NFSv2 Concurrency Explorer

Junjiang Li[1] and Rowshni Tasneem Usha[1]

University of Toronto, Toronto, Canada
{junjiangli, rowshni.usha}@cs.toronto.edu

Abstract. In this paper we explain the implementation of a simulator that explores concurrency in the network file system. The proposed simulation features a variable number of clients carrying out the user's instructions. These instructions will be a subset of existing network-based file system protocols. Depending on the status of attribute cache and the way client code is written, concurrent operations can produce a wide range of unintuitive behavior. With our simulation, we wish to strengthen the user's understanding of distributed file systems. The simulator will investigate all possible event sequences as seen by the server and generate visuals for each scenario while drawing a unique conclusion.

1 Introduction

The basis of this research is focused on the Network File System (NFS), which is a distributed file system that permits users to browse files and directories on remote machines. NFS is a reasonably inexpensive and simple-to-use network file sharing solution that uses current internet protocol infrastructure.

The implementation of the NFS client raises some issues. Consider the scenario where two NFS clients access the same file, with one client writing to the file and the other reading to the file. Given that both requests begin at the same time, and the writing finishes before the server can respond to the reading client, the read requests will return an outdated version of the file. The purpose of this research is to build a simulator to illustrate the concurrent operations of a network file system. The simulation can encapsulate all allowable permutations of writes that occur concurrently in a network file system.

Depending on the order by which the server receives write requests from clients, the operations of a network file system can lead to unexpected results. Moreover, the state of the cache, and the consistency model of the file system, might affect a network file system's operations towards unanticipated outcomes. With the help of a simulation displaying concurrent operations, we can better explore the behavior of NFS. The simulator will examine all conceivable event sequences as viewed by the server and provide exploration for each scenario with a distinct conclusion. The analysis necessitates the use of simulation as it enables viewers to compare and draw conclusions from the insights gained, which otherwise could be overlooked.

In this paper, we replicate the basics of NFS structure and implement a simulator to explore concurrency in the file system. In the first two sections we discussed our motivation and literature studies. Section 3 describes the step by step implementation process of the simulation. The rest of the paper discusses our findings, results from the simulation, some limitations and future works in sections 4 and 5.

2 Related Works

A significant amount of work has been done on the network file system. Sun NFS was among the first and is still in use. The target of a Network File System is to share files among nodes on the same LAN. One of the early applications of distributed client/server computing was in distributed file systems [1][2]. In such an environment, there are several client machines and one or a few server machines; the server stores data on its disks and clients request data using well-formed protocol messages.

An NFS [3][4] server is composed of a node that exports its local file system to clients that access it via a remote mounting action. NFS is a stateless protocol. No state is maintained on the server-side, therefore each action is self-contained. The file system requirements for an NFS appliance differ from those for a general-purpose UNIX system because an NFS appliance must be optimized for network file access as well as user-friendly [5].

A study discusses the NFS protocols which are defined by a collection of procedures [6], their arguments and outcomes, and their consequences. Because of its centralized server design, NFS has a potential scalability barrier as clusters become larger. As a response, various improvements are currently underway in both the areas of improving NFS and delivering new solutions [7]. Another prior research discusses the basic Network file system (NFS) and presents a comparative study of different variations of it [8]. Relying on the standard NFS protocols, cooperative client-side caching has been extensively explored in another study [9]. According to this research, a common file access pattern found in cluster applications is concurrent read sharing: applications running on multiple sites read to access the same dataset concurrently.

According to a study on concurrency [10], the implementation of the NFS client raises the following issue: if two NFS clients access the same file, with one client writing to the file and the other reading to the file, and both requests begin at the same time, but the writing client finishes writing to the file before the server can respond to the reading client, the read requests will return an outdated version of the file. As a result, NFS cannot ensure write-to-read consistency.

The NFS client, on the other hand, guarantees close to open consistency. When a file is closed, all outstanding writes are completed before the file is closed completely. The process of closing a file is quite slow. As a result, when client 2 accesses the identical file, it will have the necessary modifications [10]. If a server cannot keep up with the incoming NFS write requests, it may be required to reduce the quantity of NFS requests. This is accomplished by reducing the number of NFS async threads; the kernel RPC method continues to function without the async threads, but less effectively [11].

Several studies guaranteeing stronger consistencies so that a distributed system behaves more like a single-user system have been extensively explored. For example, when Amazon's S3 still followed the eventual consistency model until late 2020 [12], many tools existed to implement a strong consistency layer on top of S3 [13]. While the semantics of the NFS protocol is stronger than eventually consistent, the use of cache recreates cache consistency problems [14]. There are tools such as [15] that simulate some aspects of a distributed file system, however, there is not a systematic exploration of all the possible scenarios given a specified user code.

3 Design and Implementation

In this section, we discuss the important design choices and implementation details of our concurrency explorer. We will first look at how is the NFSv2 protocol implemented in our simulation, and then discuss the organization of the simulator itself, before finally moving on to an optimization that prunes the search space.

3.1 Implementing the NFSv2 Protocol

The NFSv2 protocol defines a set of procedures to manipulate the files and directories on a remote file server. An NFS client, which is typically the file system layer of a user machine's operating system, translates the usual file system operations such as `open` and `write` issued by the end user into sequences of procedures defined in the NFSv2 protocol, and then executes them remotely using the remote procedure call protocol. This interaction is summarized in figure 1.

For conceptual simplicity, we chose to replicate this structure in our simulation. A `Server` object in our simulation holds a pointer to the root of an in-memory tree structure representing the server's local file system. Additionally, it implements a subset of the procedures in the NFSv2 protocol as instance methods that modifies the directory structure it owns. Because the NFSv2 protocol is extensive, we only implemented basic operations to demonstrate the effect of concurrency in an NFS system. We also implemented the non-idempotent procedures to further explore the interaction between non-idempotency and concurrency. See table 1 for the full list of implemented procedures. Note however that our simulation has been designed so that new NFS procedures can be easily added. The implemented NFS procedures are invoked on the requests generated by `ClientFileSystem` objects. As the name suggests, these objects mimic the file system layer in operating systems. In particular, they are stateful – they keep track of the files opened by the user. Each client holds a reference to a `Server` object as all NFS clients know the address of the remote file server, and exposes file system operations as instance methods. These methods generate sequences of RPC requests to the NFS procedures implemented by the server. These requests are to be fulfilled by a scheduler (see section 3.2) for the client's operations to

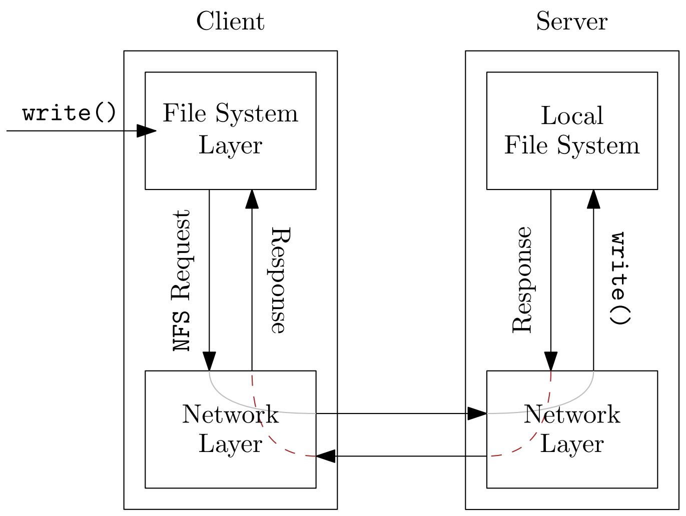

Fig. 1: Typical intearction between clients and servers in the NFSv2 protocol.

take effect. Note that the correspondence between the methods exposed by the client and the server are not one-to-one — for example, the client exposes an `append` method, which attempts to append sequences of characters to the end of an opened file. As the NFSv2 protocol does not define an `append` function, the client implements this method by first issuing an `NFSPROC_GETATTR` request to find the latest file size and then issuing an `NFSPROC_WRITE` request with the offset set to the previously read file size.

Procedure Name	Description
`NFSPROC_GETATTR`	Gets the attribute of a file
`NFSPROC_LOOKUP`	Lookup whether a file exists
`NFSPROC_READ`	Read a specified amount of bytes with a given offset from a file
`NFSPROC_WRITE`	Write specified bytes starting from a given offset to a file
`NFSPROC_CREATE`	Create a new file
`NFSPROC_REMOVE`	Remove a file
`NFSPROC_MKDIR`	Create a new directory
`NFSPROC_RMDIR`	Remove a directory

Table 1: Subset of NFSv2 operations supported in our simulator.

In addition to the behaviors of the procedures, the NFSv2 protocol also outlines several data structures used as arguments and return values of the NFS procedures. The two most important ones are file attributes and file handles. In NFSv2, the file attribute struct records metadata of files and directories, including fields such as file sizes, access rights, and modification times, among other attributes. As a basic demonstration, we only used the file size field, but more fields could be easily added to the simulation should the need arise. On the other hand, the NFSv2 protocol did not define the precise content of the file handle struct, other than stipulating that it exists as a unique identifier of the files and directories on the server. Thus, we simply took it to be the path through the directory tree of the server.

One of the biggest source of consistency issues in NFSv2 is the attribute cache. Instead of performing an RPC on every file operation, thereby stressing the network, the NFSv2 protocol allowed each client process to hold a local cache of file attributes that is periodically invalidated by the client. However, it is possible for the attribute cache to contain outdated information, and clients performing updates that use stale fields demonstrate race conditions. We have implemented attribute cache in our simulation, but the cache currently remains unused. That is, for all client file system operations that need to access the attribute of the file, that operation will issue a new `NFSPROC_GETATTR` request. The reason for this is two folds. First, since our goal is to demonstrate the unexpected effects of concurrency, doing so without involving attribute cache makes our findings even more convincing – the race conditions stem from the protocol itself, not just the attribute cache. Second, since the cache may or may not be invalidated before each request, exploring the effect of attribute will double the amount of decisions in each execution. Because we explore all possible scenarios via backtracking (section 3.3), the number of configurations we need to search grows as $O(2^{\bar{N}})$ where $\bar{N}$ is the expected number of NFS operations over all possible ways that concurrent operations are interleaved. Therefore, doubling $\bar{N}$ will drastically increase the size of the search space, making the simulator impractical to use even for very small input sizes.

3.2 Concurrency Exploration

Beyond just being able to simulate the effect of a fixed sequence of NFS operations, our simulator explores all the possible outcomes of m client processes $p_1, \ldots, p_m$ each executing a potentially different program. We achieve this by examining all allowable permutations of the NFS requests generated by all processes. Note that our simulator accepts as input full programs with control structures, so the number of NFS requests issued by each process may be different depending on the state of the server and what NFS requests has been served thus far. To see this, suppose there are two processes p_1 and p_2 both executing the program in algorithm 1. If p_1 finishes the program before p_2 starts, then p_2 will issue only 1 NFS request (`NFSPROC_GETATTR`) to check the size of the open file, and then immediately terminate. On the other hand, if p_1 and p_2 execute in lock steps, meaning that p_2 issues an NFS request right after p_1 issues an request, then both processes will issue 2 NFS requests, since they will both read that $\mathcal{F}$ is empty.

To ease the discussion of the searching algorithm and the subsequent pruning optimization (section 3.3), we formalize the notion of steps, histories, and execu-

Algorithm 1 Example program demonstrating history dependence

```
1: $\mathcal{F} \leftarrow$ an initially empty open file
2: if size of $\mathcal{F} = 0$ then
3:     Write # to $\mathcal{F}$
4: end if
```

tions. Suppose there are m processes $p_1, \ldots, p_m$ that are all clients of the same NFS server $\mathcal{S}$, each executing a potentially different *deterministic* programs. The programs, of which algorithm 1 is an example, consist of local computation steps separated by NFS requests.

We say that process p_i *takes a step* when p_i receives a response from the previous NFS request and finishes all of its local computation steps until a new NFS request is issued, or terminates if there are no more NFS requests for p_i to make. Note here that the definition assumes an outstanding NFS request exists for each process's programs. When processes first start up, they might have local comptuation steps before making their first NFS requests. Therefore, our definition here have assumed (without loss of generality) that the processes start at the state after they issue their first request. We call these states the *initial states*.

We let a *history* h be a finite sequence $h = (h_0, h_1, \ldots)$ of process labels $(1 \leq h_i \leq m)$. A history records the processes that took steps in the order that they appear in the sequence. For example, the history $(0, 1, 0)$ indicates that process p_0 took a step, followed by a step of process p_1, finally followed by a step of p_0. We remark that because we strict the programs executed by each process to be deterministic, specifying the history completely specifies the state of all p_i and the state of $\mathcal{S}$.

Finally, we define an execution α to be a history for which every process has terminated. We say that two executions α and β are indistinguishable, and write $\alpha \sim \beta$, if (i) every process p_i takes the same number of steps in α and β, (ii) the response from each NFS request by p_i is the same in both α and β, and (iii) the state of the local file system of $\mathcal{S}$ is the same in both α and β. Note that the definition of indistinguishability can also be applied to histories. The goal of our simulator is thus to find the maximum set A of executions for which no two different executions $\alpha \in A$ and $\alpha' \in A$ satisfy $\alpha \sim \alpha'$.

The idea of our searching algorithm is very simple. We use a depth-first search algorithm to incrementally build histories until every process terminates, at which point we add the execution to the set A if no execution in A is indistinguishable from the newly constructed execution. The outline of the algorithm is given in algorithm 2. In the algorithm, h stands for the history upon which the Search algorithm is trying to extend, and r is the sequence of NFS responses in accordance with h. That is, $|r| = |h|$ and r_i is the result of letting p_{h_i} take a step. Finally, $P = \{p_i : 1 \leq i \leq m\}$ is the set of all processes with their programs at their initial states. Note that a depth-first search automatically handles the history dependence of processes.

Algorithm 2 Searching algorithm without pruning

```
 1: procedure SEARCH($h, r, P$)
 2:     $A \leftarrow \emptyset$                                  ▷ The set of all executions to return
 3:     $allTerm \leftarrow \top$              ▷ Keep track of whether all processes have terminated
 4:     for $p_i \in P$ do                                                  ▷ Backtrack by $p_i$
 5:         if $p_i$ has not terminated then
 6:             $allTerm \leftarrow \bot$
 7:             Replay $h$              ▷ Ensures all $p \in P$ are at the state described by $h$

 8:             $h' \leftarrow (h, i)$              ▷ Append process label $i$ to the end of history $h$
 9:             $resp \leftarrow$ handle $p_i$'s outstanding request         ▷ Record returned values
10:             $r' \leftarrow (r, resp)$                           ▷ Append $resp$ to the end of $r$

11:             Let $p_i$ take a step
12:             $A \leftarrow A \cup \{\alpha \in \text{SEARCH}(h', r', P) : \nexists \alpha' \in A \text{ s.t. } \alpha \sim \alpha'\}$
13:         end if
14:     end for
15: end procedure

16: if $allTerm = \top$ then                                ▷ The passed in $h$ is an execution
17:     return $\{h\}$                               ▷ The singleton $h$ is a set of executions
18: else
19:     return $A$
20: end if
```

In algorithm 2, we assumed the ability for our search algorithm to control client programs by only allowing them to take one step at a time. This could either be achieved via multithreading with synchronization control, where each client process runs in a separate thread with the simulator controlling execution by semaphores. However, this quickly becomes impractical since as line 7 shows, a new set of threads must be constructed for each iteration in this approach. Therefore, we chose to use the coroutine syntax available in Python. Each time that a client code would issue an RPC, it instead yields an request object documenting its intention and the arguments of the call, and transfers control back to the simulator, knowing that the simulator would eventually serve this request and send the responses back to the process. In some sense, this mirrors the cooperate scheduling approach in multithreading, where the simulator is the scheduler.

3.3 Search Space Pruning

While algorithm 2 is correct, it can potentially perform many redundant computation. To see this, suppose there are two processes p_1 and p_2 each only performing I/O to files `foo` and `bar`, respectively. In this case, we easily see that all of p_1 and p_2's NFS requests *commute*, and regardless of how the requests are interleaved, any two executions are indistinguishable. However, algorithm 2 will faithfully search all permutations of requests, oblivious to the fact that all requests commute. In this section, we introduce a technique that prunes the search space by taking the commutativity of requests into account.

Formally, we say that two requests by p_i and p_j at some history h (where necessarily $p_i \neq p_j$) *commute* if $(h, i, j) \sim (h, j, i)$. That is, whether or not p_i takes a step before p_j or p_j takes a step before p_i does not change the outcome. Note that this relation is symmetric.

It is easy to see that, for the set of NFS operations we support, two requests r and r' commute if any one of the following is satisfied —

1. r and r' are both file operations (`GETATTR`, `LOOKUP`, `READ`, `WRITE`, `CREATE`, `REMOVE`) and they operate on different files;
2. r and r' are both directory operations (`MKDIR`, `RMDIR`) and they operate on different directories,
3. r is a file operation and r' is a directory operation, and the file that r operates on is not contained in the directory r' operates on;
4. r and r' operate on the same file, and both are of types `GETATTR`, `LOOKUP`, or `READ`.

Note that the last condition is true stems from our choice to only include the size field of file attributes. In a more complete implementation of the NFSv2 protocol, file attributes would contain a last accessed field which gets updated when a `READ` is performed.

It is tempting to modify algorithm 1 in the following way: for every given h, we partition P into disjoint sets $\{P_i\}$ where for all $p_\ell, p_m \in P_i$, the requests by p_ℓ and p_m commute. Then, we backtrack not on individual processes (line 4), but rather the sets of processes we obtained by partitioning P. Two things are of note with this approach. First, the commute relation is not transitive. Indeed, let requests r_1 and r_2 be writes to file `foo`, and let request r_3 be a write to file `bar`. Then, both r_1 and r_2 commutes with r_3, yet r_1 does not commute with r_2. This issue does not disprove the correctness of the approach, however, since a partition of P with the stated property can always be constructed — the trivial partition $\{\{p_i\} : p_i \in P\}$ is one example. To see that this approach is indeed incorrect, consider the example program in algorithm 3. Suppose there are two processes p_1 and p_2 that execute the program concurrently. By the partitioning approach, the first requests of p_1 and p_2 commute, and thus will be taken as a group. Then, the second requests of p_1 and p_2 commute again, and thus these two steps will again be taken as a group. From this point onward, however, $\mathcal{F}$ can only contain 1 character when both programs finish — either 1 or 2. Yet it is possible for $\mathcal{F}$ to contain 12 (21), if p_2 (p_1) is scheduled to execute after p_1 (p_2) completely finishes. Thus, such an approach is incorrect.

The way we approach pruning is through memoization, where we make two modifications to algorithm 2. First, for each history h, we generate its "canonical representation" $\mathcal{C}(h)$, and record that $\mathcal{C}(h)$ has already been explored after all of the recursive calls (line 4) finishes. Second, whenever Search gets called, we first check whether the canonical form of the argument h has already occurred in the memoization table. If so, we are assured that all executions stemming from

Algorithm 3 Example programs demonstrating the incorrectness of a partitioning approach

```
Require: $i$ is the process label
1: $\mathcal{F} \leftarrow$ open file /foo.txt          ▷ Request to call NFSPROC_LOOKUP
2: $s \leftarrow$ size of $\mathcal{F}$                  ▷ Request to call NFSPROC_GETATTR
3: write $i$ to the beginning of $\mathcal{F}$           ▷ Request to call NFSPROC_WRITE
```

h has already been explored, and thus we can safely terminate. The pseudocode for creating a canonical representation is given in algorithm 4.

Algorithm 4 Canonical representation for a history h

```
 1: function $\mathcal{C}(h)$
 2:     $r \leftarrow ()$                          ▷ $r$ stores the canonical representation
 3:     $p \leftarrow ()$                          ▷ $p$ stores part of $r$
 4:     for $i \in h$ do
 5:         if $p \neq ()$ and $i$ does not commute with the last element of $p$ then
 6:             $r \leftarrow (r, \text{sorted } p)$
 7:             $p \leftarrow ()$
 8:         end if
 9:         $p \leftarrow (p, i)$                   ▷ Append $i$ to $p$
10:     end for
11:     $r \leftarrow (r, \text{sorted } p)$
12:     return $r$
13: end function
```

4 Results

In this section, we give actual examples of using our concurrency explorer and look at a few interesting scenarios generated by our program. We will also briefly discuss the effects of search space pruning.

4.1 Exploring Concurrency

Consider the scenario of two processes p_1 and p_2 both trying to write their process labels (1 or 2) to the file `/foo.txt` hosted on the same NFS file server. For simplicity, our sever starts off with having files `foo.txt` and `bar.txt` in its top-level directory. The source code to explore the outcomes in this scenario through our program is given in listing 1. The "main" functions for the processes are given in `proc_1_main` and `proc_2_main`, respectively. Note that the simulator object (on line 23) takes pointers to Python function objects, so all evaluations are done dynamically. Furthermore, the client code are almost written as normal programs with I/O calls, where we expect the file system to be stateful and that it manages pointer to the file buffer. As we have previously discussed, the only notable change is to use a coroutine syntax on every file system operation to transfer control back to the simulator.

```python
from sim.sim import Sim
from sim.server import Server
from sim.client_filesys import ClientFileSystem


def proc_1_main(server: Server):
    fs = ClientFileSystem(server)
    fd = yield from fs.open('/foo.txt')

    for _ in range(3):  # Write "1" three times
        yield from fs.write(fd, '1')


def proc_2_main(server: Server):
    fs = ClientFileSystem(server)
    fd = yield from fs.open('/foo.txt')

    for _ in range(3):  # Write "2" three times.
        yield from fs.write(fd, '2')


if __name__ == '__main__':
    sim = Sim([proc_1_main, proc_2_main])
    sim.explore(verbose=True, prune=True)
    sim.summarize()
```

Listing 1: Source code to explore all scenarios in which processes p_1 and p_2 write their process labels to the same file three times.

Because the writes happen concurrently, we expect there to be 8 scenarios. This is because each of the 3 characters in `/foo.txt` after both p_1 and p_2 terminate could be either 1 or 2, creating a total of 2^3 possibilities. As expected, the simulator reports finding 8 unique executions. Listing 2 shows a partial output of running the program. The first two lines in each scenario shows the responses seen by each process over the course of the execution. The history line prints the exact execution that produces the current scenario, and the last line prints the directory tree of the server in a JSON format. Note that multiple histories could correspond to the same scenario, but we only print a representative one in the output.

```
==================================================
The simulation found 8 unique executions.
```

```
==================================================

--------------------------------------------------
Scenario #1
--------------------------------------------------
p1: [('NFS_OK',), ('NFS_OK',), ('NFS_OK',), ('NFS_OK',)]
p2: [('NFS_OK',), ('NFS_OK',), ('NFS_OK',), ('NFS_OK',)]
History: [1, 1, 2, 2, 2, 1, 1, 2]
Server : {"foo.txt": "212", "bar.txt": ""}
--------------------------------------------------

--------------------------------------------------
Scenario #2
--------------------------------------------------
p1: [('NFS_OK',), ('NFS_OK',), ('NFS_OK',), ('NFS_OK',)]
p2: [('NFS_OK',), ('NFS_OK',), ('NFS_OK',), ('NFS_OK',)]
History: [1, 2, 2, 1, 1, 1, 2, 2]
Server : {"foo.txt": "122", "bar.txt": ""}
--------------------------------------------------

(... 6 more scenarios after this)
```

Listing 2: Partial output of the program in listing 1.

A more interesting scenario is obtained by changing the `fs.open` calls in lines 11 and 19 of listing 1 to `fs.append`. As discussed in Section 3.1, we implemented an append method to our clinet filesystem layer, in which we issue two NFS requests to first get the size s of the file and then to write to the file with s as the offset. It might be tempting to think that at the end, `/foo.txt` will for sure contain 6 characters, however such is not the case. In total, our simulator found 62 unique executions, where the file can contain 3 to 6 characters. Some of the scenarios are shown in 3. Indeed, it is even possible for the file to contain `111`, if p_1 and p_2 execute in lock steps.

```
--------------------------------------------------
Scenario #7
--------------------------------------------------
p1: [('NFS_OK',), ('NFS_OK',), ('NFS_OK',), ...]
p2: [('NFS_OK',), ('NFS_OK',), ('NFS_OK',), ...]
History: [1, 1, 1, 2, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2]
Server : {"foo.txt": "121212", "bar.txt": ""}
--------------------------------------------------

--------------------------------------------------
Scenario #28
--------------------------------------------------
p1: [('NFS_OK',), ('NFS_OK',), ('NFS_OK',), ...]
```

```
p2: [('NFS_OK',), ('NFS_OK',), ('NFS_OK',), ...]
History: [1, 1, 2, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2, 1]
Server : {"foo.txt": "111", "bar.txt": ""}
--------------------------------------------------
```

Listing 3: Partial output after changing the `fs.open` calls in lines 11 and 19 of listing 1 to `fs.append`.

4.2 Effects of Pruning

To test the correctness of our pruning algorithm, we have compared the output for both scenarios described in the previous section with and without the pruning enabled. In both cases, the pruning made no difference to the output. Additionally, we ran a test with 4 processes, each first writing its process label to file `/foo.txt`, and then writing the process label to file `/bar.txt`. The goal of the second test case is to create situations where many requests commute with each other but do not commute transitively. As expected, the simulator found 16 unique executions, since the character in each file could be any character in $\{1, 2, 3, 4\}$, creating $4 \times 4 = 16$ possibilities.

We have also investigated the effectiveness of our pruning procedure. Because the effectiveness of pruning depends on the number of commuting operations, we considered two sets of scenarios, as follows:

- *Scenario A_n:* Both processes p_1 and p_2 write their process labels to file `/foo.txt` n times,
- *Scenario B_n:* Processes p_1 and p_2 append their process labels to files `/foo.txt` and `/bar.txt` n times, respectively.

Table 2 summarizes our findings for both scenarios for several values of n. Each number is the average of 7 runs using the CPython interpreter, with standard deviations reported in parentheses. These values are obtained on a 2018 model 15-inch Macbook Pro with 2.6GHz Intel i7 processors. When most requests do not commute, as is the case in scenario A_n's, our pruning optimization does not make a big difference. In fact, it slows down the overall execution due to the overhead in reading from and writing to a memoization table. However, all requests commute in the set of B_n scenarios, so our pruning should be effective. Indeed, our findings confirm the intuition.

4.3 Limitations

One of the primary limitations of our work is that even though pruning is introduced to reduce the exponentially large search space, the time complexity of our algorithm is inherently high, and pruning may not be as effective when almost no requests commute. This means that our tool is best suited to explore concurrency in small models with not too many requests. However, even complicated concurrency behavior arise from small examples that our tool could handle, so

n	Scenario A		Scenario B	
	No Pruning	Pruning	No Pruning	Pruning
1	3.974 (0.188)	3.707 (0.096)	3.823 (0.134)	2.679 (0.095)
2	20.276 (0.297)	20.659 (0.646)	20.868 (0.173)	15.1 (0.061)
3	106.096 (6.445)	102.612 (1.478)	106.005 (0.303)	76.574 (0.365)
4	481.88 (5.886)	485.195 (9.42)	466.465 (20.178)	350.738 (17.151)
5	2188.83 (13.02)	2223.53 (30.37)	2146.90 (49.86)	1570.55 (34.581)
6	11376 (1284)	11178 (436)	9406 (101)	6891 (87.5)
7	62080 (6178)	70565 (9216)	60605 (2790)	32829 (1725)

Table 2: Benchmarked run times in miliseconds.

we think our tool is valuable as an instructional demonstration. Additionally, the tool as of right now is single threaded, and parallelization could potentially speed the program up. However, the processes need to frequently access a shared memoization table to lookup and store search spaces that have already been explored. Furthermore, it is tricky to coordinate the processes to not concurrently explore equivalent subspaces.

Another drawback is the lack of involvement of the attribute cache in the concurrency exploration. As discussed in section 3.1, however, including the attribute cache will greatly increase the time complexity of our search, making the tool practically impossible to use. Besides, we felt it was enough to demonstrate the inherent consistency issues in the protocol itself, without adding the issue of cache consistency into the mix.

Finally, as a initial demonstration, we only supported a subset of the NFSv2 protocol. However, the code is organized such that new procedures could be easily added.

5 Conclusion

In this paper we presented a simulator that can demonstrate the concurrent activities of a network file system. The simulation includes all possible variations of concurrent writes in a network file system. With our simulation tool, we wish to strengthen the user's understanding of distributed file systems.

Depending on the order in which the server receives write requests from clients, the operations of a network file system may have unforeseen consequences. Furthermore, the state of the cache and the file system's consistency model may have an unforeseen impact on the operations of a network file system. The simulator will analyze all conceivable event sequences as observed by the server and provide images for each scenario, each with its own conclusion.

As an instructive example, our suggested approach is best suited to investigate concurrency in tiny models with fewer queries. We only supported a part of the NFSv2 protocol in the original investigation. In the future, we want to take this into consideration and build on the new network file system protocols. We plan to work towards enabling larger models into the simulation while taking the NFS attribute cache into account.

References

1. Sandberg, Russel & Goldberg, David & Walsh, Dan & Lyon, Bob. (2002). Design and Implementation of the Sun Network Filesystem. Innovations in Internetworking. 379.
2. Lombard, Pierre & Denneulin, Yves & Valentin, Olivier & lèbre, Adrien. (2003). Improving the Performances of a Distributed NFS Implementation. 3019. 405-412. 10.1007/978-3-540-24669-5_53.
3. Nowicki, W.I. (1989). NFS: Network File System Protocol specification. RFC, 1094, 1-27.
4. Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., & Hitz, D. (1994). NFS Version 3: Design and Implementation. USENIX Summer.
5. C, Rev & Hitz, Dave & Lau, James & Malcolm, Michael. (2000). File System Design for an NFS File Server Appliance.
6. Sandberg, R. (2001). The Sun Network Filesystem: Design, Implementation and Experience.
7. Muntz, D. (2001). Building a Single Distributed File System from Many NFS Servers -or- The Poor-Man's Cluster Server.
8. Kulkarni, O., Gawali, D., Bagul, S.S., & Meshram, D.B. (2011). Study of Network File System (NFS) And Its Variations.
9. Xu, Ying & Fleisch, Brett. (2004). NFS-cc: tuning NFS for concurrent read sharing. IJHPCN. 1. 203-213. 10.1504/IJHPCN.2004.008349.
10. UCLA. (n.d.). CS 111: Lecture 16 - Robustness, Parallelism, and NFS. `Http://Web.Cs.Ucla.Edu`.
11. O'Reilly & Associates. (n.d.). NFS async thread tuning (Managing NFS and NIS, 2nd Edition). `https://Docstore.Mik.Ua/Orelly/Bookshelfs.Html`
12. Amazon S3 Update Strong Read-After-Write Consistency. (n.d.). Amazon Web Services. `https://aws.amazon.com/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/`
13. Apache Hadoop Amazon Web Services support – S3Guard: Consistency and Metadata Caching for S3A. (n.d.). Apache. `https://hadoop.apache.org/docs/r3.0.3/hadoop-aws/tools/hadoop-aws/s3guard.html`.
14. Matthieu Perrin, 2 - Overview of Existing Models, Editor(s): Matthieu Perrin, Distributed Systems, Elsevier, 2017, Pages 23-52, ISBN 9781785482267, `https://doi.org/10.1016/B978-1-78548-226-7.50002-1`.
15. Homework. (n.d.). `Https://Pages.Cs.Wisc.Edu`. Retrieved December 12, 2021, from `https://pages.cs.wisc.edu/7Eremzi/OSTEP/Homework/homework.html`.