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 file systems 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 NFS client until the actual data is transferred back.

We propose Asynchronous read, where the server simply acknowledge the receipt 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 separate 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 the 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 the 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 the 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. The server fetches the data from disk, 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.

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. Pynfs is an user level implementation of NFS client and server in *python*, 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, 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 NFS-Ganesha, there are two types of threads, 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, 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 *filehandle* back to the client. The client then issues an OP_OPEN on the *filehandle*. 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 NFS ASYNCHRONOUS READ using callback mechanism (3.2).

3.1 NFS READ

Figure 1 depicts the sequence diagram 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 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

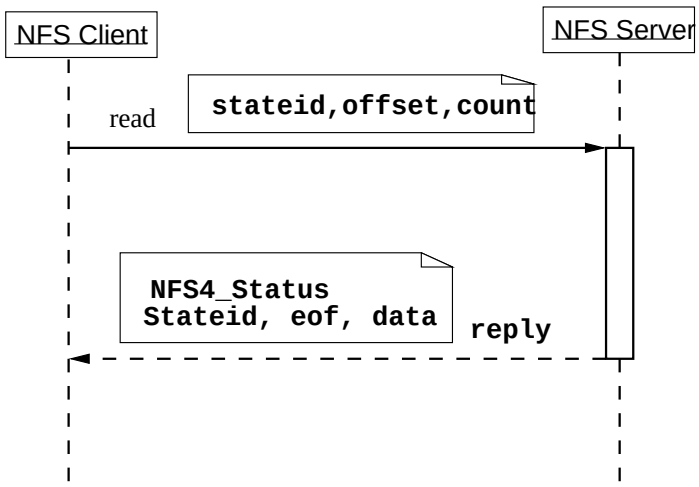

Figure 1: NFS Read Sequence Diagram

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 sequence diagram 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 new requests will dequeue this new request and process it. This involves performing all the necessary checks like the *stateid* validation 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 field 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 field 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 receive a callback in the mentioned *timeout*, the client treats the request as a failure and resends the request.

Hence forth we will call NFS READ as normal read.

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]. 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

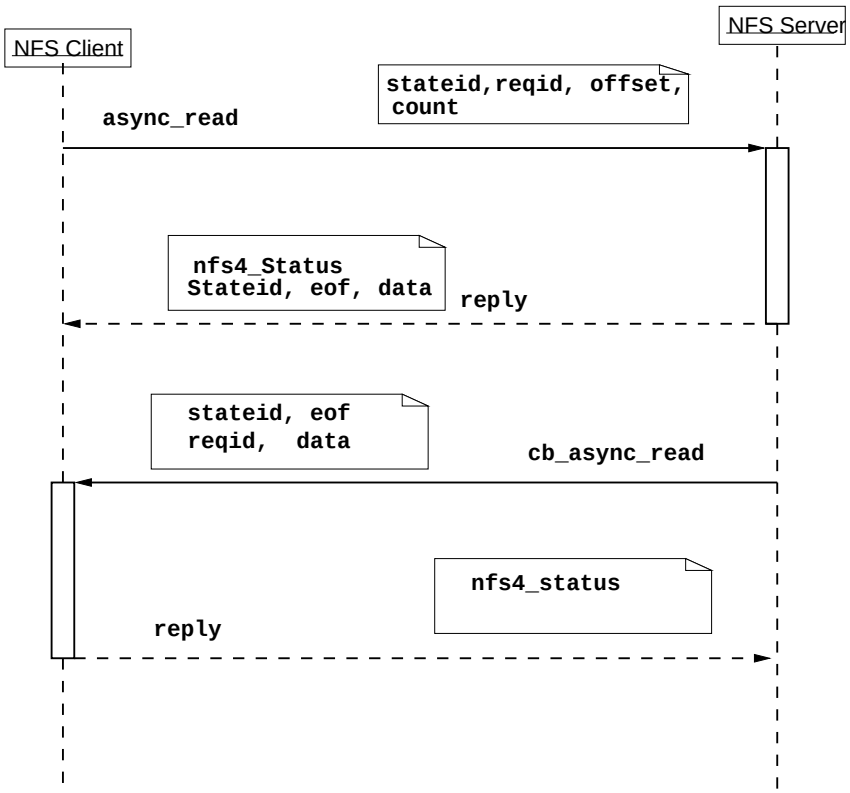

Figure 2: NFS Async Read Sequence Diagram

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 *reqid* 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* is 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: *timeout* is a configurable field in milliseconds. This field specifies the maximum time that the server can take to respond with a callback once it receives the asynchronous read request. This field is important for the client to identify server crashes and any network outages. If the client doesn't receive the response in the specified time, it treats the request as a failure and resends the request.

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

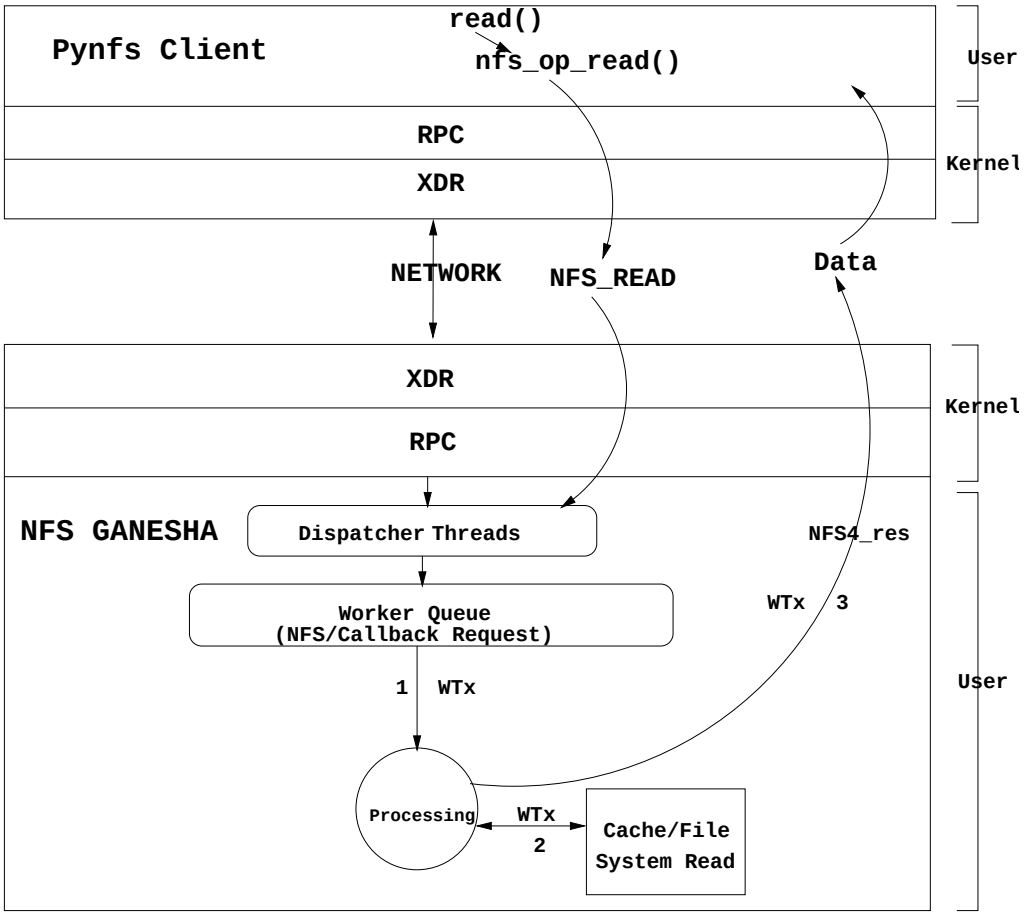

Figure 3: NFS Read Architecture Diagram

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 status 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.

We will now discuss the existing implementation of NFS_READ in NFS-Ganesha. Understanding NFS_READ is important as we have reused the design of NFS_READ

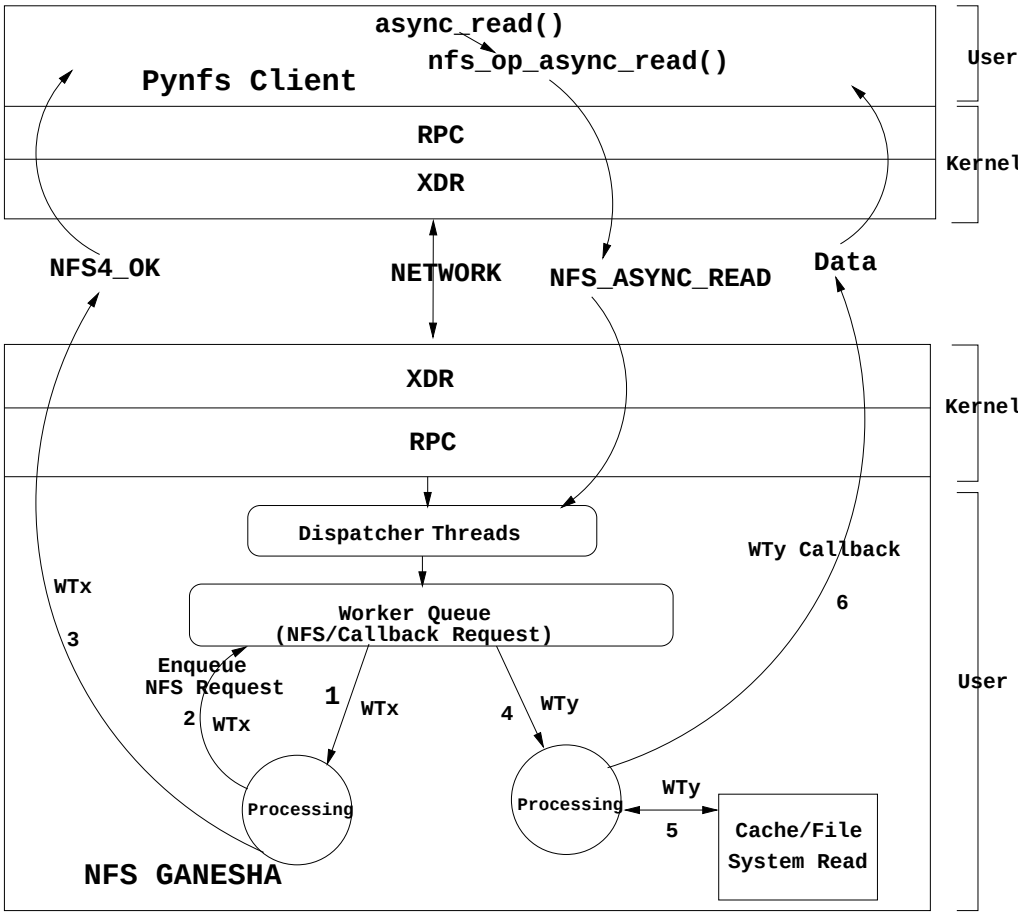

Figure 4: NFS Asynchronous Read Architecture Diagram

for performing actual read during callback in asynchronous read operation.

4.2 Implementation of NFS READ in Pynfs Client

In NFSv4.0 and above, all the NFS operation are performed using compound requests. A compound request can consist of multiple requests. The server will process the COMPOUND procedure by evaluating each of the operations within the COMPOUND procedure in order. For a normal read, the Pynfs client prepares a NFS4_COMPOUND request consisting of two operations. The first operation is the PUTFH. PUTFH is used as the first operator in an NFS REQUEST to set the context for following operations. This operation replaces the current *filehandle* with the *filehandle* of the file to be read. It is then followed by an actual READ operation. The READ operation reads data from the regular file identified by the current *filehandle*. Once the NFS clients sends the request, it waits for a configurable amount of time. If it does not receive the response in the stipulated time, an RPC timeout error will be signalled. Once the client receives the response, several validation checks are performed to identify the failures.

4.3 Implementation of NFS READ in Pynfs Server

Pynfs server on receiving the READ request, spawns a thread to process the request. The server first performs validation on the *stateid*. If mandatory file locking is ON for the file, and if the region corresponding to the data to be read from file is write locked by an owner not associated with the *stateid*, the server will return the NFS4ERR_LOCKED error. If all the tests are passed, the server then does a read on the file identified by the current *filehandle*. The file is read from the offset indicated in the request. If the read ended at the end-of-file, *eof* is returned as TRUE. Otherwise it is FALSE. The response is then encoded in XDR format and is sent back to the client.

4.4 Implementation NFS READ in NFS-Ganesha

The request processing in NFS-Ganesha is done by pthreads. Figure 3 depicts the architecture diagram of NFS_READ in NFS-Ganesha. The dispatcher thread will listen for incoming NFS REQUEST but will not decode them. It will choose the least busy worker and add the re-

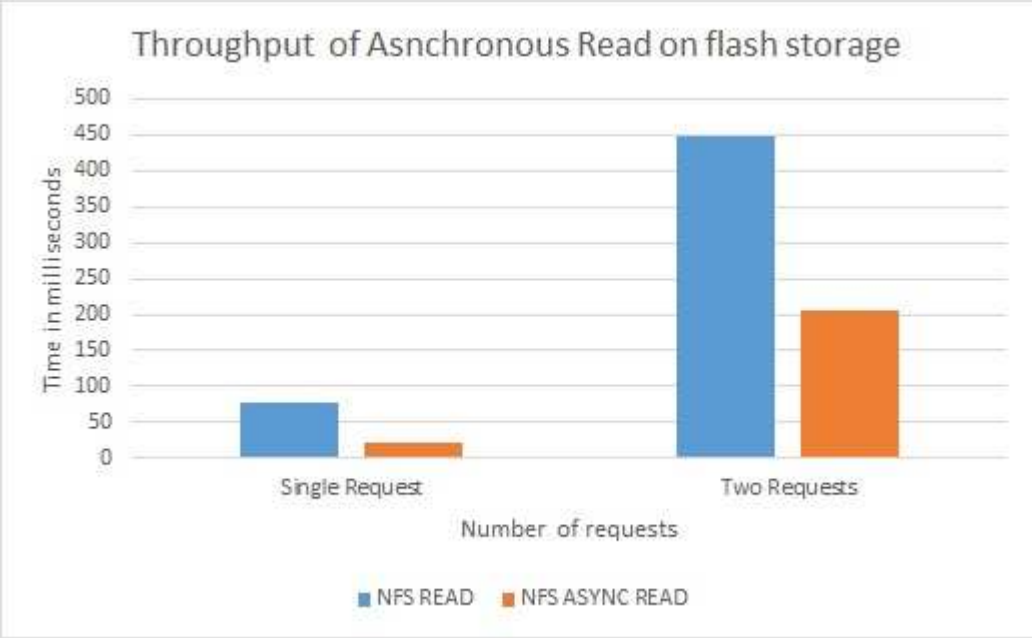

Figure 5: Throughput for NFS Asynchronous read and normal NFs read on a flash storage

quest to its lists of requests to process. Duplicate request management is also done by the dispatcher thread. One of the worker thread will then dequeue the request from the worker queue and process it. The worker threads do most of the job. These threads are very core to the NFS processing in NFS-Ganesha. The processing of a read request involves validating the client access on the requested file. On success, the worker thread will decode it and use Cache Inode API and File Content API calls to perform the read required for this request. Once the server has retrieved the data successfully from the underlying file system, it will then send the response back to the client with the requested data.

4.5 Implementation of NFS Asynchronous Read in Pynfs Client

The client side operations for asynchronous read are similar to that of a normal read. Once the request is sent, the client keeps track of them based on their *reqid* and *stateid*. If the client doesnot hear back from the server in the *timeout* specified in the request, the client times out and resends the request. The client uses both *reqid* and *stateid* to keep track of the requests that are sent to the server. Here *reqid* is essential because the same file owner might have triggered multiple request on the same file.

4.6 Implementation of NFS Asynchronous Read in Pynfs Server

NFS ASYNC READ implementation is mostly similar to that of a NFS READ operation. On receiving the request the server performs validation on the *stateid*. If mandatory file locking is ON for the file, and if the region corresponding to the data to be read from file is write locked by an owner not associated with the *stateid*, the server will return the NFS4ERR_LOCKED error. The client should try to get the appropriate read record lock via the LOCK operation before re-attempting the READ. Otherwise, if the file is free to read, the client will spawn a second thread to process the read request. The first thread that received the request will reply back with a NFS4_OK status to the client. The second thread then does a read on the file identified by the current *filehandle*. Once it fetches the required data to serve the request, it then triggers a callback request to the client with the data. The server also keeps track of the all the *reqid*s for which an NFS CALLBACK is yet to be sent.

4.7 Implementation NFS Asynchronous Read in NFS-Ganesha

Figure 4 depicts the architecture diagram of NFS_ASYNCREAD in NFS-Ganesha. The dispatcher thread will listen for incoming NFS requests but will not decode them. It will choose the least busy worker and add the request to its lists of requests to process. The worker thread then performs validation on the *stateid*. If the checks succeed, an NFS4_CALLBACK request is enqueued back into the list of requests to be processed. This involves preparing a new NFS REQUEST for the callback and then en-queuing the request into the worker queue. After en-queuing the callback, the worker thread responds with an NFS4_OK back to the client. On receiving the initial response, client is freed from the blocking call and can perform further tasks. Now on the server

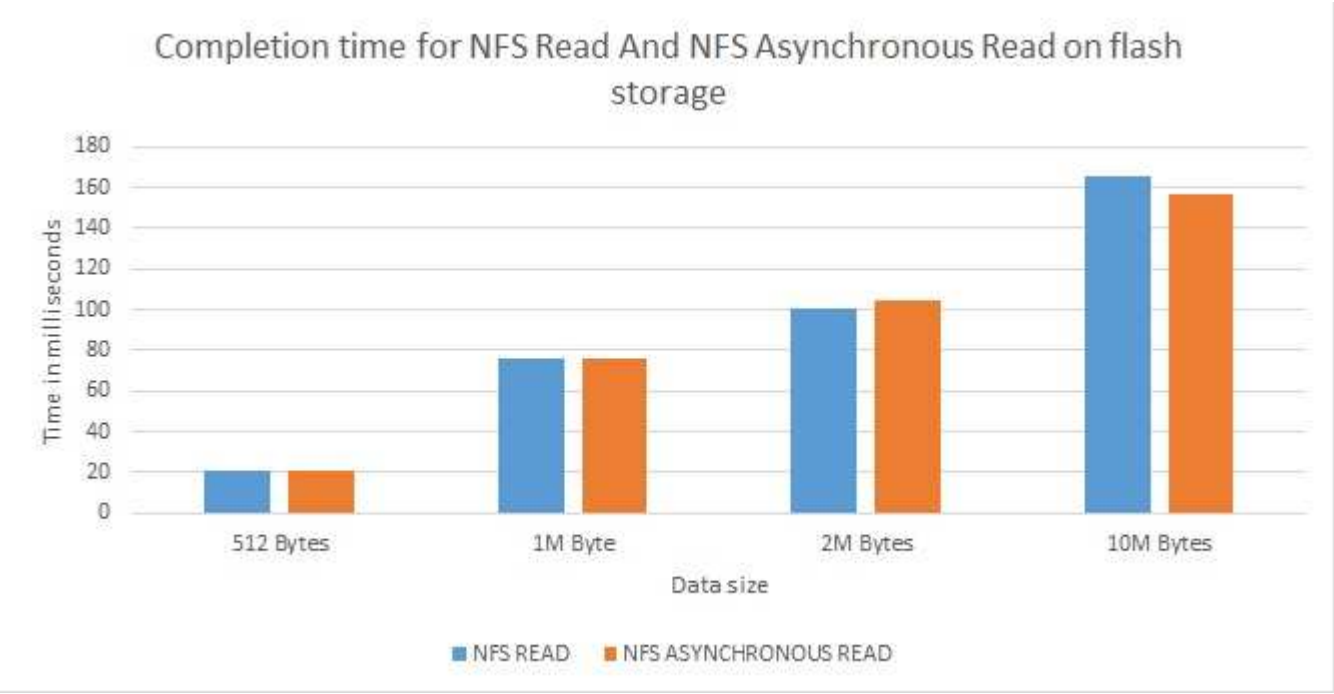

Figure 6: Completion Time for NFS Asynchronous read and normal NFs read on a flash storage

side, a least busy worker will dequeue this new request and process it. This worker thread will perform the actual file system read/cache read depending on the presence of data. After reading the data, CB_ASYNC_READ request is sent to the client via the callback channel. The worker thread which has performed the actual read dispatches the callback request to the client.

5 Evaluation

We have ran our experiments by running the NFS-Ganesha server on Amazon EC2 and Pynfs client on the local machine to replicate a real world application. 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.

Figure 6 shows the difference in the completion times between normal read and asynchronous read. we observed that the time taken for normal and asynchronous read is almost similar. This suggests that the overhead due to an additional callback incase of Asynchronous Read is not significant. Thus Asynchronous READ operation can be used in all the scenarios where client does not require data to continue its operations as there is no overhead. Also the growth in the file size had a similar effect on both NFS READ and NFS Asynchronous READ.

Figure 5 compares the time taken for the intial NFS4_OK to reach the client incase of Asynchronous READ and the total time taken by the NFS READ. Once the client recieves a NFS4_Ok incase of Asynchronous READ from the server, it is free to perform other actvities. Thus the time difference can be considered as the throughput gain for the client. We have analysed the throughput gain incase on single request and 2 consequitve requests on a file size of 1MB. The offsets in the file were choosen carefully in a way that neither the dcache nor READ ahead will effect the final outcome. In case of a single request it took 78 millliseconds for the NFS READ to get completed. And it took around 21 milliseconds incase of Asynchronous READ for the initial NFS4_Ok to reach the client.

Figure 7 shows an intersting observation. When we have made two consecutive reads using NFS READ on a file of size 1GB. While the first read has a offset of 1MB, the second read has a offset of 512 KB. The size of the data read is 512KB. The file offsets are choosen this way to prevent READ AHEAD caching. While it has taken 88 milliseconds in NFS READ for both the requests to get completed, it has taken only 68 milliseconds for NFS Asynchronous READ. This can attributed to the scenario depicted in Figure 8. Incase of NFS READ, second read request is made only after the client recieves the reply to the first request. Thus there is no chance for both of them to be scheduled together to the disk. But in the case of Asynchronous READ, Since the second request is triggered immediately after receiving the NFS4_OK to the first one, the two requests can be scheduled together to the disk. Thus there is a time gain since the seek head does not have to do a full rotation to reach the second position as both the experiments are scheduled together.

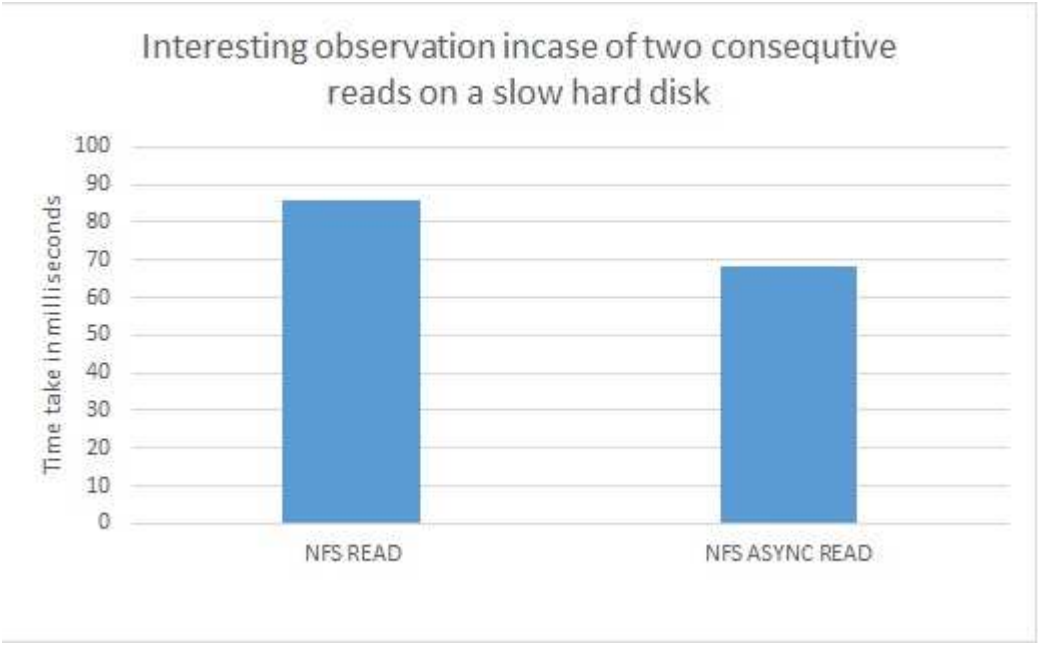

Figure 7: Intersting observation incase of two consequtive requests on a hard disk

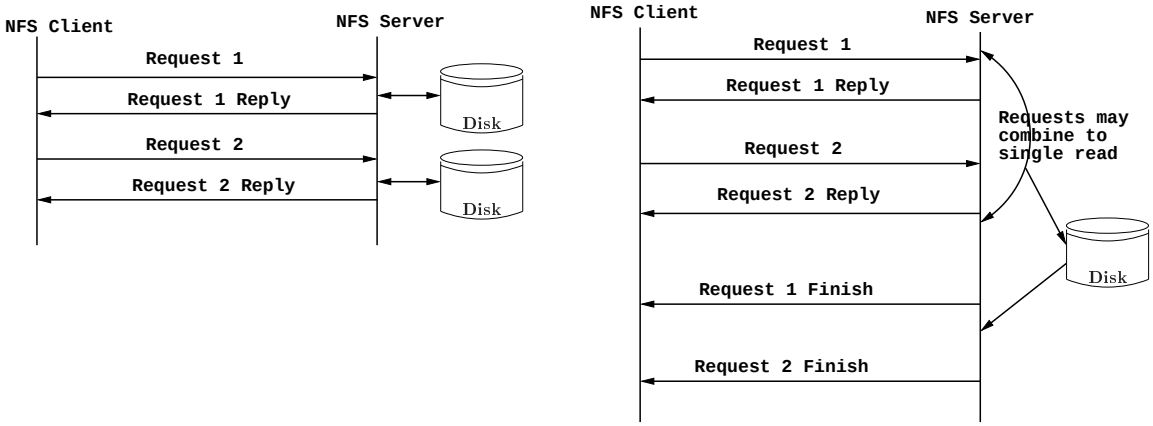

Figure 8: Throughput for NFS Asynchronous read and normal NFs read on a flash storage

6 Discussion

The NFS server on receiving the asynchronous read operation, the worker thread has to reply to the client with NFS4_OK indicating that it has received the request. The server will then perform the actual read and reply to the client asynchronously using callbacks. The worker thread which receives the request of asynchronous read creates a new thread for performing this actual read operation and then sending reply to the client using callbacks. This implementation is not scalable because as the number of requests increases to some thousands simultaneously we will end up creating thousand odd threads. On a system with average RAM this can lead to a memory crash. We have then decided to use NFS-Ganesha's request flow for performing the actual read operation and sending the data using callback to the client. NFS-Ganesha creates a fixed number of pool of threads during the start of the server and this can be configurable by the user. The threads are classified as worker threads and dispatcher threads. Worker threads are responsible for processing the request and as these threads are in fixed number, this design does not suffer from memory crash problem. This design is also scalable as we can increase the number of threads to run in order to handle more number of requests simultaneously.

7 Conclusions

We proposed a novel approach for asynchronous read operation in NFSv4.1 using callbacks. We observed that current asynchronous I/O operations on an NFS mounted device are triggered sequentially. This blocks the NFS client until the actual data is transferred back to the server. We thus implemented Asynchronous READ operation to improve the throughput of the client by freeing it

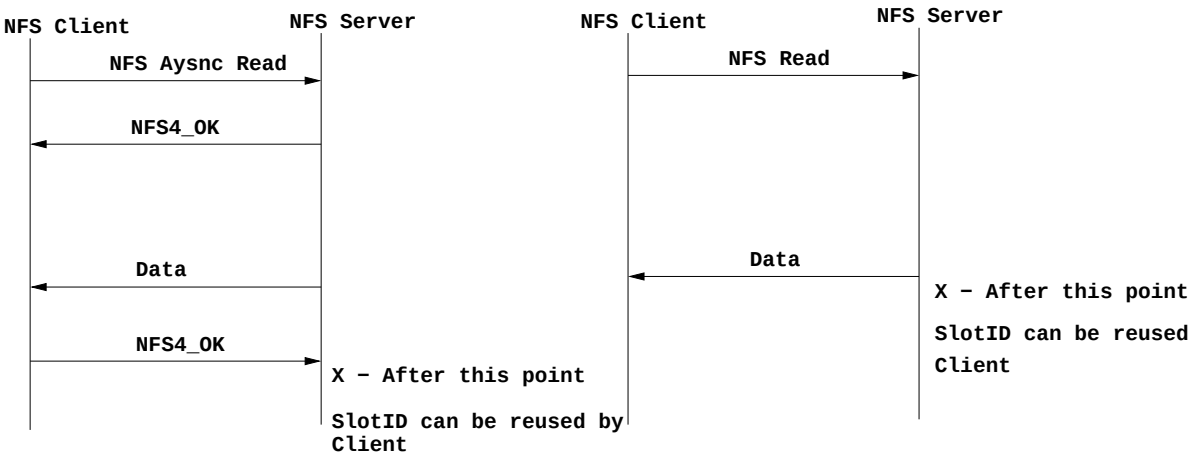

Figure 9: NFS Asynchronous Read Slot Reuse Diagram

immediately after recieving a NFS4_OK from the server. We found out that the overhead caused by the additional callback to the client is minimal and thus performance degradation is minor even in the case of single read. We identified that there is a performance gain on performing asynchronous reads on same file mainly at consecutive offsets. This is because of the decreased seek time on the server as mentioned in section 5. We also noted that there is a significant throughput gain depending on the size of the data.

These findings are not specific to our hardware setup. All the observations presented above are agonastic to the network speeds.

8 Future Work

8.1 Slotid reuse in Asynchronous Read

NFSv4.1 supports exactly once semantics model. This means that the request is processed by the NFS server once and only once. This also helps the server in identifying the duplicate requests. NFS server does this identification with the help of SLOTID and SEQID that will be present for each request. If another request uses the same SLOTID and SEQID which the server has already seen in the session then that is a duplicate request and then searches in the reply cache to check if it has already processed. If so, it replies the same reply to the client else it will reply current status of the previous request. In normal NFS4_COMPUND operations the client generally reuses the SLOTID immediately after receiving the response from the server. But in case of asynchronous read the client should not reuse the SLOTID after receiving the initial NFS4_OK, because the client has not received the data yet and thereby the request is not completed. In NFS terms, the requester still has an outstanding request on that SLOTID, hence it cannot be reused and Figure 9 depicts this scenario. We need to modify the implementation reusing of SLOTID in case of asynchronous read. This idea can be depicted by the Figure 9. If the slot is reused before receiving the callback then this causes server to delete the previous entry from the reply cache. In the mean time if the client sends the same request in case of a failure then the server will process the request again which will violate exactly one semantics model.

8.2 Other Asynchronous operations

In this paper we have mentioned designed and implemented asynchronous read operation. Similarly we can implement asynchronous write operation in NFS. As write operation is not completely synchronous as read operation, hence we feel we may not observe similar benefits to asynchronous read. But nevertheless we may observe some interesting results in case of simultaneous writes.

9 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*.