A Survey on SEDA and Using SEDA to Optimize HDFS Read Operation

Isuru Suriarachchi, School of Informatics and Computing, Indiana University

***Abstract*—Handling high concurrency is a critical issue in almost all internet services. A web server is a perfect example where a large amount of concurrent users are served by managing limited resources. There has been number of different architectures proposed to optimize the resource usage and maximize the throughput in such applications. Thread-per-request model and event-driven model are two heavily used such architectures. In 2001, Welsh et al. [1] pointed out the performance issues in those two architectures under high concurrency and proposed Staged Event Driven Architecture (SEDA) to address the issues. SEDA uses a hybrid of thread pools and eventing concept to utilize the advantages of both techniques. Since the introduction, there have been number of applications of SEDA in highly concurrent systems. In this paper, first we present few most interesting applications of SEDA selected through a survey. Then we discuss SOR-HDFS [2] which is an application of SEDA to improve HDFS *Write* operation. There we try to highlight the shortcomings and restrictions as well in SOR-HDFS design. Finally we present a design to use SEDA for HDFS *Read* operation and discuss how it improves read performance.**

Keywords*—SEDA, Concurrency, Throughput, SOR-HDFS, HDFS Write, HDFS Read.

I. Introduction

With the rapidly increasing usage of internet based services, millions of user requests hit the service hosting servers every minute. For example, the incoming traffic for e-commerce sites like Amazon and eBay on a thanksgiving day is enormous. It is a huge challenge for a internet service to handle this large amount of requests while preserving the must have qualities like high availability, low latency and robustness. This problem cannot be addressed only by increasing the amount of compute resources. Therefore, researchers have come up with different architectures for highly concurrent applications to optimize the resource utilization and increase the throughput.

One of the most common techniques used in internet services is the thread-per-request model associated with a bounded thread pool. This design uses a fixed size thread pool which is initialized on application start-up. Each incoming request gets a thread from the thread pool and keeps the thread for the entire duration of the request. Assigned thread is released back to the thread pool when the request is served. Number of popular Web Servers and Application Servers like Apache HTTP server [3], IIS [4], Apache Tomcat [5] and IBM WebSphere [6] have been using this design for a long time. However, the limitation of this design is that it starts to drop connections when all the available threads are busy. This can happen under heavy loads. Therefore, requests tend to queue up in the network and client applications might have to retry to establish a connection.

Event-driven model is another approach for handling high concurrency. Most of the time, this model uses a limited number of threads that loop continuously. And there is an event queue which contains events (processing requests) submitted by the sub components of the system. Continuous threads pick the events from the queue and process one after the other. This model does not drop excess requests like the thread pool model. All requests are queued as events and processed when they reach the top of the queue. This model is used by Web Servers like thttpd [7] and Flash [8]. Event-driven model generally shows better robustness due to usage of event queues. However, it provides less throughput and blocked event handling threads can cause severe performance issues.

In 2001, Welsh et al. [1] pointed out the limitations and performance issues in above two architectures under high concurrency and proposed Staged Event Driven Architecture (SEDA) to address the issues. SEDA combines the power of threading model with the robustness provided by event queues. In SEDA architecture, an application is decomposed into number of *stages* which are connected by *event* queues. A stage has its own thread pool and each thread picks up a set of events from the input queue and writes the output as events to one or more output queues after processing. In addition to that, each stage has a *dynamic resource controller* which can adjust the resources allocated for the stage according to load at a given time.

Since the introduction, there has been number of applications which use the SEDA architecture both in industry and research prototypes. Apache Cassandra [9] [10] is a NoSQL store which uses SEDA for its message processing pipeline. Being an open source project, Cassandra is used by number of major projects to handle large volumes of data including big companies like Twitter and Netflix. Cassandra is one of the best examples which proves the ability of SEDA to handle high concurrency levels. Amazon Dynamo [11] is another key-value store where high availability is considered a must with massive incoming loads. Mule Enterprise Service Bus [12] is also an open source project which uses SEDA in its internal message processing architecture. Apache Mina [13] is an open source network application framework which uses SEDA. In addition to those, there are plenty of research publications which uses SEDA to implement prototypes. Two good examples are Tapestry [14] peer-to-peer routing infrastructure and BGPMon [15]. We will be discussing these applications of SEDA in more detail in section III.

SOR-HDFS [2] is a recent application of SEDA to improve HDFS *Write* operation. When executing the *Write* operation,

a data block is transferred as packets from clients to HDFS DataNodes. DataNodes accept the packets and use a single thread for each block to persist data. This is called One-Block-One-Thread (OBOT) architecture. SOR-HDFS applies SEDA to improve performance of *Write* operation by identifying major stages in the *Write* process and separating them. Later we try to evaluate and discuss this design and identify the shortcomings.

As the main contribution of this paper, we present a SEDA-based architecture for HDFS *Read* operation. We looked into all other HDFS block operations like *Replace*, *Copy* and *Transfer*. But the frequency of those operations compared to *Write* and *Read* operations is very low. And also, SEDA is designed to improve performance under high loads. Therefore, applying SEDA for less frequent operations will over complicate the system without showing a considerable improvement. As pointed out in SOR-HDFS, default HDFS DataNode uses the OBOT architecture for all operations. Therefore the entire *Read* process is carried out using a single thread. Each request to read a block goes through number of steps. Access verification, waiting for ongoing writes, reading block metadata, sending status responses, creating packets by reading the disk and writing packets to output socket are the main steps in the default HDFS *Read* operation. In our new design, we separate these steps into different SEDA stages to optimize read performance. We identify the steps in which the threads consume more time and carefully design the sages to minimize the back pressure under high loads.

Remainder of the paper is organized as follows. In section II we discuss SEDA and SOR-HDFS in more details to set the background. Then we present the results of our survey on SEDA and go through 5 of the most interesting applications of SEDA in section III. Then in section IV we present our thoughts on SOR-HDFS and discuss the limitations and restrictions of the design. Finally in section V we present our SEDA-based design to improve performance of HDFS *Read* operation and conclude in section VI.

II. Background

A. Staged Event-Driven Architecture

As pointed out earlier, thread-per-request architecture with bounded thread pools works well for average loads. But it starts to drop connections and show very high response times under heavy loads. Event-driven model is robust, but can show very low throughput when worker threads block. Staged Event-Driven Architecture (SEDA) has been designed to address these issues by combining the powerful features of both these models. SEDA splits an application into a chain of *stages* which are connected by *event queues*. SEDA uses *dynamic resource controllers* associated with each stage to dynamically control the resource allocation depending on the load. There are four very important properties of SEDA.

- **Support massive concurrency:** SEDA supports massive concurrency by incorporating event queues to avoid performance degradations due to busy threads. This combination of events and threads makes sure that the system is robust under high loads.

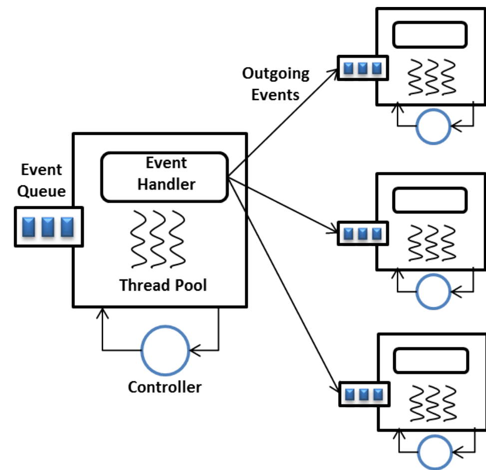

Fig. 1. A stage in SEDA

- **Support load conditioning:** Load conditioning is the property which makes sure that the throughput of the application is kept constant even under massive loads. When the application reaches saturation, response time increases due to queuing delay. But the increase of response time is linear with the number of requests. This makes sure that the requests are not rejected and queued even under heavy loads and eventually severed with only a linear increase of response time with load.
- **Support self-tuning resource management:** SEDA is capable of dynamically adjusting the resource allocation for each stage using resource controllers. For example, the size of the thread pool associated with a particular stage is adjusted dynamically by looking at the number of events in the incoming queue.
- **Ease of engineering:** An application developer does not have to worry about the concurrency when developing on top of a SEDA enabled framework (like Sandstorm presented in [1]). Developer just has to carefully identify the stages of the application and implement the stages by using the interfaces provided by the framework.

Figure 1 (adapted from [1]) shows the components of a SEDA stage which is the basic processing unit of the architecture. A stage consists of an incoming event queue, an event handler, a thread pool and a controller. A thread in the thread pool can pull a batch of events from the incoming event queue and process the batch by invoking the event handler. Event handler contains the business logic related to the stage. Upon completion, the results can be written to one or more event queues for further processing by other stages. Controller is responsible for dynamically tuning parameters like the size of the thread pool and number of events included in a single batch.

A complete application is built by connecting number of stages using event queues. SEDA provides the flexibility of executing some stages in parallel as well. And also, at times the same thread pool can be shared across more than one stage to improve resource utilization in some applications.

B. SOR-HDFS

Islam et al. [2] present SOR-HDFS to improve performance of the HDFS *Write* operation. In their earlier work, they propose RDMA-enhanced HDFS [16] [17] which is a solution for communication overheads in default HDFS using pipelined and parallel replication schemes. In SOR-HDFS, they try to further improve the HDFS write performance by proposing a SEDA-based architecture to replace the One-Block-One-Thread (OBOT) architecture in default HDFS. To evaluate both optimizations together, they have implemented the SEDA based architecture on top of RDMA-enhanced HDFS.

In default HDFS *Write* operation, data blocks are split into packets and transferred from clients to DataNodes. When a DataNode receives a block as a series of packets, it uses a single thread to write all the packets in the block. This can cause congestion when all threads become busy due to high *Write* traffic. SOR-HDFS applies SEDA to solve this issue by splitting the *Write* process into stages. They identify *Read*, *Packet Processing*, *Replication* and *I/O* as the four stages in the process. Each stage is handled by a dedicated thread pool. Therefore, the *Write* process overlaps at packet level too in addition to block level.

Read stage consists of a receiver which supports multiple endpoints. Different clients can connect to different endpoints. There is a pool of buffers associated with the *Read* stage and it is used to write data received for endpoints. When data arrives at an endpoint, it is written to a free buffer in the pool. Then the receiver thread returns the buffer pointer of each packet to the Process Request Queue in *Packet Processing* stage. The Process Request Controller (PRC) in *Packet Processing* stage is responsible of assigning a thread from thread pool to each packet. When a thread picks the header packet of a block, PRC makes sure that all subsequent packets of the same block are assigned to the same thread. This preserves the packet order in a block. Finally the *Packet Processing* stage aggregates packets by blocks into an aggregation cache. After processing, packets are sent to the Replication Request Queue. *Replication* stage threads pick packets from the queue and replicates for multiple endpoints. Finally the *I/O* stage threads pick packets from I/O Request queue and flush the aggregated data from the cache to the disk.

III. SURVEY ON SEDA

We conducted a survey on the applications of SEDA both in research works and in well known software projects. Here we present five of the most interesting applications and explain how SEDA is being used in each of those. Other than those five, Amazon Dynamo [11] is also an interesting application of SEDA which handles high concurrency. However, we are unable to find details on Dynamo architecture as it is a proprietary software.

A. Apache Cassandra

Apache Cassandra [10] is a NoSQL distributed database for managing very large amounts of data. Cassandra guarantees scalability and high availability without compromising performance. It is specifically designed as a fault tolerant framework which can be run on commodity hardware or on cloud resources. Cassandra was initiated by Facebook and number of companies like Twitter and Netflix currently use it to handle large amounts of data.

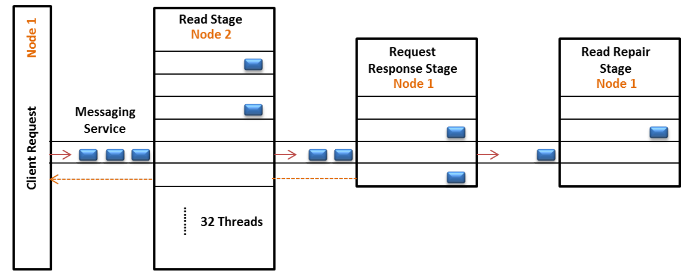

Fig. 2. SEDA for reads in Cassandra

Cassandra organizes data for a single application in a keyspace similar to the schema in relational databases. A keyspace can contain multiple column families. In Cassandra, total data managed by the cluster is represented by a ring. All nodes in the cluster participate in the ring and each node stores a range of row keys. Depending on the replication factor of the keyspace, there can be multiple replicas of same data stored in different nodes. To find the node where the first replica of a row is stored, the ring is traversed clockwise till it locates the node which contains the correct row key range.

There are number of tasks running inside each node in a Cassandra cluster. Those tasks include handling read requests, handling write requests, replication, flushing cached data into disk, communicating with neighbours etc. Cassandra uses SEDA to manage these tasks by defining stages and incorporating task queues to connect them. Some examples of the stages running in a Cassandra node are *ReadStage*, *RequestResponseStage*, *MutationStage*, *ReadRepairStage*, *GossipStage*, *Migration Stage* and *FlushWriter*. A complete list of Cassandra stages can be found in [18]. There are thread pools associated with each of these stages. Status of these thread pools can be checked by running the *"nodetool tpstats"* command on a Cassandra node.

In Cassandra, all nodes in the cluster are treated equally and there is no concept of a master node. Therefore, a client request can hit any node in the cluster. Figure 2 (adapted from [18]) shows how SEDA based implementation operates for a read request. This assumes that the request is received by Node 1 in the diagram and data is stored in Node 2. After receiving the request, Node 1 enqueues a task on the *ReadStage* in Node 2 with a reference to a callback in Node 1. Node 2 *ReadStage* picks up the task from the queue when one of the threads in its thread pool becomes free. After reading the row in Node 2, *ReadStage* enqueues the result as a task for *RequestResponseStage* in Node 1 which returns the result of the request to the client through the callback. Finally the *RequestResponseStage* in Node 1 might trigger an asynchronous task for the *ReadRepairStage* which makes sure that the caches of all replicas for the given row key are written to the disk.

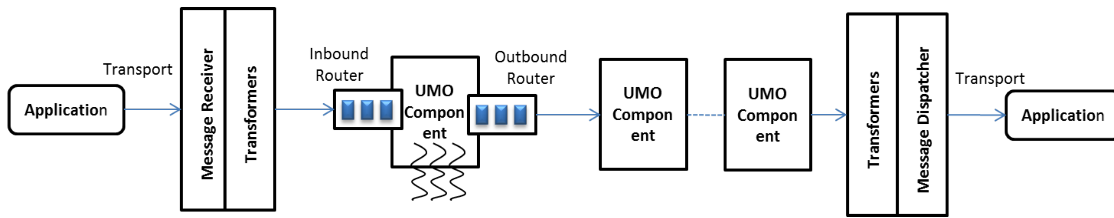

Fig. 3. SEDA based component architecture in Mule ESB

B. Mule ESB

Mule Enterprise Service Bus (ESB) is a well known open source ESB which is used for various kinds of SOA integrations. Mule ESB is used by number of large companies like Yahoo, Amazon, at&t and Verizon to handle very high levels of traffic. Some of its heavily used features are Service Mediation, Service Orchestration, Message Routing and Data Transformation. The basic processing module in Mule ESB is called Universal Messaging Object (UMO) component. Mule ESB comes with a set of default components which can be used to build service mediation chains. Some examples are *BridgeComponent*, *LogComponent*, *PassThroughComponent* and *RestServiceWrapper*. In addition that, users can implement their own business logic by writing a new component.

In Mule ESB, the components are executed using a SEDA-based architecture. Figure 3 shows how a message flows through the ESB in a simple scenario where few components are used for message mediation. Each component represents a stage in SEDA and has its own thread pool. Thread pool size for each component can be configured using the Mule configuration. However, there is no dynamic resource controller used in Mule ESB to adjust the thread pool size in runtime. Inbound router (also called the endpoint) acts as the input event queue. Business logic in each component represents the Event Handler in SEDA. Whenever a thread in a component becomes free, it picks up a message from inbound router and processes it. Outbound router forwards the processed message to the inbound router of the next component.

C. Apache Mina

Apache Mina [13] is a framework for developing high performance and highly scalable network applications. Mina framework provides an even-driven asynchronous API over number of transports such as TCP and UDP via Java NIO. Mina uses SEDA architecture to achieve high throughputs using even-driven model. Mina has been used for building various network applications like HTTP servers/clients, FTP servers/clients and SSH servers/clients. Few example sub projects under Mina are Apache FtpServer [19] and Apache SSHD [20].

Figure 4 shows the basic architecture of Apache Mina. The I/O Service listens on the network layer for incoming packets. When a packet arrives, it just pushes the packet into a queue and returns. I/O Filter chain is the main extension point in Mina. Application developers can easily implement filters to perform any processing steps on incoming packets using Mina API. The filter chain has been developed using SEDA such that each filter becomes a stage. Mina framework takes care of creating message queues between filters. A dedicated thread pool is associated with each filter and thread pool parameters can be configured programmatically using Mina Filter API. After going through the filter chain, a message reaches the I/O Handler which contains the actual business logic of the application.

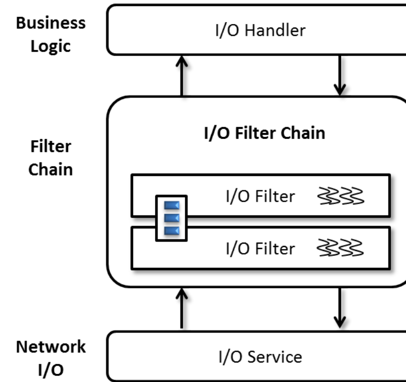

Fig. 4. Apache Mina Architecture

D. Tapestry

Tapestry [14] is a scalable peer-to-peer overlay routing infrastructure which is built on TCP/IP. It is an implementation of the Decentralized Object Location and Routing (DOLR) API [21]. Tapestry can be seen as a resource virtualization layer which routes messages to endpoints such as nodes or object replicas. Endpoints are identified by *globally unique identifiers* (GUIDs) rather than IP addresses. Therefore, any message routed by Tapestry is addressed by a GUID. Tapestry internally routes the message to the physical host which contains the resource identified by that GUID. Because of this virtualization, the application which uses Tapestry does not have to worry about the locations of resources. There are many network applications which uses Tapestry. OceanStore [22] persistent data store is a good example.

Tapestry uses SEDA to integrate components in a single node targeting high throughput and scalability. Figure 5 (adapted from [14]) shows the SEDA-based node architecture in Tapestry. There are five main components (SEDA stages) named *Core Router*, *Node Membership*, *Mesh Repair*, *Patchwork* and *Network Stage*. These internal components

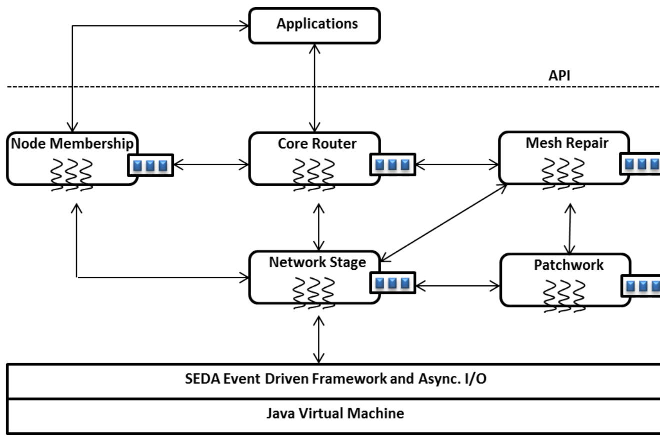

Fig. 5. Internal SEDA-based architecture of a Tapestry node

communicate via events using a subscription model and each component has an associated thread pool. We do not go into details of the functionalities of these components as that is beyond the scope of this paper.

E. BGPMon

BGPMon [15] is a Border Gateway Protocol (BGP) monitoring tool which provides scalable real time monitoring. BGPMon can connect to large number of peers (mostly routers) and clients at the same time. Clients can subscribe to BGPMon system to receive routing information. All routing events received from peers are serialized into a single XML stream and sent to all subscribed clients.

BGPMon uses an architecture similar to SEDA for the routing event pipeline. Figure 6 (adapted from [15]) shows the BGPMon architecture. Two main stages in the system are Labelling stage and XML conversion stage. Each *peer thread* in BGPMon connects to a router which is being monitored. Peer threads enqueue all BGP messages received from routers into the *peer queue*. Then the messages are processed by the label thread and the XML thread. Finally the routing results are serialized as an XML and written to the XML queue. One difference of this architecture when compared to SEDA is that the three queues used in BGPMon have fixed length. That can cause congestion when the routers send large spikes of routing messages. BGPMon removes a message from XML queue only after all clients have read it. Therefore, if the incoming message speed or writing speed is larger than the speed of slowest reader, system becomes saturated. To overcome this issue, BGPMon uses two techniques. First one is called *pacing writers* where the write speed is controlled by the system when the queue length exceeds some threshold. Other one is called *dropping slow readers* where the system automatically drops the slowest readers to avoid congestion.

IV. Limitations of SOR-HDFS

The evaluations done in [2] using well-known Hadoop benchmarks and HBase tests shows that SOR-HDFS performs better compared to default HDFS and RDMA-based HDFS.

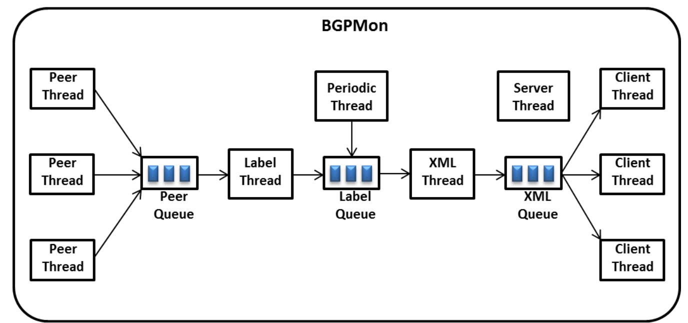

Fig. 6. SEDA-based BGPMon Architecture

However there are few limitations in the design which can lead to some issues under different scenarios compared to the test clusters used to evaluate SOR-HDFS.

As we discussed earlier, SEDA is designed to maximize throughput under high loads by making sure all requests are queued and eventually served by different stages. SOR-HDFS shows better throughput due to this reason. And also it might reduce the latency under high loads compared to default HDFS by avoiding congestion. That is because a congested OBOT model can drastically increase the latency under high loads. However, SOR-HDFS should show higher latency under low and average loads compared to OBOT model. In other words, non-saturated OBOT model should show lower latency compared to SOR-HDFS. That is because, in SEDA, a packet is handled by different threads in different stages. That adds context switch delay in each stage. And also each packet is queued between stages. That adds a considerable queuing delay as well.

SOR-HDFS does not implement *dynamic resource controllers* proposed in SEDA to control the resource allocation in runtime. Therefore, the sizes of the thread pools associated with each stage are constant during the entire runtime. And also, the thread pool sizes for different stages can depend on hardware configurations of the cluster. For example, number of replicator threads depend on the NIC bandwidth and number of I/O threads depend on whether HDDs or SSDs are used. Therefore, users always have to tune the thread pool sizes according to their hardware while deploying the cluster. It might need some additional tests as well to decide proper thread pool sizes. In SOR-HDFS evaluation, they explain how they conducted such tests. This is an extra burden which is not there in default HDFS and false thread pool sizes can give very bad throughputs. One other problem with this is that there can be many network failures and node failures within an HDFS cluster in runtime as HDFS is designed to run on commodity hardware. Therefore, initially tuned thread pool sizes can become non-optimal in runtime after such hardware failures. Implementing a dynamic resource controller to adjust thread pool sizes and other parameters according to inter node communication details is an ideal solution for this kind of issues.

In packet processing stage and in I/O stage of SOR-HDFS, if some thread picks the header packet of some block, all the

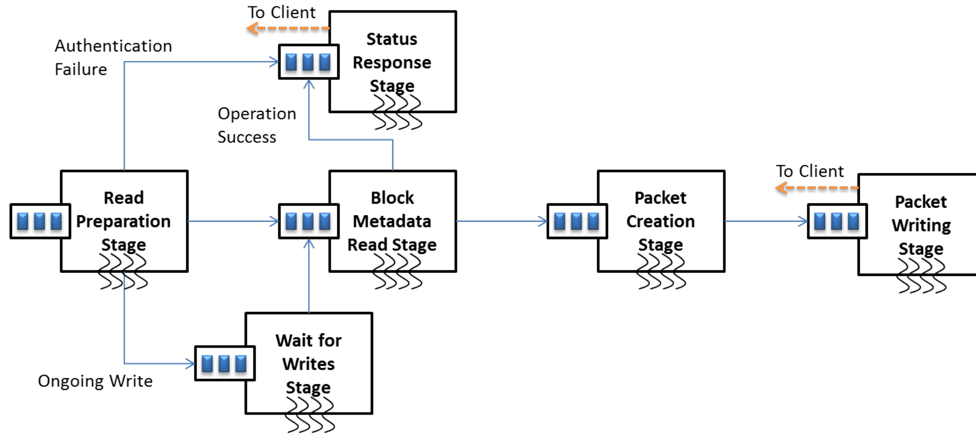

Fig. 7. SEDA-based architecture for HDFS *Read* operation

subsequent packets of the same block have to be picked by the same thread. This is because the packet order within the block must be preserved. This reduces the power of completely overlapped stages proposed in SEDA.

SOR-HDFS is implemented on top of RDMA-based HDFS [17]. It takes the advantage of RDMA connection objects which support multiple endpoints in *Read* stage. It will be interesting to see how SEDA works on default HDFS which does not have the option of multiple endpoints per connection. And also, in SOR-HDFS test clusters they have always set the *Read* stage receiver thread count to one because the RDMA connection creation is more expensive than Socket creation in default HDFS. This might become a bottleneck in SOR-HDFS as only single thread is used to receive packets in *Read* stage.

V. SEDA FOR HDFS READ OPERATION

Here we present a SEDA-based architecture to improve HDFS *Read* operation performance under high loads. Our design is based on the latest HDFS code base [23] which was released under Apache Hadoop [24] version 2.5.0.

Like the *Write* operation, default HDFS *Read* operation also is implemented using OBOT architecture. When some HDFS client or a peer node executes a *Read* command for a particular block on some DataNode, a thread from the DataNode server thread pool picks up the request and performs the entire *Read* operation. First it reads the HDFS protocol *Read* request header to extract all needed parameters like client name, block Id, block token, block offset etc. Then it performs an access check to authenticate the client using the block token. If there is some ongoing write operation for the same block, the reading thread waits till minimum read length is written. Then it reads block metadata to set up input streams for block checksum and block data. Finally the same thread loops through the data range and creates packets by reading the disk and writes packets to the output stream.

There are few steps in this read process during which the read thread goes through severe delays. Best example is the wait for ongoing writes where the thread may sleep up to three seconds. And also steps like writing status messages to output socket, reading data from disk and writing packets to output socket can experience considerable delays due to I/O operations. Such waited threads limit the number of available threads in the DataNode server thread pool. That can lead to connection drops under high loads and very low throughputs due to saturation. In our SEDA based architecture, we try to improve the read throughput under high loads by reducing the back pressure. We carefully design stages to make sure that the steps with higher delays are performed by separate stages to improve the responsiveness of other stages.

Figure 7 shows our new SEDA-based architecture for HDFS *Read* operation. After a new connection is accepted by a thread in the DataNode server thread pool, it reads the command sent by the client. If the command is a *Read*, it enqueues a task in the *Read Preparation Stage* input queue with pointers to input and output streams. Server thread is immediately returned back to the thread pool and this makes the server lot more responsive under high loads. As shown in Figure 7, there are six stages in our SEDA-based architecture. Each stage picks up tasks from its input queue and may enqueue tasks to one or more other stages. A dedicated thread pool is associated with each stage with a dynamic resource controller module to control the thread pool size depending on the queue length.

A. *Read Preparation Stage*

When a task is received through the input queue, first of all the HDFS protocol message is processed by the *Read Preparation Stage* to extract the parameters sent by the client. Parameters like *block token* and *client name* are found in message header and other parameters like *offset* and *read length* are found in message body. Then an authentication step is performed using the block token sent by the client. If the authentication fails, read operation is aborted and an error message is enqueued to *Status Response Stage* with a pointer to the output stream. If the client is authenticated to execute the command, *Read Preparation Stage* thread continues and checks whether there is an ongoing *Write* operation which writes to the same block to be read. In that case, if the requested read length is not yet available to be read, a task is

enqueued in the *Wait for Writes Stage* as the *Read* operation must be delayed. Otherwise, a task is enqueued in the *Block Metadata Read Stage* and the thread is returned.

B. Status Response Stage

The *Status Response Stage* is responsible for sending intermediate response messages to the client. Authentication failure messages and read success messages are examples of such responses. A task coming into the input queue for this stage consists of the message to be sent and a pointer to the socket output stream. Some free thread in the thread pool picks up the message and writes it into the output stream. As this process involves socket I/O, having a dedicated stage avoids possible delays in threads associated with main stages.

C. Wait for Writes Stage

In some cases, the block to be read is still being written by some other process. HDFS allows a block to be read while it is being written. However, if the length of bytes requested to be read by the client is still not available on the disk, reader waits up to three seconds to have enough data on the disk. The *Wait for Writes Stage* is introduced to avoid delays in main stages. If such scenario is found where the reader has to wait, the *Read Preparation Stage* sends a task to the *Wait for Writes Stage*. A thread in *Wait for Writes Stage* picks up the task and waits in a loop up to three seconds by periodically checking whether the block to be read has enough bytes on the disk. If it finds enough bytes or time expires, the thread exits the loop and enqueues a task in the *Block Metadata Read Stage* to continue the read process.

D. Block Metadata Read Stage

The *Block Metadata Read Stage* is responsible for reading metadata associated with the block being read. When a thread in this stage picks up a task, first it reads the metadata stream and verifies the checksum. Sometimes the user requests the block checksum to be sent with data packets. In that case, this stage verifies the checksum and prepares it to be read when the packets are created. Then it calculates the start offset and end offset of the data block to be read according to input parameters. If all client parameters are properly verified, a read operation success message is sent to *Status Response Stage*. Finally a task is enqueued in the input queue of the *Packet Creation Stage*.

E. Packet Creation Stage

An HDFS block can consist of number of packets depending on block size used in the deployment. The *Packet Creation Stage* is responsible of creating packets by reading bytes from start offset to end offset of the block. This stage should be configured with a relatively higher number of threads under high loads as it involves disk I/O. The associated resource controller increases the thread count by looking at the queue length in such situations. In addition to that, the *Packet Creation Stage* consists of a pool of buffers. When a new packet is created, a free buffer is selected from the pool and the packet is written to it. Each packet consists of header, checksum and data sections which are written in order into the buffer. Once a packet is completely written to the buffer, a task is enqueued to the *Packet Writing Stage* with a pointer to the buffer and the output stream. This process is continued in a loop until all packets in the block are created. It is important to note that this stage splits each incoming task to multiple outgoing tasks as a single block can create number of packets.

F. Packet Writing Stage

The *Packet Writing Stage* is the last step of the read process in which the packets are sent to the client. Each incoming task represents a packet and comes with a pointer to a butter which consists of packet data. A thread in *Packet Writing Stage* picks up the task and reads the packet from the buffer and writes it to the provided socket output stream. Then the thread waits for a status message from the client to confirm the receipt of the packet. This stage uses a bigger thread pool compared to other stages as a single read generates multiple tasks for this stage.

VI. Conclusion and Future work

In this paper, first we discussed the well-known SEDA architecture in detail. We highlighted five of the most interesting applications which use SEDA to handle highly concurrent systems. The power of SEDA architecture can be understood by looking at those applications. Then we went into details of SOR-HDFS which is a recent application of SEDA to improve the HDFS *Write* operation. We discussed the limitations and restrictions of the SOR-HDFS system as well.

Finally as the main contribution of this paper, we proposed a SEDA-based architecture to improve performance of HDFS *Read* operation. We carefully identified the main stages of *Read* operation to improve responsiveness and throughput under high loads. It will be interesting to implement this model on HDFS and compare performance with default HDFS as a future work.

References

[1] M. Welsh, D. Culler, and E. Brewer, “Seda: An architecture for well-conditioned, scalable internet services,” in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’01. New York, NY, USA: ACM, 2001, pp. 230–243. [Online]. Available: http://doi.acm.org/10.1145/502034.502057

[2] N. S. Islam, X. Lu, M. W.-u. Rahman, and D. K. D. Panda, “Sor-hdfs: A seda-based approach to maximize overlapping in rdma-enhanced hdfs,” in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC ’14. New York, NY, USA: ACM, 2014, pp. 261–264. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600715

[3] Apache HTTP Server. [Online]. Available: http://httpd.apache.org/

[4] Microsoft IIS. [Online]. Available: http://www.iis.net/

[5] Apache Tomcat. [Online]. Available: http://tomcat.apache.org/

[6] IBM Websphere Application Server. [Online]. Available: http://www-03.ibm.com/software/products/en/appserv-was

[7] Tiny/Turbo/Throttling HTTP server. [Online]. Available: http://www.acme.com/software/thttpd/

[8] V. S. Pai, P. Druschel, and W. Zwaenepoel, “Flash: An efficient and portable web server,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’99. Berkeley, CA, USA: USENIX Association, 1999, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268708.1268723

[9] A. Lakshman and P. Malik, “Cassandra: A decentralized structured storage system,” *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[10] Apache Cassandra. [Online]. Available: http://cassandra.apache.org/

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value store,” in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP ’07. New York, NY, USA: ACM, 2007, pp. 205–220. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281

[12] Mule ESB. [Online]. Available: http://www.mulesoft.org/

[13] Apache Mina. [Online]. Available: https://mina.apache.org/

[14] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, “Tapestry: A resilient global-scale overlay for service deployment,” *IEEE J.Sel. A. Commun.*, vol. 22, no. 1, pp. 41–53, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/JSAC.2003.818784

[15] H. Yan, R. Oliveira, K. Burnett, D. Matthews, L. Zhang, and D. Massey, “Bgpmon: A real-time, scalable, extensible monitoring system,” in *Proceedings of the 2009 Cybersecurity Applications & Technology Conference for Homeland Security*, ser. CATCH ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 212–223. [Online]. Available: http://dx.doi.org/10.1109/CATCH.2009.28

[16] N. Islam, X. Lu, M. Wasi-Ur-Rahman, and D. Panda, “Can parallel replication benefit hadoop distributed file system for high performance interconnects?” in *High-Performance Interconnects (HOTI), 2013 IEEE 21st Annual Symposium on*, Aug 2013, pp. 75–78.

[17] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, “High performance rdma-based design of hdfs over infiniband,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 35:1–35:35. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389044

[18] Guide to Cassandra Thread Pools. [Online]. Available: http://planetcassandra.org/blog/guide-to-cassandra-thread-pools/

[19] Apache FtpServer. [Online]. Available: https://mina.apache.org/ftpserver-project/index.html

[20] Apache SSHD. [Online]. Available: https://mina.apache.org/sshd-project/index.html

[21] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, “Towards a common api for structured peer-to-peer overlays,” in *Peer-to-Peer Systems II*, ser. Lecture Notes in Computer Science, M. Kaashoek and I. Stoica, Eds. Springer Berlin Heidelberg, 2003, vol. 2735, pp. 33–44. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45172-3_3

[22] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, “Awarded best student paper! - pond: The oceanstore prototype,” in *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, ser. FAST ’03. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1090694.1090696

[23] HDFS 2.5.0 code base. [Online]. Available: http://svn.apache.org/repos/asf/hadoop/common/tags/release-2.5.0/hadoop-hdfs-project/

[24] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/