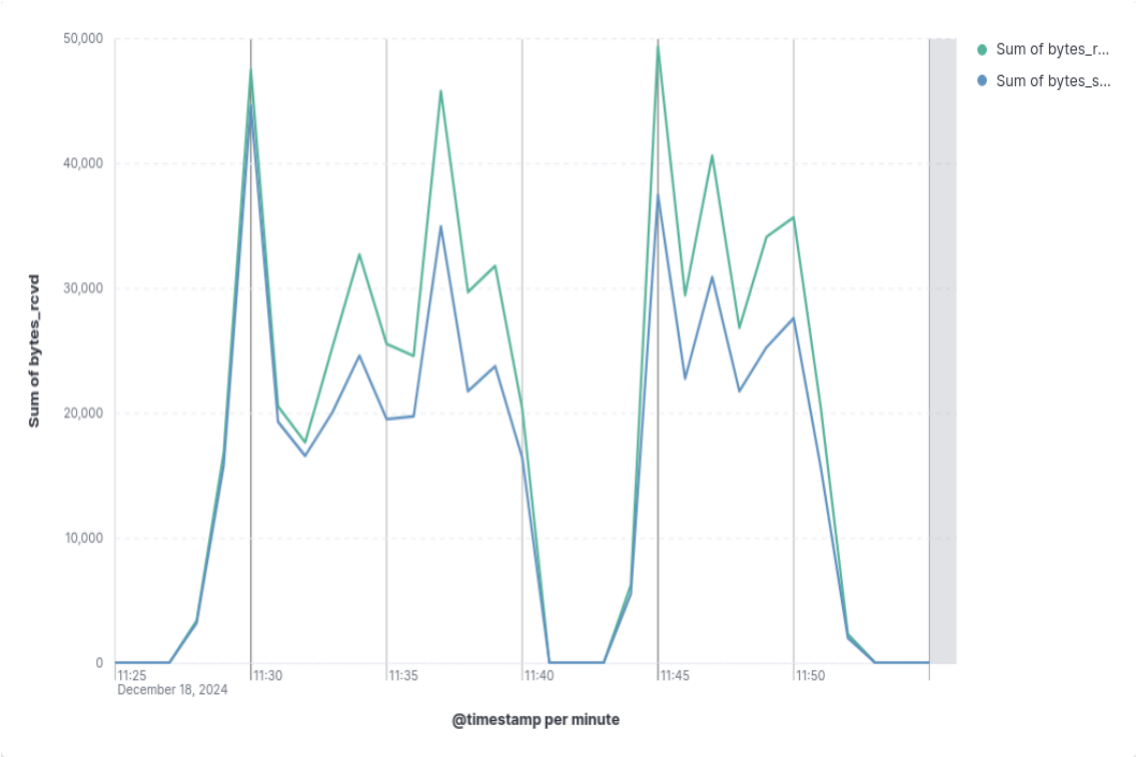

Figure 5.5: A Kibana query result showing the NFSv4+ bytes sent (blue) and received (green) by a process with given PID on the batch1255 worker node

Figure 5.6 shows the total amount of bytes that is transferred across the network interfaces of the worker node. A comparable progression in the amount of transferred data over the same time span can be recognized here as well.

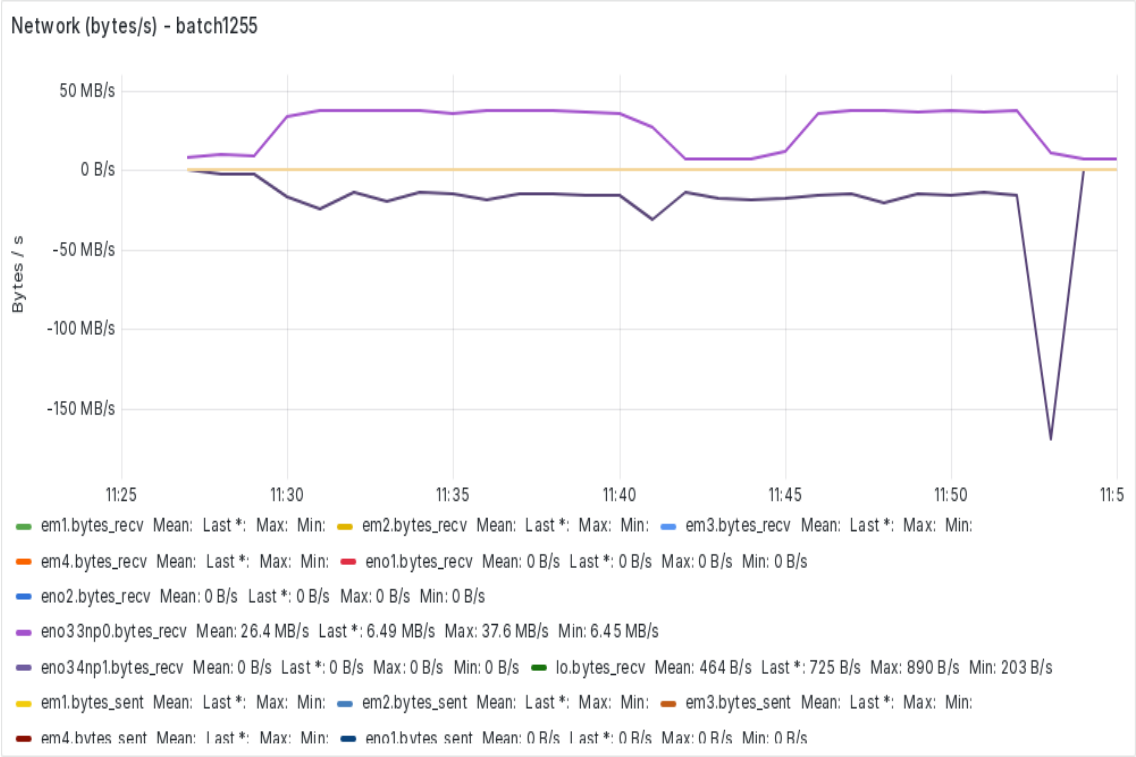

Figure 5.6: Kibana query result showing the sum of all bytes sent (lower) and received (upper) through the network cards of the batch1255 worker node

6 Discussion and Conclusion

This section concludes this thesis by discussing the implications and limitations of the achieved objectives. Additionally, it presents a brief outlook on future application and extension.

The outcome of the research presented in Section 4 and 5 seems promising with respect to its potential in accelerating the administrators' process of narrowing down issues to their actual origin. Especially, the aggregation and visualization of the obtained metrics with the help of Elastic Kibana opens up a faster and more intuitive way of interpreting them and recognizing inherent patterns. Whether it is a storage pool under heavy load or a stale NFSv4+ client, most of the issues in a complex environment show characteristic patterns. The capability of recognizing the cause of an issue by studying its pattern can be learned and comes naturally with experience [67]. Still, the basis for every meaningful pattern to emerge is the presence of meaningful metrics. Which of the metrics obtained by means of the three custom BPF programs are to be considered more meaningful than others is a question that needs further experimentation and evaluation. But the results show that the initially proposed combination of a user process' PID, UID and the path to open files on NFS shares favorably contribute to the acceleration in issue management. Especially, the simultaneous gathering of the stated PID and path information in concert with the transferred bytes per user process gives rise to a facilitated pattern recognition process as the Kibana plots in Section 5.4 show.

As already stated, the total of 33 new metrics can be considered a valuable starting point for a future extension of the list. Through the high degree of customization possible in BPF programming, new metric queries can be added to the existing programs. A limitation to the number of metrics gathered within one BPF program is the complexity constraint enforced by the BPF verifier. Although seemingly impeding at first, the BPF verifier's constraints have a good reason: They protect the aspiring BPF developer from developing BPF programs that can potentially throttle the otherwise most efficiently performing Linux kernel. Despite the fact that the verifier's constraints guarantee a minimal performance overhead [2], the accumulation of time penalties caused by the simultaneous application of multiple BPF programs can be non-negligible. Especially in high-throughput-computing environments such as the NAF or the Grid at DESY, this has to be taken into consideration. The exploration of the

exact circumstances that affect the performance of BPF programs in conjunction with measurements of performance penalties caused by them could be the basis for future research. In addition, a future extension of the already initiated work could comprise the development of BPF programs for the other supported access protocols such as, e.g. WebDAV.
To gather further results and explore deeper insights in the potential advantages of using BPF technology a longer in-production test phase will be needed. For the purpose of this thesis and the research objectives within the frame of this work, however, the results clearly confirm the benefit of leveraging BPF technology for the acceleration of issue management performed by system administrators at DESY. Furthermore, they attest the feasibility of obtaining and attributing the amount of bytes transferred via the NFSv4+ protocol to the PID of the user process that originally requested the transferred data. This establishes the foundation for the future development of NFS-related BPF probes that can be deployed on a per-user-process basis. Moreover, the creation of new BPF probes is accelerated by this essential step. The custom-built BPF programs created during the work on this thesis are publicly available[1] and hopefully helpful to other members of the community as well.

[1] The codebase for the three BPF programs is freely available at https://github.com/sandro108/ebpf_programs under the Apache License 2.0. Contributions are most welcome.

References

[1] Wireshark Foundation: *Wireshark - Network Protocol Analyzer* . – URL https://www.wireshark.org. – Retrieved on: 30.11.2024

[2] Gregg, B.: *BPF Performance Tools*. Addison-Wesley Professional, 2019. – ISBN 978-0-13-655482-0

[3] Mkrtchyan, T.: *personal communication*. – 05.11.2024

[4] unknown, DESY Website: *The decoding of matter*. 2024. – URL https://www.desy.de/about_desy/desy/index_eng.html. – Retrieved on: 03.01.2025

[5] unknown, DESY Website: *Photon science*. 2024. – URL https://www.desy.de/research/photon_science/index_eng.html. – Retrieved on: 03.01.2025

[6] Particle physics Research Division at DESY: *Any Light Particle Search*. unknown. – URL https://alps.desy.de/our_activities/axion_wisp_experiments/alps_ii. – Retrieved on: 01.01.2025

[7] unknown, DESY Website: *Particle physics*. 2024. – URL https://www.desy.de/research/particle_physics/index_eng.html. – Retrieved on: 03.01.2025

[8] Del Rosso, A.: *How CERN IT keeps up with the data deluge*. 2024. – URL https://home.cern/news/news/computing/how-cern-it-keeps-data-deluge. – Retrieved on: 01.01.2025

[9] unknown, DESY Website: *Documentation for the Maxwell HPC Cluste*. 2024. – URL https://docs.desy.de/maxwell/. – Retrieved on: 03.01.2025

[10] Haupt, A. ; Kemp, Y. ; Nowak, F.: Evolution of Interactive Analysis Facilities: from NAF to NAF 2.0. In: *Journal of Physics: Conference Series* 513 (2014), 06, S. 032072

[11] Mkrtchyan, T. ; Chitrapu, K. ; Garonne, V. ; Litvintsev, D. ; Meyer, S. ; Millar, P. ; Morschel, L. ; Rossi, A. ; Sahakyan, M.: dCache: Interdisciplinary storage system. In: *EPJ Web Conf.* 251 (2021), S. 02010. – URL https://doi.org/10.1051/epjconf/202125102010

[12] Hartmann, T.: *personal communication*. – 23.07.2024

[13] Cordeiro, C. ; Southwick, D. ; Giordano, D. ; Barbet, J.-M. ; Medeiros, M. F.: *HEP Benchmark Suite*. – URL https://gitlab.cern.ch/hep-benchmarks/hep-benchmark-suite. – Retrieved on: 22.04.2024

[14] Alef, M. ; Cordeiro, C. ; Salvo, A. D. ; Girolamo, A. D. ; Field, L. ; Giordano, D. ; Guerri, M. ; Schiavi, F. C. ; Wiebalck, A.: Benchmarking Cloud Resources for HEP. In: *Journal of Physics: Conference Series* 898 (2017), oct, Nr. 9, S. 092056. – URL https://dx.doi.org/10.1088/1742-6596/898/9/092056

[15] Beyer, C. ; Bujack, S. ; Dietrich, S. ; Finnern, T. ; Flemming, M. ; Fuhrmann, P. ; Gasthuber, M. ; Gellrich, A. ; Guelzow, V. ; Hartmann, T. ; Reppin, J. ; Kemp, Y. ; Lewendel, B. ; Schluenzen, F. ; Schuh, M. ; Sternberger, S. ; Voss, C. ; Wengert, M.: Beyond HEP: Photon and accelerator science computing infrastructure at DESY. In: *EPJ Web Conf.* 245 (2020), S. 07036. – URL https://doi.org/10.1051/epjconf/202024507036

[16] HTCondor Community: *HTCondor Overview*. – URL https://htcondor.org/htcondor/overview/. – Retrieved on: 23.04.2024

[17] Sion, R.: Strong WORM. In: *2008 The 28th International Conference on Distributed Computing Systems*, URL https://dl.acm.org/doi/10.1109/ICDCS.2008.20, 2008, S. 69–76

[18] Gebhardt, L.: *Erkennung von verfügbarkeitsgefährdendem Nutzerverhalten im dCache-System*, HAW Hamburg, Bachelorarbeit, 2023

[19] Christians, F.: *Anomaly Detection in verteilten Speichersystemen: Vorbereiten einer MAPE Loop*, HAW Darmstadt, Bachelorarbeit, 2024

[20] Dong, X. ; Liu, Z.: Multi-dimensional detection of Linux network congestion based on eBPF. In: *2022 14th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, 2022, S. 925–930

[21] Lian, Z. ; Li, Y. ; Chen, Z. ; Shan, S. ; Han, B. ; Su, Y.: eBPF-based Working Set Size Estimation in Memory Management. In: *2022 International Conference on Service Science (ICSS)*, 2022, S. 188–195

[22] Laforet, V. ; Lozi, J.-P. ; Lawall, J.: BPF Hybrid Lock: Using eBPF to communicate with the scheduler. (2023). – URL https://inria.hal.science/hal-04266815

[23] Zhang, X. ; Liu, Z. ; Bai, J.: Linux Network Situation Prediction Model Based on eBPF and LSTM. In: *2021 16th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*, 2021, S. 551–556

[24] Deri, L. ; Sabella, S. ; Mainardi, S. ; Degano, P. ; Zunino, R.: Combining System Visibility and Security Using eBPF. In: *ITASEC* Bd. 2315, 2019

[25] Gregg, B.: *Systems Performance*. Addison-Wesley Professional, 2021. – ISBN 978-0-13-682015-4

[26] Dubuc, T. ; Vicat-Blanc, P. ; Olivier, P. ; Callau-Zori, M. ; Hubert, C. ; Tchana, A.: Tracklops: Real-Time NFS Performance Metrics Extractor. In: *Proceedings of the 4th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. New York, NY, USA : Association for Computing Machinery, 2024 (CHEOPS '24), S. 1–8. – URL https://doi.org/10.1145/3642963.3652202. – ISBN 9798400705380

[27] Bovet, D. ; Cesati, M.: *Understanding The Linux Kernel*. Oreilly & Associates Inc, 3rd edition, 2005. – ISBN 978-0-596-00565-8

[28] Love, M.: *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. – ISBN 978-0-672-32946-3

[29] Billimoria, K.N.: *Linux Kernel Programming: A Comprehensive Guide to Kernel Internals, Writing Kernel Modules, and Kernel Synchronization*. Packt Publishing, Limited, 2024 (Expert insight). – ISBN 9781803232225

[30] Enberg, P. and Lameter, C.: */sys/kernel/slab/<cache>/ Documentation*. 2007-2011. – URL https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-kernel-slab. – Retrieved on: 23.11.2024

[31] Corbet, J.: *Per-CPU variables and the realtime tree*. 2011. – URL https://lwn.net/Articles/452884/. – Retrieved on: 25.11.2024

[32] 0xAX and Community: *Linux Insides*. last commit in 2020. – URL https://github.com/0xAX/linux-insides/blob/master/Concepts/linux-cpu-1.md. – Retrieved on: 26.11.2024

[33] Stallman, R. M. ; GCC Developer Community the: *Using the GNU Compiler Collection for GCC Version 14.2.0*. GNU Press, Free Software Foundation, Boston, USA, 2024

[34] Advanced Micro Devices Inc: *AMD64 Architecture Programmer's Manual Volume 1, Rev. 3.23*. 2020. – URL https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24592.pdf. – Retrieved on: 27.11.2024

[35] Lettieri, G: *Stack Canaries*. 2023. – URL https://lettieri.iet.unipi.it/hacking/canaries.pdf. – Retrieved on: 26.11.2024

[36] Mauerer, W.: *Professional Linux Kernel Architecture*. John Wiley & Sons., 2008. – ISBN 978-04-703-4343-2

[37] Torvalds, L.: *RE: Userspace breakage*. 2005. – URL https://lore.kernel.org/lkml/CA+55aFy98A+LJK4+GWMcbzaa1zsPBRo76q+ioEjbx-uaMKH6Uw@mail.gmail.com/. – Retrieved on: 26.10.2024

[38] Torvalds, L.: *WE DO NOT BREAK USERSPACE!* 2012. – URL https://lore.kernel.org/all/Pine.LNX.4.64.0512291451440.3298@g5.osdl.org/T/#u. – Retrieved on: 26.10.2024

[39] Brown, N.: *Object-oriented design patterns in the kernel*. 2011. – URL https://lwn.net/Articles/444910/. – Retrieved on: 26.10.2024

[40] Charatan, Q. ; Kans, A.: *Java in Two Semesters*. Springer Nature Switzerland AG 2019, 2019. – URL https://doi.org/10.1007/978-3-319-99420-8. – ISBN 978-3-319-99419-2

[41] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 978-03-217-0069-8

[42] Venners, B.: *Design Principles from Design Patterns A Conversation with Erich Gamma, Part III*. 2005. – URL https://www.artima.com/articles/design-principles-from-design-patterns. – Retrieved on: 25.10.2024

[43] Kleiman, S. R.: Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In: *Proceedings of the USENIX Summer Conference, Altanta, GA, USA, June 1986*, USENIX Association, 1986, S. 238–247

[44] Brown, N.: *Filesystems in the Linux kernel » Pathname lookup*. 2015. – URL https://www.kernel.org/doc./html/next/filesystems/path-lookup.html. – Retrieved on: 19.11.2024

[45] Gooch, R.: *Filesystems in the Linux kernel » Overview of the Linux Virtual File System*. 2005. – URL https://www.kernel.org/doc/html/v5.4/filesystems/vfs.html. – Retrieved on: 19.11.2024

[46] Torvalds, L.: *comp.os.mimix*. 1991. – URL https://web.archive.org/web/20130509134305/http://groups.google.com/group/comp.os.minix/msg/b813d52cbc5a044b. – Retrieved on: 01.01.2025

[47] Sandberg, R. ; Goldberg, D. ; Kleiman, S. L. ; Walsh, D. ; Lyon, B.: Design and implementation of the Sun network filesystem. In: *USENIX* (1985), S. 119–130. – URL https://api.semanticscholar.org/CorpusID:61413305

[48] Haynes, T. ; Noveck, D.: *Network File System (NFS) Version 4 Protocol*. RFC 7530. März 2015. – URL https://www.rfc-editor.org/info/rfc7530

[49] Internet Engineering Task Force (IETF): *RFC 8881, Network File System (NFS) Version 4 Minor Version 1 Protocol*. 2020. – URL https://datatracker.ietf.org/doc/rfc8881/. – Retrieved on: 08.11.2024

[50] Haynes, T.: *Network File System (NFS) Version 4 Minor Version 2 Protocol*. RFC 7862. November 2016. – URL https://www.rfc-editor.org/info/rfc7862

[51] Eisler, M.: *XDR: External Data Representation Standard*. RFC 4506. Mai 2006. – URL https://www.rfc-editor.org/info/rfc4506

[52] Thurlow, M.: *RPC: Remote Procedure Call Protocol Specification Version 2*. RFC 5531. Mai 2009. – URL https://www.rfc-editor.org/info/rfc5531

[53] Myklebust, T. ; Lever, C.: *Towards Remote Procedure Call Encryption by Default*. RFC 9289. September 2022. – URL https://www.rfc-editor.org/info/rfc9289

[54] Eisler, M.: *IANA Considerations for Remote Procedure Call (RPC) Network Identifiers and Universal Address Formats*. RFC 5665. Januar 2010. – URL https://www.rfc-editor.org/info/rfc5665

[55] IANA Internet Assigned Numbers Authority: *Service Name and Transport Protocol Port Number Registry*. – URL https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?&page=37. – Retrieved on: 28.11.2024

[56] Millar, A. ; Baranova, T. ; Behrmann, G ; Bernardt, C ; Fuhrmann, P. ; Litvintsev, D. ; Mkrtchyan, T. ; Petersen, A ; Rossi, A.: DCache, agile adoption of storage technology. In: *Journal of Physics Conference Series* 396 (2012), 12

[57] Gibson, G.: *pNFS Problem Statement*. https://datatracker.ietf.org/doc/html/draft-gibson-pnfs-problem-statement-00.txt. 2004

[58] Fuhrmann, P.: A perfectly normal namespace for the DESY open storage manager, URL http://www-zeuthen.desy.de/CHEP97/paper/409.ps, 1997

[59] Mkrtchyan, T. ; Chitrapu, K ; Litvintsev, D. ; Meyer, S. ; Millar, P. ; L. Morschel, L. ; Rossi, A. ; Sahakyan, M.: DB Back-ended Filesystem for Science. In: *Proceedings of the 35th GI-Workshop on Foundations of Databases* (2024), Mai. – URL https://ceur-ws.org/Vol-3710/paper9.pdf

[60] dCache Developers Team: *dCache -The Book- A general guide for administrators*. dcache.org, The Book v10.2, 2024. – URL https://www.dcache.org/manuals/Book-10.2

[61] Halevy, B. ; Haynes, T.: *Parallel NFS (pNFS) Flexible File Layout*. RFC 8435. August 2018. – URL https://www.rfc-editor.org/info/rfc8435

[62] dCache Developers Team: *About us*. 2007. – URL https://dcache.org/about/. – Retrieved on: 02.01.2025

[63] Fuhrmann, P.: *dCache*. 2007. – URL https://indico.cern.ch/event/20080/contributions/1484377/subcontributions/134209/attachments/300209/419545/lcg-reliable-services-meeting-261107.pdf. – Retrieved on: 06.12.2024

[64] dCache Developers Team: *dCache Github Repository*. – URL https://github.com/dCache/dcache. – Retrieved on: 08.12.2024

[65] Mkrtchyan, T. ; Adeyemi, O. ; Fuhrmann, P. ; Garonne, V. ; Litvintsev, D. ; Millar, A. ; Rossi, A. ; Sahakyan, M. ; Starek, J. ; Yasar, S.: dCache - storage for advanced scientific use cases and beyond. In: *EPJ Web of Conferences* 214 (2019), 01, S. 04042

[66] The Apache Software Foundation: *Apache Zookeeper*. – URL https://zookeeper.apache.org/. – Retrieved on: 08.12.2024

[67] Voss, C.: *personal communication*. – 11.10.2024

[68] Olszewski, M. ; Mierle, K. ; Czajkowski, A. ; Brown, A. D.: JIT instrumentation: a novel approach to dynamically instrument operating systems. 41 (2007), Nr. 3. – URL https://doi.org/10.1145/1272998.1273000. – ISSN 0163-5980

[69] Zannoni, E.: *An Introduction To Linux Tracing And Its Concepts*. 2021. – URL https://www.linuxfoundation.org/webinars/an-introduction-to-linux-tracing-and-its-concepts. – Retrieved on: 25.05.2024

[70] Cambridge University Press & Assessment 2024: *The Cambridge Essential American English Dictionary*. – URL https://dictionary.cambridge.org/dictionary/essential-american-english/. – Retrieved on: 20.12.2024

[71] Bootlin - embedded linux and kernel engineering: *Linux debugging, profiling and tracing training*. – URL https://bootlin.com/doc/training/debugging/debugging-slides.pdf. – Retrieved on: 20.12.2024

[72] Rostedt, S.: *ftrace - Function Tracer*. – URL https://www.kernel.org/doc/html/v5.14/trace/ftrace.html. – Retrieved on: 02.11.2024

[73] Corbet, J.: *2.6.27: what's coming*. 2008. – URL https://lwn.net/Articles/289990/. – Retrieved on: 25.05.2024

[74] Gebai, M. ; Dagenais, M. R.: Survey and Analysis of Kernel and Userspace Tracers on Linux. In: *ACM Computing Surveys (CSUR)* 51 (2018), S. 1 – 33. – URL https://api.semanticscholar.org/CorpusID:4556704

[75] Desnoyers, M.: *Using the Linux Kernel Tracepoints*. unknown. – URL https://docs.kernel.org/trace/tracepoints.html. – Retrieved on: 23.12.2024

[76] Hiramatsu, M.: *Kernel Probes (Kprobes)*. unknown. – URL https://docs.kernel.org/trace/kprobes.html. – Retrieved on: 25.05.2024

[77] Mavinakayanahalli, A. ; Panchamukhi, P. ; Keniston, J. ; Keshavamurthy, A. S. ; Hiramatsu, M.: Probing the Guts of Kprobes, URL https://api.semanticscholar.org/CorpusID:221597225, 2010

[78] Corbet, J.: *BPF: the universal in-kernel virtual machine*. 2014. – URL https://lwn.net/Articles/599755/. – Retrieved on: 24.12.2024

[79] Jia, J. ; Zhu, Y. ; Williams, D. ; Arcangeli, A. ; Canella, C. ; Franke, H. ; Feldman-Fitzthum, T. ; Skarlatos, D. ; Gruss, D. ; Xu, T.: *Programmable System Call Security with eBPF*. 2023. – URL https://arxiv.org/abs/2302.10366

[80] Hadi, H. J. ; Adnan, M. ; Cao, Y. ; Hussain, F. B. ; Ahmad, N. ; Alshara, M. A. ; Javed, Y.: iKern: Advanced Intrusion Detection and Prevention at the Kernel Level Using eBPF. In: *Technologies* 12 (2024), Nr. 8. – URL https://www.mdpi.com/2227-7080/12/8/122. – ISSN 2227-7080

[81] Corbet, J.: *The BPF system call API, version 14*. 2014. – URL https://lwn.net/Articles/612878/. – Retrieved on: 24.12.2024

[82] Kernelnewbies Community: *Linux 3.18 has been released on Sun, 7 Dec 2014*. 2017. – URL https://kernelnewbies.org/Linux_3.18. – Retrieved on: 24.12.2024

[83] Alden, D.: *Modernizing BPF for the next 10 years*. 2024. – URL https://lwn.net/Articles/977013/. – Retrieved on: 24.12.2024

[84] Kernel Developer Community: *HOWTO interact with BPF subsystem*. – URL https://docs.kernel.org/bpf/bpf_devel_QA.html. – Retrieved on: 05.01.2025

[85] Linux Kernel Developer Community: *libbpf Overview*. unknown. – URL https://docs.kernel.org/bpf/libbpf/libbpf_overview.html. – Retrieved on: 25.12.2024

[86] Go Developer Community: *The eBPF Library for Go*. 2023. – URL https://ebpf-go.dev/. – Retrieved on: 25.12.2024

[87] Fleming, M.: *An introduction to the BPF Compiler Collection*. 2017. – URL https://lwn.net/Articles/742082/. – Retrieved on: 25.12.2024

[88] OpenSearch Developer Community: *The OpenSearch project*. 2024. – URL https://github.com/opensearch-project. – Retrieved on: 25.12.2024

[89] Rice, L.: *Learning eBPF*. O'Reilly, 2023. – ISBN 978-1098135126

[90] Nakryiko, A.: *BPF CO-RE reference guide*. 2021. – URL https://nakryiko.com/posts/bpf-core-reference-guide/. – Retrieved on: 26.12.2024

[91] Elastic - The Search AI Company: *Entdecken, iterieren und beheben mit ES|QL auf Kibana*. – URL https://www.elastic.co/de/kibana. – Retrieved on: 05.01.2025

[92] The kernel development community: *Filesystems in the Linux kernel » Filesystem Mount API*. 2016. – URL https://www.kernel.org/doc/html/v5.14/filesystems/mount_api.html#the-filesystem-context. – Retrieved on: 09.11.2024

Acknowledgments

The author wishes to express his gratitude to the following individuals:

Thomas Hartmann and Christian Voss for their support at DESY. Philine Pommerencke, Luca Gebhardt, Simon Boehling and Aaron Friedenberg for just being there. Nadja Grizzo and Tigran Mkrtchyan for their patience and Heike Peper for everything else including the above.

A Appendix

A.1 Expansion Macros

```
1  #define __pcpu_type_8 u64
2  #define __pcpu_op2_8(op, src, dst) op "q " src ", " dst
3  #define __percpu_arg(x)   __percpu_prefix "%" #x
4  #ifdef CONFIG_SMP
5  #define __percpu_prefix   "%%"__stringify(__percpu_seg)":"
6  #ifdef CONFIG_X86_64
7  #define __percpu_seg    gs
8  #define __pcpu_reg_8(mod, x) mod "r" (x)
```

Listing A.1: Definitions in `/arch/x86/include/asm/percpu.h` used to expand the `percpu_stable_op` macro (see 2.8)

A.2 Mounting a NFSv4+ Share

In order to demonstrate an example of the ability the VFS layer has in switching control to the correct filesystem implementation, a mount request for a NFSv4+ share issued by a user on the Linux command shell is followed from userland into kernelspace and described in more detail here. By utilizing the shell's system call tracing utility `strace`[1], the userland section of the mount process can be traced and logged. The pursuit starts with a mount(8)[2] command issued from within a Linux command shell, like the `bash`.

```
# strace mount -o vers=4.1 dcache-door01.desy.de:/pnfs /pnfs
```

Listing A.2: The mount(8) command issued from the Linux command shell.

Listing A.3 shows the terminal output caused by invocation of the `strace` command issued in Listing A.2. As can be seen in line 1 of Listing A.3, a new user process with PID 9893 was created. It executes an `execve`, which in turn executes the `mount` binary that lives in the `/usr/bin/` directory. It causes `openat()` syscalls into libraries like `libmount.so.1` and `libc.so.6`, searching for the appropriate implementation of the `mount()` syscall to be used for the transition into kernelspace. After cloning itself into a child process with PID 9894, `execve` executes a more specific mount binary named mount.nfs(8) from the `/sbin/` directory. The manual pages found with `man 8 mount.nfs` state the following information about this NFS mount helper command: «mount.nfs is a part of nfs(5) utilities package, which provides NFS client functionality. mount.nfs is meant to be used by the mount(8) command for mounting NFS shares.» And furthermore about the synopsis of the command: «mount.nfs remotetarget dir [...][-o options]» and a brief description of the arguments: «remotetarget is a server share usually in the form of servername:/path/to/share. dir is the directory onto which the file system is to be mounted.» All of the stated arguments to `mount.nfs` listed in line 11 of Listing A.3 were specified by the user and passed to the mount(8) command on the command shell as shown in Listing A.2.

Since the NFSv4+ client requests the NFS share from an NFSv4+ server across a network, it is necessary to open a socket and establish a network connection to the NFSv4+ server address referenced by the hostname first. The syscalls to `socket`, `bind` and `connect` are listed in the `strace` output in lines 14-18 of Listing A.3. Finally, in line 22, the original `mount` command from Listing A.2 is turned into its last userland permutation:
the `mount()` system call.

[1] See also strace manual pages: `$ man strace`

[2] see also mount(8) manual pages: `$ man 8 mount`

```
1  PID  syscall(args) = return value
2  ----------------------------------------------------------------------
3  9893 execve("/usr/bin/mount", ["mount", "-o", "vers=4.1", "dcache
4      -dot-door01.desy.de:/pnfs", "/pnfs"], 0x7ffc512eebe8) = 0 ...
5  9893 openat(AT_FDCWD, "/lib64/libmount.so.1", O_RDONLY|O_CLOEXEC)
6      = 3 ...
7  9893 openat(AT_FDCWD, "/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
   ...
8  9893 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|
9      CLONE_CHILD_SETTID| SIGCHLD, child_tidptr=0x7f469265cad0)
10     = 9894 ...
11 9894 execve("/sbin/mount.nfs", ["/sbin/mount.nfs", "dcache-dot-
12     door01.desy.de:/pnfs", "/pnfs", "-o", "rw,vers=4.1"],
13     0x7ffe40331638) = 0 ...
14 9894 socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP) = 3
15 9894 bind(3, {sa_family=AF_INET, sin_port=htons(0),
16     sin_addr=inet_addr("0.0.0.0")}, 16) = 0
17 9894 connect(3, {sa_family=AF_INET, sin_port=htons(0),
18     sin_addr=inet_addr("131.169.XYZ.ABC")}, 16) = 0
19 9894 getsockname(3, {sa_family=AF_INET, sin_port=htons(29161),
20     sin_addr=inet_addr("131.169.XYZ.ABC")}, [28 => 16]) = 0
21 9894 close(3) = 0
22 9894 mount("dcache-dot-door01.desy.de:/pnfs", "/pnfs", "nfs", 0,
23     "vers=4.1,addr=131.169.XYZ.256,cli"...) = 0
24 /* from here onward kernel routines are executed */
```

Listing A.3: Excerpt from the output of the `strace mount` call (user PID: 9893)

Similar to the syscall tracing command `strace` that was used in Listing A.2, the kernel offers a tracing infrastructure called *ftrace* [72]. This infrastructure is discussed in more detail in Chapter 3 in Section 3.1.5. In these examples it is used to visualize the call graph starting form the point where `strace` hands over control to kernel routines. In the transition from userland to kernelspace the syscall interface translates the `mount()` system call to the kernel routine `__x64_sys_mount()` shown in line 1 of Listing A.4. This first function within kernelspace is the starting point for numerous examples beginning with Section A.3.1.

A.3 Filesystem Context and Superblock Objects with NFSv4+

The question of interest in the NFSv4+ context is: What defines a superblock for a remote network filesystem type like NFSv4+ as opposed to a local, disk-based mount? How is the superblock accessed and constituted in NFSv4+, since it cannot be simply read and copied from a local disk partition into kernel memory?

With NFSv4+, the superblock's content is collected from two main sources of information. The mount options, which the user provides across the mount(2) syscall, form the first source of information about the forthcoming filesystem mount. These mount options, like the filesystem type or the flags which determine whether the mount should be a read-only mount or whether write access should also be allowed, are stored in an object called `struct fs_context`. The kernel documentation of the mount API states: «The creation and reconfiguration of a superblock is governed by a filesystem context» [92]. It is a common view to perceive this filesystem context as a preliminary superblock configuration data structure used by the VFS as long as no other information sources are available yet (see also documentation in the kernel sources e.g. in `/fs/nfs/fs_context`)

The second source of filesystem information in the case of the network filesystem is the NFSv4+ server. Being the owner of the filesystem it exports, the NFSv4+ server possesses all knowledge about its attributes and shares these with the NFSv4+ client at an early stage of a NFSv4+ session via the GETATTR[3][4] operation. Starting from network packet with number 775 in Figure A.1 (leftmost column) the client actively requests the server to transfer the filesystem metadata, specified via bitmap data structures defined in `/fs/nfs/nfs4proc.c`. As a reference to the filesystem in question, the client sends the filehandle (FH) of the root directory, obtained earlier in the commencing handshakes, with every GETATTR operation request. As a matter of fact, every GETATTR listed in the packet capture of Figure A.1, is a compound procedure consisting of a SEQUENCE, a PUTFH and a GETATTR operation all transmitted inside of a single RPC request (more in Section 2.8).

Kernel function calls involved in the gradual gathering of filesystem metadata from both, user and server source which are ultimately used to fill the NFSv4+ superblock appear in the function call graphs of listings A.4 and A.6 through A.8, initially triggered by the mount request issued from userland as shown in Listing A.2.

[3] Most presumably resolves to *get attributes* semantically.
[4] Capitalization of NFSv4+ operations follows common practice inside RFC 8881 [49]

A.3.1 Entering kernelspace

Resuming the description of the mount process from where it was left at the end of Section A.2, the first set of functions within the `__x64_sys_mount` syscall simply copy the options and parameters passed to the `mount(8)` invocation of Listing A.2 into kernel memory. Since the path to the desired local mountpoint is part of the parameter list, the kernel performs a path lookup (lines 5 through 12 of Listing A.4) in order to resolve the directory entries of the path components to their corresponding inodes. After the path lookup, the actual kernel-side mount process is initiated with the `path_mount()` routine defined in `/fs/namespace.c`. Therein, the mount options passed by the user are saved in two bitmaps called `mnt_flags` and `sb_flags`.

```
1  __x64_sys_mount() {
2      /* copy mount arguments from userland to kernelspace */
3      copy_mount_options() {...}
4        /* perform path lookup */
5        filename_lookup() {
6            path_lookupat() {
7                path_init() {...}
8                link_path_walk() {...}
9                walk_component() {...}
10               complete_walk() {...}
11               terminate_walk() {...}
12       }
13     /* set mount flags */
14     path_mount() {
15       do_new_mount() {
16         /* find filesystem type by name */
17         get_fs_type() {...}
18           fs_context_for_mount() {
19               /*
20               * function pointer call to
21               * 'init_fs_context' inside:
22               */
23               alloc_fs_context() {
24                 /* ... does actually invoke: */
25                 nfs_init_fs_context [nfs]() {...}
26                 ... /* tbc in next listing */
27           }
```

Listing A.4: Excerpt from a function graph traced with `ftrace` showing the begin of the in-kernel call sequence for the `__x64_sys_mount()` syscall (comments added for clarity)

The path_mount() routine returns calling the do_new_mount() function, passing the two bitmaps as arguments. The do_new_mount() function invokes routines for the initialization of a filesystem context. The control switch from the VFS to the NFS can be observed in the call graph beyond this point as described below.

After retrieving the filesystem type with the get_fs_type() routine, as described in Section 2.6.1, the routine fs_context_for_mount() defined in /fs/fs_context.c invokes the alloc_fs_context() routine in turn. As the routine's name suggests, a struct fs_context is allocated in kernel memory (line 6) and filled with all filesystem metadata available up to this point (line 8 of Listing A.5). Furthermore, a function pointer with the signature int (*init_fs_context)(struct fs_context *) is declared in line 3.

```
1  struct fs_context* alloc_fs_context(...) {
2      /* declare function pointer */
3      int (*init_fs_context)(struct fs_context *);
4      struct fs_context *fc;
5      /* allocate kernel memory */
6      fc = kzalloc(sizeof(struct fs_context), GFP_KERNEL);
7      /* save struct file_system_type to fs context*/
8      fc->fs_type = get_filesystem(fs_type);
9      ...
10     /*
11     * assign memory address of filesystem specific
12     * implementation to function pointer
13     */
14     init_fs_context = fc->fs_type->init_fs_context;
15     ...
16     /* here nfs_init_fs_context() is actually called */
17     ret = init_fs_context(fc);
18     ...
19 }
```

Listing A.5: Excerpt from the body of the alloc_fs_context() routine (comments and omissions added for clarity)

In the description of struct file_system_type in Section 2.6.1, Listing 2.13 it is seen that the NFSv4 module assigns the memory address of a NFS specific implementation of the init_fs_context() routine to a function pointer. This very function pointer is now assigned to the function pointer in the alloc_fs_context() routine as listed in line 14 of Listing A.5. As a consequence, not the init_fs_context() routine, but the NFS

specific routine called `nfs_init_fs_context()` is actually invoked in line 17 of the same listing.

The commentary for the `nfs_init_fs_context()` definition in `/fs/nfs/fs_context.c` reads: «Prepare superblock configuration. [...]». Thus, the routine allocates a NFS specific `struct nfs_fs_context` and a `struct nfs_fh` (a data structure containing a NFS filehandle) in kernel memory. The `struct nfs_fs_context` is subsequently initialized with default filesystem metadata which indicate that information about the forthcoming filesystem is still unspecified.

After returning from the `fs_context_for_mount()` routine in line 18 of Listing A.4 the next milestone for the VFS in the mount process is the attempt to obtain more metadata about the NFS share in question and use those metadata to create a client-side superblock representation in memory.

```
28  vfs_get_tree() {
29    nfs_get_tree [nfs]() {
30      nfs4_try_get_tree [nfsv4]() {
31        do_nfs4_mount [nfsv4] {
32        /* no tree without server, so first: */
33        nfs4_create_server [nfsv4]() {
34          nfs_alloc_server [nfs]() {...}
35          nfs4_init_server [nfsv4]() {
36            /* secondly: */
37            nfs4_set_client [nfsv4]() {
38              nfs_get_client [nfs]() {
39                nfs4_alloc_client [nfsv4]() {...}
40                  /* thirdly, one layer down: */
41                  nfs_create_rpc_client [nfs]() {...}
42                    /*
43                    * perform procedure NULL op: 0
44                    * across the network,
45                    * to test connectivity between NFSv4 client
46                    * and server
47                    */
48                    rpc_ping [sunrpc]() {
49                      rpc_call_null_helper [sunrpc]() {...}
50                    }
51                  /* if that was successful: */
52                  nfs41_init_client [nfsv4]() {
53                    ... /* tbc in next listing */
```

Listing A.6: Sequel #1 of the in-kernel call sequence for the `__x64_sys_mount()` syscall (comments added for clarity)

This is accomplished by invoking the `vfs_get_tree()` routine (top line of Listing A.6) defined in `/fs/super.c`. By means of the function pointer mechanism seen earlier, this routine delegates a task from the VFS to the NFS layer by invoking the `get_tree()` function prototype alias `nfs_get_tree()`. The latter routine is a decision point. If a field of type boolean, inside `struct nfs_fs_context` called `internal` is set to *false*, the function `try_get_tree()` alias `nfs4_try_get_tree()` is called. Or else, `nfs_get_tree_common()` is called. Since at this point the `internal` field is still untouched, it evaluates to *false*.

The `nfs4_try_get_tree()` routine is a wrapper for the `do_nfs4_mount()` routine, both defined in `/fs/nfs/nfs4super.c`. It expects a `struct nfs_server` as first argument. But since no `struct nfs_server` has been defined in the course of the call graph yet, a routine called `nfs4_create_server()` is passed as a parameter. This routine invokes a sheer cascade of functions that help setting up the NFSv4+ client and server structs in kernel memory (lines 33 through 39 of Listing A.6), create an RPC client (line 41) and start network communication across the TCP/IP stack of the kernel (lines 48 through 50). All of this aims at obtaining more filesystem metadata for initialization of the NFS server structure and the creation of a VFS superblock, still pending at this point.

Lines 60 through 70 of Listing A.7 show the consecutive invocation of the NFSv4 routines called `nfs4_server_capabilities()`, `nfs4_do_fsinfo ()` and `nfs4_proc_pathconf()`. Each of them requests filesystem attributes from the remote NFSv4 server. (The term *attributes* is interchangeably used with the term *metadata*).

```
54    nfs4_alloc_session [nfsv4]() {...}
55      /* EXCHANGE_ID op: 42 */
56      nfs4_proc_exchange_id [nfsv4]() {
57      /* PUTROOTFH op: 24 */
58      nfs4_lookup_root [nfsv4]() {...}
59    /* PUTFH & GETATTR ops: 22 & 9 */
60    nfs4_server_capabilities [nfsv4]() {...}
61    /*
62    * PUTFH & GETATTR ops: 22 & 9
63    * with a 'nfs4_fsinfo_bitmap'
64    */
65    nfs4_do_fsinfo [nfsv4]() {...}
66    /*
67    * PUTFH & GETATTR ops: 22 & 9
68    * with a 'nfs4_pathconf_bitmap'
69    */
70    nfs4_proc_pathconf [nfsv4]() {...}
71    ... /* tbc in next listing */
```

Listing A.7: Sequel #2 of the in-kernel call sequence for the `__x64_sys_mount()` syscall showing NFS specific function invocations (comments added for clarity)

It is worth mentioning two aspects regarding the creation of `struct nfs_client` and `struct nfs_server`. Interestingly, both are declared in `/include/linux/nfs_fs_sb.h`, while the ...`fs_sb`... part of the header file's name, most probably expands to **f**ilesystem super**b**lock.

The first aspect about the NFSv4+ client creation is, that inside the `nfs4_set_client()` routine in line 37 of Listing A.6, all NFSv4+ client-related information are assigned to the fields of a structure called `struct nfs_client_initdata`. Three data structures filled with NFSv4 specific functions are saved into a structure called `struct nfs_subversion` defined in `/fs/nfs/nfs4super.c`. One of these data structures is of type `struct file_system_type` called `nfs4_fs_type` described in Section 2.6.1. A second data structure is of type `struct super_operations` declared in `include/linux/fs.h`, which contains superblock-related routines. The third data structure is called `struct nfs_rpc_ops` and contains 47 single function pointer assignments and 4 further collections of function pointers. Two of the collections are of type `struct inode_operations`, one of type `struct dentry_operations` and the last one, not surprisingly, of type `struct`

file_operations. This shows that in the definition of struct nfs_subversion all filesystem related routines for all common file model objects are stored together and form the interface to the VFS. In conclusion, inside the function body of nfs4_set_client() the memory address of this whole palette of routines (that also includes other client related metrics such as the hostname, a corresponding IP address and so forth) is assigned to a field called nfs_mod inside struct nfs_client_initdata . These metrics get passed to the client in the course of its creation, namely in the body of the nfs4_alloc_client() routine (line 39 of Listing A.6) to a field called cl_nfs_mod of type struct nfs_client. From this point onward, the NFSv4+ client is fully capable of servicing every request made by the VFS by invoking proper NFSv4+ specific implementations.

Interestingly, the struct nfs_client maintains a list to known NFS servers, filled with struct nfs_server entries, which is called *cl_superblocks*. This identifier gives a hint to what the VFS superblock representation for the NFSv4+ layer might be, and leads to the second aspect mentioned earlier. This second aspect regarding the NFS server reveals itself by examining the contents of the struct nfs_server. Apart from entries dealing with network related metrics, like the server's hostname, IP address, port or protocol specific timeout values and authentication related information - all things the VFS is not even remotely interested in - the structure contains very filesystem specific data. These comprise the minimum and maximum attribute cache timeouts for regular files as well as for directories, read and write sizes, allowed name length, a numerical filesystem identifier (with major and minor number) and a set of bitmask arrays of type unsigned 32-bit integer, as a space saving means to transfer attributes supported by the exported filesystem to the NFS client.

The struct nfs_server also includes its VFS equivalent in form of a struct super_block, with which it shares filesystem attributes like the maximum file size allowed and, more importantly, the block size that is used on the server's storage devices. This all shows that with respect to the NFS filesystem type, the struct nfs_server is the actual in-kernel representation of the filesystem metadata. Still sharing VFS related metadata with the struct super_block, whose creation process will be briefly and conclusively described in the following.

Returning to the point in the call graph of the mount process, where the nfs4_create_server() routine (line 33 of Listing A.6) returns to its caller after setup and activation of necessary components for the network communication as well as successful retrieval of filesystem metadata from the remote NFSv4+ server.

The first relevant actions occuring inside the body of the `do_nfs4_mount()` function (line 72 of Listing A.8) is the assignment of a boolean value equal to *true* to the above mentioned `internal` field of type `struct nfs_fs_context`.

Secondly, the retrieved filesystem metadata are parsed and saved in the superblock configuration context, namely the same `struct nfs_fs_context`.

And thirdly, a routine called `fc_mount` is invoked, which immediately starts a second attempt at creating the superblock by calling `vfs_get_tree()` once more. Reaching the decision point within the `nfs_get_tree()` routine, with `internal` set to *true*, this time the second branch is followed. Since it is not longer necessary to *try* to get the tree, now, with all filesystem metadata at hand, it is possible to simply *get* it. This is accomplished with a call to `nfs_get_tree_common()` routine defined in `/fs/nfs/super.c`. Inside which, the function `sget_fc()` gets called with the `nfs_set_super()` callback function set in its parameter list.

After allocating and initializing a blank superblock, the first metadata from the filesystem context, like the filesystem type, flags, name and the like, are assigned. The `nfs_set_super()` callback function (line 85 of Listing A.8 adds the dentry operations structure from the NFSv4+ client to the newborn superblock. A final assignment of filesystem attributes occurs after returning from the `nfs_set_super()` callback to the `nfs_get_tree_common()` routine, which invokes a routine named `nfs_fill_super()` from within its function body. Here, the superblock receives its `struct super_operations` from the NFSv4+ client, which maintains all VFS function pointer assignments of NFSv4 specific implementations, as described earlier. But also filesystem metadata like the block size, time granularity, maximum file size, etc., are assigned to the `struct super_block`, finalizing its creation in kernel memory. To demonstrate this has been the actual purpose of this deep dive into the NFSv4+ mount process up to this point.

```
72  do_nfs4_mount [nfsv4]() {
73    vfs_parse_fs_param() {...}
74      fc_mount() {
75      /*
76      * Now, with more information about
77      * the filesystem exported by the server,
78      * try to setup a superblock in client memory.
79      */
80        vfs_get_tree() {
81          nfs_get_tree [nfs]() {
82            nfs_get_tree_common [nfs]() {
83              sget_fc() {...}
84                alloc_super() {...}
85                nfs_set_super [nfs]() {...}
86              /* finally constitute superblock */
87              nfs_fill_super [nfs]() {...}
88              nfs_get_root [nfs]() {
89                /* PUTFH & GETATTR ops: 22 & 9 */
90                nfs4_proc_get_root [nfsv4]() {...}
91                  nfs4_proc_getattr [nfsv4]() {...}
92                nfs_fhget [nfs]() {
93                  iget5_locked() {
94                      ilookup5() {...}
95                      alloc_inode() {...}
96                      inode_insert5() {...}
97                  }
98                  ...
99              d_make_root() {
100                 __d_alloc() {...}
101                 d_set_d_op();
102               __d_instantiate() {
103                 __d_set_inode_and_type() {...}
104               }
105             ...
106             }
107 } /* end __x64_sys_mount() */
```

Listing A.8: Sequel #3 of the in-kernel call sequence for the `__x64_sys_mount()` syscall (comments added for clarity)

A.4 Inode Objects and NFSv4+

In his book about Linux kernel development, Robert Love states that «[...] the inode object is constructed in memory in whatever manner is applicable to the filesystem» [28]. The rest of this section will attempt to shed some light on how this is brought about in the case of NFSv4+ by scrutinizing the corresponding Linux kernel sources.

As a sequel to previous descriptions, the NFSv4+ mount process will be the basis of the ongoing investigation once again. At the end of Section A.3.1 the description of the mount process' call graph was left at the point where a superblock object representing the NFS share was assigned filesystem attributes provided by the remote NFS server.

After having completed the collection of general filesystem information, a NFSv4+ request is initiated, which aims at retrieving metadata about the root directory of the forthcoming filesystem tree shared across the network. Line 88 of Listing A.8 shows the invocation of the `nfs_get_root()` routine, which is defined in `/fs/nfs/getroot.c`. Before the request is prepared and subsequently sent to the remote NFS4+ server, a NFS4+ specific data structure called `struct nfs_fattr` (*fattr*, shorthand for *file attributes*), which is declared in `/include/linux/nfs_xdr.h`, has to be allocated. Examining the contents of this essential data container, the impression arises that it could be considered a *traveling* inode, since it covers most fields also present in the `struct inode` plus some NFS+ specific metrics, like the filehandle mentioned above and a file id. It gets filled with file attributes, or changes thereof, by the NFSv4+ client and the server alternately, and continuously transported back and forth between them with every `GETATTR` or `SETATTR` request.

Thus, one of the first actions performed in the body of the `nfs_get_root()` routine is the allocation of a `struct nfs_fattr`. Wrapped inside another data structure it is passed to a function pointer call as an argument together with a `struct nfs_fh` and a `struct nfs_server` instance. The function pointer invokes the `nfs4_proc_get_root()` routine, which in turn calls `nfs4_proc_getattr()` (line 91 of Listing A.8).

After returning from the round trip across the network with all above mentioned data structures filled with the required metadata about the root directory of the NFS share, the instantiation of the new filesystem's first inode structure is started by the invocation of the `iget5_locked()` routine. This routine takes a `struct super_block`, the `fileid`, a callback function named `nfs_init_locked()` and a data structure that comprises both, the freshly obtained file handle and the file attributes in their respective structs as arguments.

The fileid argument serves as a component used by a hash() function defined in /fs/inode.c. Therein, it is used in concert with the memory address of the corresponding superblock to calculate a unique hash value for identifying and placing a newborn inode inside a data structure called inode_hashtable alias the inode cache.

But before allocating a brand new inode, an inode cache lookup is performed in order to ensure that an inode with the passed fileid (alias inode number) does not already exist in memory. This is checked by calling the ilookup5() routine immediately after entering iget5_locked().

After acquiring a lock for the inode cache the find_inode() function is invoked, inside which the inode cache is iteratively browsed for the given hash value and superblock. Since this is the middle of a mount process and the attributes for the root directory have been delivered to the NFS client only recently, no proper inode can be returned by the find_inode() routine. The unsuccessful inode inquiry is propagated back up the call graph as NULL value.

Having reached the iget5_locked() routine, the NULL value triggers a call to a VFS alloc_inode() function defined in /fs/inode.c. Using the now well-known function pointer assignment mechanism, the NFS specific routine named nfs_alloc_inode() is invoked therein. As the name suggests, this routine causes an allocation of a struct nfs_inode within the inode cache area reserved for the NFSv4 in kernel memory. Additionally, it returns the memory address to the above mentioned vfs_inode field of the struct nfs_inode, since a return value of type pointer to struct inode is expected from the call to the nfs_alloc_inode() function pointer.

The initialization of the newly allocated inode takes place inside the inode_init_always() call, where all fields of the blank struct inode are initialized to default values, except one. Only the field of type struct super_block is assigned the appropriate address of the superblock the new inode belongs to.

Returning from the alloc_inode() routine inside the body of iget5_locked(), a last sanity check has to be performed. The point here is, that while the control flow described above left the allocating routine and entered the initialization process inside inode_init_always(), another process could have managed to allocate, initialize and insert the same inode in the inode cache. To avoid this error-prone situation of having two inodes with the same hash in the inode cache, a second iterative search through the inode cache is indispensable. For this purpose the inode_insert5() routine is called, which rehashes

the inode's `fileid` value passed in earlier and calls the already known `find_inode()` routine again. Amusingly enough, in case the mentioned scenario really occurs and an inode with the same hash value was inserted into the inode cache by another process, the kernel developer's commentary in the source code of the `inode_insert5()` in `/fs/inode.c` reads as follows:

> «*Uhhuh, somebody else created the same inode under us.*
> *Use the old inode instead of the preallocated one.*»

Fortunately, this did not occur in this case study. Thus, the new inode can be added firstly to the inode cache and secondly to a list of inodes, which the superblock manages in its function as administrator for one specific filesystem within the kernel's VFS layer.

The new inode for the root directory of the exported NFS share is finally successfully returned from the `iget5_locked()` routine into the body of the `nfs_fhget()` function again.

This walk-through has hopefully made it clear that the VFS involves its own routines to a great extent in the process of the creation of a single inode while falling back onto filesystem specific implementations only where needed. Or, as the authors of [27] put it more concisely:

> «In some sense, the VFS could be considered a "generic" filesystem,
> that relies, when necessary, on specific ones.»

A.5 Dentries in the NFS Mount Process

Starting from line 99, Listing A.8 shows a few example function calls that are invoked whenever the root dentry is created and associated with its inode in the mount process. The invocation of the `d_make_root()` routine (this and all following are defined in `/fs/dcache.c`) which takes the recently acquired inode as an argument, essentially calls the `__d_alloc()` function that in turn takes the superblock from the inode as an argument. As could be deduced from the name, the `__d_alloc()` routine allocates a new `struct dentry` in the dcache. Additionally, since it is the root directory, a forward slash is assigned to its `struct qstr` name field. Additionally, a reference to the superblock parameter is added. Inside the `__d_alloc()` routine the newborn dentry receives the `struct dentry_operations`, again taken from the superblock parameter in a subsequent call to `d_set_d_op()`. If a valid dentry is finally returned from `__d_alloc()`, it is time for its instantiation in a routine called `d_instantiate`. The kernel source documentation for this routine happily states:

«d_instantiate - fill in inode information for a dentry [...]
This turns negative dentries into productive full members of society.»

And that is exactly what happens in a subsequent call to `__d_set_inode_and_type()`, which finally assigns the passed inode to the `d_inode` field of the new root dentry structure.

A.6 XDR Encoding of a NFSv4+ LOOKUP Operation

Some of the topics touched upon in Section 2.8 have been considered separately from the circumstances in the Linux kernel. In order to gain a better insight into the interactions within the kernel, the relevant part of the function call graph shown in Listing A.10 will form the basis of illustration and be discussed in more detail with a special focus on the occurrence of the XDR encoding and the NFS-RPC layer interplay.
Following the relevant kernel source files, it will be demonstrated how the NFSv4 layer handles a request coming in from the VFS layer and prepares it for transmission across the network in conjunction with the RPC layer. Let an arbitrary VFS operation like the lookup of an inode object serve as an example. In order to make the comprehension of the call graph easier to pursue, it shall be defined that all occurrences of the shorthand fragment `proc`, appearing in the following kernel source code identifiers, actually represent the term *procedure*[5]. As described in subsection 2.8.1.4, a procedure is a compound that encompasses a set of single *operations* (see also nomenclature used in RFC 8881 [49]). These NFSv4.1 operations can be filesystem specific tasks enforced by the VFS, like LOOKUP, OPEN, READ, WRITE, CLOSE etc. as well as protocol specific operations such as EXCHANGE_ID, CREATE_SESSION, GETDEVICEINFO and LAYOUTGET for instance.
When a `struct nfs_client` is first allocated in kernel memory during the execution of the `nfs_alloc_client()` routine (see Listing A.4), the memory address of a variable called `nfs_v4_clientops` of type `struct nfs_rpc_ops` is assigned to the client structure field called `rpc_ops`. After this assignment, the `rpc_ops` field points to a collection of available NFSv4 client related routines. Some of which are claimed by the VFS and therefore assigned to VFS function pointers by a known mechanism already described in Section 2.6.1. For example, the VFS function pointer called `(*lookup)` found in `struct inode_operations` inside of the `struct inode` is assigned the memory address of

[5] The interpretation of `proc` as *procedure* has to be defined here, since it can not be proven out of the kernel sources itself, but is most probable, within the given context.

a NFSv4 routine called `nfs4_proc_lookup()`. The latter routine calls `nfs4_proc_-lookup_common()`, which in turn calls `_nfs4_proc_lookup()`. Inside this function, a `struct rpc_message` is filled with information about which XDR related functions are to be used to encode the lookup operation prior to its transmission via TCP. Here is how this information is gradually gathered:
At the uppermost level it is a simple assignment of the pointer to an array entry, to a field called `rpc_proc` inside of `struct rpc_message`. The array is called `nfs4_procedures[]` and contains macros with function prototypes needed to XDR encode and decode NFSv4+ operations. It is of type `struct rpc_procinfo` and defined in `/fs/nfs/nfs4xdr.c`. The entry found at index `NFSPROC4_CLNT_LOOKUP` of the array turns out to be a macro called `PROC(LOOKUP, enc_lookup, dec_lookup)`. The purpose of this macro is first to append the prefix `nfs4_xdr_` to `enc_lookup`, resulting in the function name `nfs4_-xdr_enc_lookup`, and secondly to assign this function to a field called `p_encode` of a `struct rpc_procinfo`, which lives inside the aforementioned `struct rpc_message` and that again, as stated above, is filled inside the body of the `_nfs4_proc_lookup()` routine. Summing things up, the assignment can be represented by the following sequence of pointer dereferences:

```
1  struct rpc_message msg;
2  msg.rpc_proc->p_encode = nfs4_xdr_enc_lookup;
3  msg.rpc_proc->p_decode = nfs4_xdr_dec_lookup;
```

Listing A.9: Assigning the encode/decode routines for the NFSv4+ lookup operation

For simplicity, the focus will be put on the encoding part only from now onward. Taking a closer look into the `nfs4_xdr_enc_lookup()` encoding routine assigned in Listing A.9, it appears as a wrapper for five further routines, namely

- `encode_sequence()`
- `encode_putfh()`
- `encode_lookup()`
- `encode_getfh()`
- `encode_getfattr()`.

Before transmission, down in the RPC layer of the function call graph, the `nfs4_xdr_enc_-lookup()` routine will induce the XDR encoding of the five NFSv4+ operations specified in the suffixes of the five function names listed above. These will be transmitted collectively, as a compound procedure, within one single RPC request. A look inside `encode_lookup()`

reveals even more encoding routines. The first of which is called `encode_op_hdr()` and encodes the NFSv4+ *operation number* associated with the lookup operation represented by the enumeration type `enum nfs_opnum4` defined in `/include/linux/nfs4.h`. It is called `OP_LOOKUP` and has a numerical value equal to 15. Needless to mention, that the operation number `OP_LOOKUP` = 15, which is to be transmitted by the Linux NFSv4+ client inside its RPC message, must be understood by any NFSv4+ server implementation as a request for a lookup operation, according to the specification found in RFC 8881 [49]. This NFSv4.1 server could also be a Java based metadata server on the dCache storage end of a HTC environment, like the NAF at DESY, for instance.
Naturally, at this point, the question arises: What happens *after* the preparation of the NFSv4+ lookup procedure within the `_nfs4_proc_lookup()` routine as described above ? Unfortunately, the enormously high number of routines called on the RPC and underlying layers of the network stack prohibits a more detailed discussion, since it leads far beyond the scope of this thesis. Therefore, only the main cornerstones of the function call graph starting from the `_nfs4_proc_lookup()` routine, down to the point where the actual XDR encoding of the NFSv4+ operations is triggered, is displayed in Listing A.10.

```
1       ...
2       _nfs4_proc_lookup [nfsv4]()
3         nfs4_do_call_sync [nfsv4]()
4           rpc_run_task [sunrpc]()
5             /* rpc_message is assigned to rpc_task here */
6             rpc_task_set_rpc_message [sunrpc]()
7           rpc_execute [sunrpc]()
8             nfs4_setup_sequence [nfsv4]()
9             nfs4_find_or_create_slot [nfsv4]()
10          call_start [sunrpc]()
11          call_allocate [sunrpc]()
12          call_encode [sunrpc]()
13            rpc_xdr_encode [sunrpc]()
14              rpcauth_wrap_req_encode [sunrpc]()
15                /* encoding of NFSv4 ops triggered here */
16                nfs4_xdr_enc_lookup [nfsv4]()
17                  encode_compound_hdr [nfsv4]()
18                  encode_sequence [nfsv4]()
19                  encode_putfh [nfsv4]()
20                  encode_getattr [nfsv4]()
21                  ...
```

Listing A.10: Excerpt from the mount process function call graph in the NFS and RPC layer (curly brackets and most ommission dots omitted comments added for clarity)

In the context of eBPF development it is indispensable to know the exact locations in the function call graph where XDR encoding and decoding take place. This knowledge allows for a more targeted approach when choosing the adequate kernel functions for obtaining relevant NFSv4+ metrics in their unencoded form.

No.	Proto	Length	tcp Len	Info
131	NFS	110	44	V4 NULL Call (Reply In 136)
136	NFS	94	28	V4 NULL Reply (Call In 131)
757	NFS	362	296	V4 Call (Reply In 759) EXCHANGE_ID
759	NFS	222	156	V4 Reply (Call In 757) EXCHANGE_ID
761	NFS	362	296	V4 Call (Reply In 762) EXCHANGE_ID
762	NFS	222	156	V4 Reply (Call In 761) EXCHANGE_ID
763	NFS	306	240	V4 Call (Reply In 766) CREATE_SESSION
766	NFS	194	128	V4 Reply (Call In 763) CREATE_SESSION
767	NFS	214	148	V4 Call (Reply In 768) RECLAIM_COMPLETE
768	NFS	158	92	V4 Reply (Call In 767) RECLAIM_COMPLETE
769	NFS	222	156	V4 Call (Reply In 771) SECINFO_NO_NAME
771	NFS	174	108	V4 Reply (Call In 769) SECINFO_NO_NAME
772	NFS	234	168	V4 Call (Reply In 774) PUTROOTFH | GETATTR
774	NFS	350	284	V4 Reply (Call In 772) PUTROOTFH | GETATTR
775	NFS	266	200	V4 Call (Reply In 778) GETATTR FH: 0xc3ae9285
778	NFS	238	172	V4 Reply (Call In 775) GETATTR
779	NFS	262	196	V4 Call (Reply In 781) GETATTR FH: 0xc3ae9285
781	NFS	234	168	V4 Reply (Call In 779) GETATTR
782	NFS	266	200	V4 Call (Reply In 784) GETATTR FH: 0xc3ae9285
784	NFS	238	172	V4 Reply (Call In 782) GETATTR
785	NFS	262	196	V4 Call (Reply In 786) GETATTR FH: 0xc3ae9285
786	NFS	234	168	V4 Reply (Call In 785) GETATTR
789	NFS	258	192	V4 Call (Reply In 790) GETATTR FH: 0xc3ae9285
790	NFS	186	120	V4 Reply (Call In 789) GETATTR
792	NFS	266	200	V4 Call (Reply In 793) GETATTR FH: 0xc3ae9285
793	NFS	238	172	V4 Reply (Call In 792) GETATTR
794	NFS	262	196	V4 Call (Reply In 795) GETATTR FH: 0xc3ae9285
795	NFS	310	244	V4 Reply (Call In 794) GETATTR
796	NFS	270	204	V4 Call (Reply In 797) ACCESS FH: 0xc3ae9285, [Check: RD LU MD XT DL]
797	NFS	238	172	V4 Reply (Call In 796) ACCESS, [Access Denied: MD XT DL], [Allowed: RD LU]
798	NFS	278	212	V4 Call (Reply In 801) LOOKUP DH: 0xc3ae9285/pnfs
801	NFS	358	292	V4 Reply (Call In 798) LOOKUP

Figure A.1: NFS network packets exchanged between NFS client and server during the mount process started with the command in Listing A.2 (captured by `tshark`, displayed by Wireshark)

A.7 Wireshark NFSv4.1 payload dissection

```
Network File System, Ops(4): SEQUENCE, PUTROOTFH, GETFH, GETATTR
    [Program Version: 4]
    [V4 Procedure: COMPOUND (1)]
    Tag: <EMPTY>
        length: 0
        contents: <EMPTY>
    minorversion: 1
    Operations (count: 4): SEQUENCE, PUTROOTFH, GETFH, GETATTR
        Opcode: SEQUENCE (53)
            sessionid: 668aea5b0001006a0000000000000001
            seqid: 0x00000003
            slot id: 0
            high slot id: 0
            cache this?: No
        Opcode: PUTROOTFH (24)
        Opcode: GETFH (10)
        Opcode: GETATTR (9)
            Attr mask[0]: 0x0010011a (Type, Change, Size, FSID,
   FileId)
                <[5 Attr counts]>
                reqd_attr: Type (1)
                reqd_attr: Change (3)
                reqd_attr: Size (4)
                reqd_attr: FSID (8)
                reco_attr: FileId (20)
            Attr mask[1]: 0x00b0a23a (Mode, NumLinks, Owner,
   Owner_Group, RawDev, Space_Used, Time_Access, Time_Metadata,
   Time_Modify, Mounted_on_FileId)
                <[10 Attr counts]>
                reco_attr: Mode (33)
                reco_attr: NumLinks (35)
                reco_attr: Owner (36)
                reco_attr: Owner_Group (37)
                reco_attr: RawDev (41)
                reco_attr: Space_Used (45)
                reco_attr: Time_Access (47)
                reco_attr: Time_Metadata (52)
                reco_attr: Time_Modify (53)
                reco_attr: Mounted_on_FileId (55)
    [Main Opcode: PUTROOTFH (24)]
    [Main Opcode: GETATTR (9)]
```

Listing A.11: Linux NFSv4.1 client PUTROOTFH request dissected by `wireshark`

```
Network File System, Ops(4): SEQUENCE PUTROOTFH GETFH GETATTR
    [Program Version: 4]
    [V4 Procedure: COMPOUND (1)]
    Status: NFS4_OK (0)
    <Status: OK (0)>
    Tag: <EMPTY>
        length: 0
        contents: <EMPTY>
    Operations (count: 4)
        Opcode: SEQUENCE (53)
            Status: NFS4_OK (0)
            <Status: OK (0)>
            sessionid: 668aea5b0001006a0000000000000001
            seqid: 0x00000003
            slot id: 0
            high slot id: 15
            target high slot id: 15
            status flags: 0x00000000
        Opcode: PUTROOTFH (24)
            Status: NFS4_OK (0)
            <Status: OK (0)>
        Opcode: GETFH (10)
            Status: NFS4_OK (0)
            <Status: OK (0)>
            Filehandle
                length: 27
                [hash (CRC-32): 0xc3ae9285]
                version: 1
                generation: 0x00000000
                export_id: 0x00000000
                fh type: INODE (0)
                fh opaque data: 0000000000000001
        Opcode: GETATTR (9)
            Status: NFS4_OK (0)
            <Status: OK (0)>
            Attr mask[0]: 0x0010011a (Type, Change, Size, FSID,
   FileId)
                <[5 Attr counts]>
                reqd_attr: Type (1)
                    ftype4: NF4DIR (2)
                reqd_attr: Change (3)
                    changeid: 33
                reqd_attr: Size (4)
                    size: 512
                reqd_attr: FSID (8)
                    fattr4_fsid
                        fsid4.major: 17
                        fsid4.minor: 17
```

```
                reco_attr: FileId (20)
                    fileid: 1
            Attr mask[1]: 0x00b0a23a (Mode, NumLinks, Owner,
    Owner_Group, RawDev, Space_Used, Time_Access, Time_Metadata,
    Time_Modify, Mounted_on_FileId)
                <[10 Attr counts]>
                reco_attr: Mode (33)
                    ...
                reco_attr: NumLinks (35)
                    numlinks: 17
                reco_attr: Owner (36)
                    fattr4_owner: 0
                        length: 1
                        contents: 0
                        fill bytes: opaque data
                reco_attr: Owner_Group (37)
                    fattr4_owner_group: root@desy.afs
                        length: 13
                        contents: root@desy.afs
                        fill bytes: opaque data
                reco_attr: RawDev (41)
                    specdata1: 0
                    specdata2: 0
                reco_attr: Space_Used (45)
                    space_used: 512
                reco_attr: Time_Access (47)
                    seconds: 1502883811
                    nseconds: 914000000
                reco_attr: Time_Metadata (52)
                    seconds: 1720288779
                    nseconds: 5000000
                reco_attr: Time_Modify (53)
                    seconds: 1720288779
                    nseconds: 5000000
                reco_attr: Mounted_on_FileId (55)
                    fileid: 0x0000000000000001
    [Main Opcode: PUTROOTFH (24)]
    [Main Opcode: GETATTR (9)]
```

Listing A.12: NFSv4+ server reply (on the dCache end) dissected by `wireshark`

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original