Asynchronous I/O Using Callbacks in NFS

Ravi Sadineni, Sujan Bolisetti, Harshavardhan Ellanti
{*rsadineni,sbolisetti,hellanti*}*@cs.stonybrook.edu*

Abstract

NFS is a highly popular protocol that allows fast and seamless sharing of files across the network. Using NFS, remote filesystems can be mounted very much the way local file systems are mounted. Most of the I/O operations on a NFS mounted device employ similar mechanisms to that of a local device. But the behaviour of an asynchronous read differ significantly. Asynchronous I/O requests are triggered sequentially on a NFS mounted device, waiting for the reply to each of the requests before triggering the next request. This unnecessarily blocks the client until the actual data is transferred back.

We propose Asynchronous read, where the server simply acknoweldge the reciept of the request. This way clients is not blocked until the actual data is sent back. Therefore the client can continue the processing, while several I/O operations are performed in the background without having to block for the completion of I/O. This enables applications to overlap their compute and I/O processing to improve throughput on a per process basis. Also the client can trigger multiple I/O requests concurrently.The server then sends the actual data in a seperate callback channel. Callbacks is one of the several new features added as part of NFS version 4. Callbacks allow servers to make an RPC directed at client. Right now callbacks provide an efficient way for delegating a file to the client by avoiding repeated requests to the server in the absence of inter-client conflicts. Here we use Callbacks to support true asynchronous I/O, where the server can respond to the client requests after fetching the data from the disk. We have implemented asynchronous read operation using callbacks on NFS Ganesha and have tested it using Pynfs as a client.

Acronyms NFS (Network File System), RPC (Remote Procedure Call)

1 Introduction

The Network File System (NFS) allows clients to access data seamlessly over the network. This is accomplished through the same system calls that allow the access of files on the local disk. In the case of NFS these system calls trigger the client stub which sends an RPC across the network to an NFS server. These traditional reads/writes are synchronous by default, which blocks the execution of the application until each of the requested I/O operation is completed. Conversely, asynchronous I/O enables the applications to continue processing while several I/O operations are running in the background. This feature allows the applications to overlap their compute and I/O processing to improve throughput on a per process basis. In Linux, the POSIX asynchronous I/O [1] provides us with system calls to initiate one or more I/O operations asynchronously. Typically an application using an asynchronous I/O interface submits batch I/O and waits for the completion of all the requests in the batch. Similarly, batch I/O operations can also be performed on the files mounted on NFS. In the case of NFS, the client NFS stub triggers each of these requests synchronously and waits for the response before triggering the next request. The time taken for the batch I/O request can be greatly reduced if these requests can be triggered concurrently instead of client waiting for each of the requests to be completed before triggering the next. One reason for this sequential request forwarding is, until NFSv4.0, there is no way by which the server can connect to the client unless the connection is initiated by client. NFSv4.0 had provided a feature of callbacks by which servers can contact the client, there by the server in the callback request will act as a client and client will act as server. This callback feature can be used for making the NFS asynchronous I/O truly asynchronous.So, the clients can trigger the requests concurrently and then wait for the response. On fetching the data from disk, the server uses the callback mechanism to send the data to the client. On receiving the callbacks for all the requests, the client NFS stub signals the completion to the application. This would greatly enhance the throughput of the application as the wait time for the client has reduced. In summary the existing implementations of asynchronous I/O operations in NFS are not fully exploiting the benefits of asynchronous nature.

We propose an novel asynchronous read operation using callbacks which makes the read operation completely asynchronous. Our asynchronous read does not suffer from the sequential request forwarding problem as we are using callbacks to reply to the client, the requests can be sent concurrently, there by we will achieve increased throughput.

For implementing the asynchronous I/O, we have used

NFS-Ganesha [2] as the NFS server and Pynfs [3] as the NFS client.We have used the latest stable version of NFS that is v4.1 for both client and server. NFS-Ganesha is an user level implementation of the NFS server in *C* language. We chose NFS-Ganesha because it is easy to enhance NFS-Ganesha when compared to the kernel NFS source code. NFS-Ganesha is also actively used by a wider audience and also supported by major companies like IBM,Panasas and Redhat. Hence it supports all the latest features of the NFSv4.1 like sessions,callbacks etc. Pynfs is an user level implementation of NFS client and server in /textitpython, it is used as a test suite for checking the correctness of NFS protocol.

2 Background

A brief overview on Linux asynchronous I/O ,NFS Ganesha and about callbacks will help us to understand the current system better.The asynchronous I/O stable version has been first introduced in Linux 2.6. A process that issues an Asynchronous I/O request doesn't have to wait for the availability of the data. Instead, after an I/O request is submitted, the process continues to execute its code and can later check the status of the submitted request. The Linux kernel exposes 5 system calls [4] for supporting asynchronous I/O. We are listing them below with a brief description about each of them.

1. **io_setup()**: This system call is used to create an asynchronous I/O context in the kernel. Asynchronous I/O context is a set of data structures that the kernel provides to perform asynchronous I/O.
2. **io_submit()**: This system call queues the I/O request blocks for processing in the asynchronous I/O context.
3. **io_getevents()**: This system call is used to read the events from the completion queue of the asynchronous I/O context.
4. **io_destroy()**: This system call is used to destroy the asynchronous I/O context.
5. **io_cancel()**: This system call is used to cancel the asynchronous I/O operation previously submitted using io_submit().

Asynchronous I/O also works on NFS mounted files. But the underlying NFS stub initiates each of these requests and waits on the response synchronously. Using callbacks, these requests can be triggered concurrently. Callbacks provides a mechanism for the server to access the client. The client provides the server, its callback program number and port number using SETCLIENTID. The server does a backward path existence check before granting the delegation to the client. This check is done using CB_NULL callback. The use of callbacks is not to be depended upon until the client has proven its ability to receive them. Thus in the implementation of the asynchronous I/O using callbacks, we need to follow the callback initiation steps, which include checking the existence of the backward path. If these checks do not succeed, the asynchronous I/O will fall back to the existing synchronous I/O mechanism. If the checks succeed, the server uses the callbacks to send data to the client.

As we have mentioned in section 1 that NFS Ganesha is an user level implementation of NFS Server. It has a capability to serve multiple file systems at the same time. The other advantage of NFS Ganesha is, it manages huge meta-data cache. Because of this huge cache it can serve most of the nfs requests very fast. NFS Ganesha is built heavily on pthreads. All the request processing is handled only by a pool of threads. The system performs better even in case of heavy loads as it has multiple threads to handle the load. Thus the system provide good guarantees on scalability. In Ganesha, the threads are classifies into two types, one is the dispatcher thread whose task is to decode incoming the nfs request and enqueue in the worker queue. The other type of threads are the worker threads whose task is to dequeue the request, process the request and return the reply to the corresponding client. In case of a error the worker thread will return the appropriate error.

3 Design

On a normal NFS read the client waits till the server responds with the data. This involves server fetching the data from the slow disk. Thus the client is blocked until the server responds. Instead, NFS asynchronous read responds immediately without blocking the client until the data is ready. Before analysing the two versions of NFS reads, let us take a look at the initial steps performed by the client to acquire the *stateid* required to perform a NFS read. First, the NFS client triggers an OP_LOOKUP request to the server looking for the file. On finding the file the server responds with a /textitfilehandle back to the client. The client then issues an OP_OPEN on the /textitfilehandle. On a successful open the server constructs an unique *stateid* and returns it to the client along with the status. In case if the file is already opened, then the above mentioned steps are skipped. The *stateid* represents the state information of current open request. This state information includes, but is not limited to locking/share state, delegation state. Let us now analyse the sequence of steps involved in NFS Read (3.1) and then our asynchronous read using callback mechanism (3.2).

3.1 NFS READ

Figure 1 depicts the architecture of the NFS Read request processing in NFS Ganesha. On an existing system, the client uses the same *stateid* in the OP_READ request. On receiving the nfs request, the dispatcher thread as usual

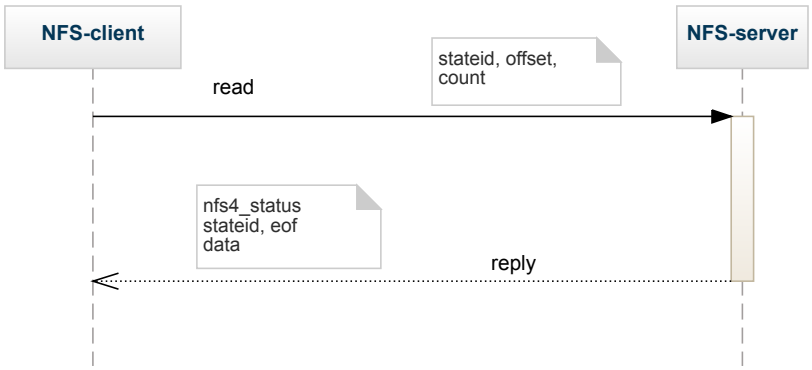

Figure 1: NFS Read Sequence Diagram

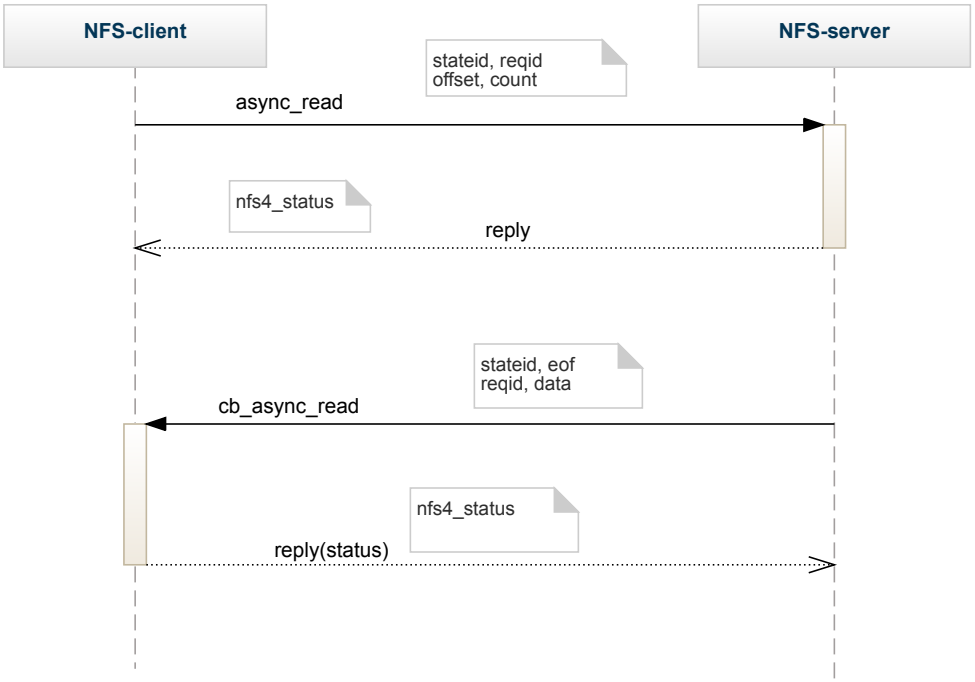

Figure 2: NFS Asynchronous Read Sequence Diagram

decode the request and populate the read arguments in the appropriate structures and enqueue the request in the worker queue. The worker thread will dequeue the request and first checks if the *stateid* is valid. If the *stateid* is invalid, the worker thread responds with a status indicating the stale *stateid*. If valid the worker then checks, if the client has acquired the read/share_read access while opening the file. Once all the checks are passed, the worker thread then performs read from the local file system/Cache.On success the worker thread responds with the NFS4_OK status along with the data. In case of an error, the worker thread responds with an appropriate error status. Note that the client is blocked until the server responds to the request. The worker thread will again go into the waiting state, waiting for the nfs requests after responding to the client.

3.2 Asynchronous Read Using Callbacks

Figure 2 depicts the architecture diagram of the asynchronous read. In the case of asynchronous read, the client will send an ASYNC_READ request. On receiving the nfs request, the dispatcher thread as usual decode the request and populate the ASYNC_READ arguments in the appropriate structures and enqueue the request in the worker queue. The worker thread will dequeue the request and prepare an NFS request with the ASYNC_READ arguments and enqueue in the worker queue. Immediately after the enqueue operation is successful the worker thread will respond to the client using NFS4_OK.On receiving the initial response, client is not blocked any more and is free to perform further tasks. On the server side any other worker thread which is waiting on the worker queue for processing the new requests will the dequeue this new request and process it. This involves performing all the necessary checks like the STATEID valida-

tion and read access validation. Once all the checks are passed the worker thread will perform the actual file system read/cache read depending on the presence of data. After reading the data the worker thread will prepare a callback request and perform the callback operation using CB_ASYNC_READ operation. Client on receiving the CB_ASYNC_READ request identifies the request owner based on the STATEID. Request owner can be a process or a thread on the client that issued the request. Client on receiving the callback, responds with the NFS4_OK back to the server or an error status if any. Now on the client side, the process or the thread that has initiated the request can be identified using *stateid*. But the same thread might have triggered multiple requests. Thus we have added a unique feild in the request called *requestid*. *requestid* enables the process or the thread to uniquely identify the request among the multiple requests that it might have triggered. We also have added an another feild named *timeout* as part of the asynchronous request to the server. *timeout* enables the client to specify the maximum time that the server can take to send a callback to the client. If the client does not recieve a callback in the mentioned *timeout*, the client treats the request as a failure and resends the request.

4 Implementation of Asynchronous Read

4.1 XDR

We generated the request and response structures for our asynchronous read using the External Data Representation (XDR) [5]. External Data Representation (XDR) is a standard data serialization format. XDR allows the data to be transferred seamlessly between different kinds of systems. Let us now look at each of these structures in detail.

```
struct ASYNC_READ4args{
        uint64_t    reqId;
        stateid4    stateid;
        offset4     offset;
        count4      count;
        uint32_t    timeout;
};
```

ASYNC_READ4args : The client sends these argument structure as part of the initial request to the server. Detailed explanation of the arguments in *ASYNC_READ4args* will follow.

reqId : Reqid is a 64 bit integer, generated by the client. This is generated based on the requested *filehandle*, processId/threadId and current_timestamp and will uniquely identify every request. The server passes the same request id to the client as part of the callback along with the requested file data. The client process/thread then uses this *reqId* to uniquely identify the request among multiple requests that it might have triggered to the server. Note that *reqId* is opaque to the server.

stateid : Stateid is a 128-bit quantity returned by a server in the initial open request. It uniquely defines the open and locking state provided by the server for a specific open or lock owner for a specific file. We are using *stateid* on the server side to check the client share/delegation access on the requested file.

offset : Offset (from the start of the file) at which the read has to start in the file.

count: Number of bytes to read from the requested file.

timeout: Number of bytes to read from the requested file.

On receiving the request from the client, the server performs the initial checks. Then the server responds with ASYNC_READ4RES to the client. Now we will explain the status passed as part of ASYNC_READ4RES to the client.

```
struct ASYNC_READ4res{
        nfsstat4            status;
};
```

status : Indicates the status corresponding to initial permission checks on the requested file. On receiving the asynchronous read request on the server side, we are checking the client's share/delegation on the requested file. On success we will return NFS4_OK. If the checks fail, we are returning appropriate error status.

Let us now understand the structure used by the server when making a callback to the client. The server passes *CB_ASYNC_READ4args* as part of the request to the client. We will now look at each of the arguments in detail.

```
union CB_ASYNC_READ4args
        switch(nfsstat4 status){
        case NFS4_OK:
         CB_ASYNC_READ4argsok argok4;
        default:
                void;
};
```

The first argument in the callback request to the client is the status. If the data is fetched successfully from the local file system, the status will be NFS4_OK . In case of an error, an appropriate error status will be sent to the client. If the data is fetched successfully, a second argument of type *CB_ASYNC_READ4argsok* will also be passed in the callback to the client. Otherwise, only sta-

tus will be passed. A detailed explanation of each of the fields in *CB_ASYNC_READ4argsok* will follow.

```
struct CB_ASYNC_READ4argsok{
        uint64_t            reqId;
        bool                eof;
        opaque   data<>;
};
```

reqId: The *reqId* received by the server from the client during the initial asynchronous read request. This is used by the client to identify the owner of the request.

eof : A boolean value indicating if the end of the file has been reached.

data : Requested file data.

On receiving the callback request from the server, client forwards the data to the respective owner based on the *reqId*. Then the client responds with *CB_ASYNC_READ4res* to the server.

```
struct CB_ASYNC_READ4res{
        nfsstat4            status;
};
```

status : Indicates the asynchronous read callback status from the client. On success, status will be NFS4_OK else the corresponding error message will be passed to the server.

5 Evaluation

We have ran our experiments by running the NFS Ganesha server on Amazon EC2 and Pynfs client on the local machine. We have decided to run our experiments on EC2 as we wanted to evaluate our asynchronous read operation in an environment closer to the real world scenarios. As usually in the real world clients and servers will be on different networks and in this way we are even considering the network latency involved in sending and receiving the requests. We have evaluated asynchronous read operation based on two metrics, one is the completion time and the other is the throughput. We have compared our results with the normal read request.The data size we read in performing normal read and asynchronous read are 512 bytes, 1KB, 1MB and 2MB. The below graph in Fig2 shows the results of these operations.

From the above graph we can observe that there is no/minute overhead involved in performing asynchronous read operation compared to normal read operation. So this result suggests that the we can perform asynchronous read instead of normal read in all the scenarios where client does not require data to continue its operations as there is no overhead.

The next metric we have evaluated is throughput achieved when using asynchronous read operation. We have observed that the clients can achieve better throughput on using asynchronous compared to normal read. The below graph shows the results of our evaluation.

As we can observe from the graph the throughput gain compared to normal read will be proportional to the size of the data we are reading. When these asynchronous read requests are batched we can even observe much higher throughput in case of asynchronous read.

6 Acknowledgements

We wish to express our sincere gratitude to Ming Chen, the sponsor of the project for his guidance and encouragement in carrying out our project work. We thank Prof. Erez Zadok for providing us with an opportunity to explore and enhance NFS as part of the course project .

References

[1] Vasily Tarasov. Asynchronous i/o explained. *www.fsl.cs.sunysb.edu/vass/linux-aio.txt*.

[2] NFS-Ganesha. Nfsv4-delegations. *https://github.com/nfs-ganesha/nfs-ganesha/wiki/NFSv4-Delegations*.

[3] Pynfs. Python nfs. *https://github.com/kofemann/pynfs/tree/master/nfs4.1*.

[4] Kendrick Smith and Andy Adamson. Linux source code. *http://lxr.free-electrons.com/source/fs/nfs*.

[5] IETF. External data representation standard. *http://tools.ietf.org/html/rfc4506.html*.