

the reality is a lot or more complex).

Okay, but local filesystems are not the problem here. When you unbar the same file into a directory provided by NFS, you will see the following output from a packet sniffer. I would like to add that no special mount options were used that could have some impact how things are working internally. This was made with a plain standard mount.

```
 1 0.000000 0.000000 10.0.2.11 -> 10.0.2.10 NFS 194 V3 LOOKUP Call, DH: 0xad487762/singlefile
 2 0.000178 0.000178 10.0.2.10 -> 10.0.2.11 NFS 186 V3 LOOKUP Reply (Call In 1) Error: NFS3ERR_NOENT
 3 0.000260 0.000438 10.0.2.11 -> 10.0.2.10 NFS 178 V3 GETATTR Call, FH: 0xad487762
 4 0.000052 0.000490 10.0.2.10 -> 10.0.2.11 NFS 182 V3 GETATTR Reply (Call In 3) Directory mode: 0777 uid: 0
gid: 0
 5 0.000242 0.000732 10.0.2.11 -> 10.0.2.10 NFS 226 V3 MKDIR Call, DH: 0xad487762/singlefile
 6 0.003430 0.004162 10.0.2.10 -> 10.0.2.11 NFS 342 V3 MKDIR Reply (Call In 5)
 7 0.000303 0.004465 10.0.2.11 -> 10.0.2.10 NFS 178 V3 GETATTR Call, FH: 0xad487762
 8 0.000053 0.004518 10.0.2.10 -> 10.0.2.11 NFS 182 V3 GETATTR Reply (Call In 7) Directory mode: 0777 uid: 0
gid: 0
 9 0.000273 0.004791 10.0.2.11 -> 10.0.2.10 NFS 182 V3 ACCESS Call, FH: 0x44856022, [Check: RD LU MD XT
DL]
10 0.000070 0.004861 10.0.2.10 -> 10.0.2.11 NFS 190 V3 ACCESS Reply (Call In 9), [Allowed: RD LU MD XT DL]
11 0.000215 0.005076 10.0.2.11 -> 10.0.2.10 NFS 178 V3 GETATTR Call, FH: 0x44856022
12 0.000076 0.005152 10.0.2.10 -> 10.0.2.11 NFS 182 V3 GETATTR Reply (Call In 11) Directory mode: 0755 uid:
100 gid: 10
13 0.000175 0.005327 10.0.2.11 -> 10.0.2.10 NFS 194 V3 LOOKUP Call, DH: 0x44856022/testfile1
14 0.000044 0.005371 10.0.2.10 -> 10.0.2.11 NFS 186 V3 LOOKUP Reply (Call In 13) Error: NFS3ERR_NOENT
15 0.000188 0.005559 10.0.2.11 -> 10.0.2.10 NFS 178 V3 GETATTR Call, FH: 0x44856022
16 0.000035 0.005594 10.0.2.10 -> 10.0.2.11 NFS 182 V3 GETATTR Reply (Call In 15) Directory mode: 0755 uid:
100 gid: 10
17 0.000179 0.005773 10.0.2.11 -> 10.0.2.10 NFS 178 V3 GETATTR Call, FH: 0x44856022
18 0.000033 0.005806 10.0.2.10 -> 10.0.2.11 NFS 182 V3 GETATTR Reply (Call In 17) Directory mode: 0755 uid:
100 gid: 10
19 0.000166 0.005972 10.0.2.11 -> 10.0.2.10 NFS 178 V3 GETATTR Call, FH: 0x44856022
20 0.000030 0.006002 10.0.2.10 -> 10.0.2.11 NFS 182 V3 GETATTR Reply (Call In 19) Directory mode: 0755 uid:
100 gid: 10
21 0.000191 0.006193 10.0.2.11 -> 10.0.2.10 NFSACL 182 V3 GETACL Call
22 0.000034 0.006227 10.0.2.10 -> 10.0.2.11 NFSACL 206 V3 GETACL Reply (Call In
21)
23 0.000203 0.006430 10.0.2.11 -> 10.0.2.10 NFS 194 V3 LOOKUP Call, DH: 0x44856022/testfile1
24 0.000033 0.006463 10.0.2.10 -> 10.0.2.11 NFS 186 V3 LOOKUP Reply (Call In 23) Error: NFS3ERR_NOENT
25 0.000155 0.006618 10.0.2.11 -> 10.0.2.10 NFS 238 V3 CREATE Call, DH: 0x44856022/testfile1 Mode:
GUARDED
26 0.001257 0.007875 10.0.2.10 -> 10.0.2.11 NFS 342 V3 CREATE Reply (Call In 25)
27 0.000189 0.008064 10.0.2.11 -> 10.0.2.10 NFS 178 V3 GETATTR Call, FH: 0xd98a8154
28 0.000032 0.008096 10.0.2.10 -> 10.0.2.11 NFS 182 V3 GETATTR Reply (Call In 27) Regular File mode: 0600
uid: 100 gid: 10
29 0.000237 0.008333 10.0.2.11 -> 10.0.2.10 NFS 1222 V3 WRITE Call, FH: 0xd98a8154 Offset: 0 Len: 1024
FILE_SYNC
30 0.001254 0.009587 10.0.2.10 -> 10.0.2.11 NFS 230 V3 WRITE Reply (Call In 29) Len: 1024 FILE_SYNC
31 0.000202 0.009789 10.0.2.11 -> 10.0.2.10 NFS 178 V3 GETATTR Call, FH: 0xd98a8154
32 0.000037 0.009826 10.0.2.10 -> 10.0.2.11 NFS 182 V3 GETATTR Reply (Call In 31) Regular File mode: 0600
uid: 100 gid: 10
33 0.000172 0.009998 10.0.2.11 -> 10.0.2.10 NFS 222 V3 SETATTR Call, FH: 0xd98a8154
34 0.001084 0.011082 10.0.2.10 -> 10.0.2.11 NFS 214 V3 SETATTR Reply (Call In 33)
35 0.000255 0.011337 10.0.2.11 -> 10.0.2.10 NFS 222 V3 SETATTR Call, FH: 0x44856022
36 0.000997 0.012334 10.0.2.10 -> 10.0.2.11 NFS 214 V3 SETATTR Reply (Call In 35)
```

As you see, a lot is going on. You may say, the performance difference between a local tar and a tar on a NFS filesystem is the network. Of course there is some latency in the story when you have to communicate that often over the network.

And so you are for sure partially right, but that is only half the story and it doesn't explain the 12ms (which are interestingly not that far away from the worst case rotational latency of a 5400 rpm disk). The rest of the story is that the

load is transformed by using NFS.

There is an important passage in RFC 1813. It states: Data-modifying operations in the NFS version 3 protocol are synchronous. When a procedure returns to the client, the client can assume that the operation has completed and any data associated with the request is now on stable storage. Furthermore it elaborates: The following data modifying procedures are synchronous: WRITE (with stable flag set to FILE_SYNC), CREATE, MKDIR, SYMLINK, MKNOD, REMOVE, RMDIR, RENAME, LINK, and COMMIT.

Not in this list is the SETATTR command, which forces the data as well to stable storage before completing at least to my knowledge and what's visible in the copies of nfs_srv.c out there in the internet.

When you are looking onto the tcpdump you see a significant number of the mentioned operations. Now imagine the same for a tar file of 290000 small files.

So essentially using NFS transform the load formerly consisting out of a multitude asynchronous write operations into one executing a significant number of operations that are synchronous by definition. What was a async write on application layer, is a sequence of synchronous writes when it hits the NFS server and thus hits the filesystem underneath. A rather total transformation of the load.

However, the situation is not that bad in reality. There are some optimizations. Let's have a look at a different tcpdump. I've created a tar file with files with four lengths in it: 1k, 32k, 33k and 64k. The tcpdumps are reduces to CREATE, WRITE and COMMIT calls.

We start with the 32k file. You see it's written as UNSTABLE. So the call is allowed to return without persisting the data to disk before. But at close there is a COMMIT forcing NFS to persist everything.

```
25 0.000284 0.007877 10.0.2.11 -> 10.0.2.10 NFS 234 V3 CREATE Call, DH: 0x24bc098f/32kfile Mode:
GUARDED
53 0.000002 0.015328 10.0.2.11 -> 10.0.2.10 NFS 846 V3 WRITE Call, FH: 0x58dd2c54 Offset: 0 Len: 32768
UNSTABLE
65 0.000366 0.018675 10.0.2.11 -> 10.0.2.10 NFS 190 V3 COMMIT Call, FH: 0x58dd2c54
```

Let's have a look onto the 1k file.

```
79 0.000351 0.021063 10.0.2.11 -> 10.0.2.10 NFS 234 V3 CREATE Call, DH: 0x24bc098f/1kfile Mode:
GUARDED
83 0.000867 0.024100 10.0.2.11 -> 10.0.2.10 NFS 1222 V3 WRITE Call, FH: 0xc5d2cd22 Offset: 0 Len: 1024
FILE_SYNC
```

The write is directly send as a FILE_SYNC write. But why? It's pretty obvious, when you look at what's missing. There is no COMMIT. At the end you save a command you don't have to send. One round-trip time less.

For the interest of curiosity, I look at the 33k size and the 64k sizes. At first 33k:

```
101 0.000348 0.036633 10.0.2.11 -> 10.0.2.10 NFS 234 V3 CREATE Call, DH: 0x24bc098f/33kfile Mode:
GUARDED
131 0.000001 0.040300 10.0.2.11 -> 10.0.2.10 NFS 846 V3 WRITE Call, FH: 0x38903d60 Offset: 0 Len: 32768
UNSTABLE
135 0.000160 0.040743 10.0.2.11 -> 10.0.2.10 NFS 1222 V3 WRITE Call, FH: 0x38903d60 Offset: 32768 Len:
1024 FILE_SYNC
142 0.001003 0.046502 10.0.2.11 -> 10.0.2.10 NFS 190 V3 COMMIT Call, FH: 0x38903d60
```

Now at 64k :

```
156 0.000239 0.050702 10.0.2.11 -> 10.0.2.10 NFS 234 V3 CREATE Call, DH: 0x24bc098f/64kfile Mode:
GUARDED
187 0.000001 0.054108 10.0.2.11 -> 10.0.2.10 NFS 846 V3 WRITE Call, FH: 0xa59fdc16 Offset: 0 Len: 32768
UNSTABLE
214 0.000001 0.055688 10.0.2.11 -> 10.0.2.10 NFS 846 V3 WRITE Call, FH: 0xa59fdc16 Offset: 32768 Len: 32768
UNSTABLE
227 0.000389 0.066764 10.0.2.11 -> 10.0.2.10 NFS 190 V3 COMMIT Call, FH: 0xa59fdc16
```

I'm not an expert for the NFS code, but my assumption about the assumption behind this behavior that it is expected that there will be more writes into the same file if you fill up the wsize. If you don't fill it, it's expected that it may be the only or the last write.

So it's not as easy as saying that NFS is always writing synchronously. It was the case in NFSv2 and was a source of a lot of NFS performance problems. NFSv3 introduced the concept of asynchronous NFS writes with a synchronous COMMIT at the end. But the "many small files"-tar counters this as you may have taken away from the tcpdumps. So: If you write just 290000 files mostly smaller than 32k for example (the configuration at that customer) you will see a lot of NFSv3 sync writes and almost no asynchronous writes and almost no commits. If you had one of few big files in the tar with the same total size you would write the amount of data much faster, because then you would have a lot asynchronous NFS writes and final synchronous NFS write with a closing NFS COMMIT.

I was able to show by this to the customer that the execution time of the tar -x and the difference between the local filesystem and a remote filesystem via NFS is the expected behavior given the configuration of their storage pool.

How do you improve this situation? The problem is there for quite a time. And there were a multitude of solutions to it.

Just to give you a perspective how long this problem exists: Perhaps some of you are old enough to remember things like the Prestoserv NFS accelerator, which was quite useful in NFSv2 as each and every WRITE Call had to be answered synchronously (which didn't know the method of a sequence of ASYNC write with a following COMMIT, obviously ... COMMIT was introduced in NFSv3). It solved this problem for example.

Basically the answer to the problem is always: Making write latency as fast as possible. And so it boils down to get rid of the need to hit rotating rust before being able to answer on the mentioned NFS calls. One solution for example used on ZFS Storage Appliances are SSD for the log devices. A write is considered as non-volatile as soon as it's on the log device.

Out of this reasons when working with customers I have two rules: 1. You don't get a ZFS Storage Appliance from me without having write-accelerating SSD as long as the you and me can't prove that your load won't be hit by the mentioned necessities of the protocol (which is quite hard, when it hides even in such a simple thing like at tar) or you tell me, that you understand the problem but still want it otherwise. At the end you are the customer and customer is king as we say in Germany. 2. When you tell me you run NFS from a server (let's say DIY ZFS Storage Appliance) you should either have a storage array behind with a non-volatile write cache (that should be really used and not switched off, just saying out of experience) or you should opt for log devices as well. Without it you have essentially a storage array with cache but with a wrecked up battery which goes into write through mode because of this and you get the performance you can expect out of such a setup.

And as a fun fact: You may think that tar -x is not that problematic because you do it only from time to time. But on the contrary: I've seen quite a number of customers in the past, where tar is production critical. For example unpacking sensor data, development file servers where quite a number of tar unpacks ran all day, diagnostic data of machines, something like that. This small little tape archive tool is used quite frequently for a multitude of tasks where you don't expect them.

What the conclusion out of this:

Well, things are most of the time not as easy as they seem.
Sometimes the simple loads are the most nasty and pathological ones.
It pays to read the RFCs

Posted by Joerg Moellenkamp in English, Solaris at 13:40

You say:

"The following data modifying procedures are synchronous: WRITE (with stable flag set to FILE_SYNC), CREATE, MKDIR, SYMLINK, MKNOD, REMOVE, RMDIR, RENAME, LINK, and COMMIT."

.....
"So essentially using NFS transform the load formerly consisting out of a multitude asynchronous write operations into one executing a significant number of operations that are synchronous by definition. What was a async write on application layer, is a sequence of synchronous writes when it hits the NFS server and thus hits the filesystem underneath. A rather total transformation of the load."

I think that all these operations: WRITE (with stable flag set to FILE_SYNC), CREATE, MKDIR, SYMLINK, MKNOD, REMOVE, RMDIR, RENAME, LINK, and COMMIT." are also synchronous in ufs, zfs or other filesystem. They can be optimized with a logging filesystem, etc. but they are synchronous.

Anonymous on Apr 27 2017, 22:31

Blog Export: c0t0d0s0.org, <http://www.c0t0d0s0.org/>

At least with ZFS this isn't correct. A rmdir for example doesn't trigger a zil_commit, as long as you don't specify SYNC_ALWAYS.
Anonymous on Apr 28 2017, 13:47