Optimize Apache Storm in a container environment using InfiniBand RDMA and Performance tuning

In recent years, big data has become a hot topic, and the efficient and rapid processing of massive data challenges the design of software. Unlike Hadoop, which distributes batch processing of large volumes data, Storm is a representative distributed real-time stream-processing framework. It is fast, scalable, fault-tolerant and is being used in many data-center applications (e.g. YAHOO, Twitter, etc.).

For such distributed data-sensitive applications, low-latency, high-throughput communication modules have a critical impact on application performance. The communication framework used by the existing Storm is Netty, which is a non-blocking I/O client-server framework. Due to limitations of Ethernet network and TCP/IP protocol, the Netty communication framework limits the performance of Storm.

InfiniBand is a communication standard in high-performance computing. It provides higher bandwidth and lower latency than Ethernet and its Remote Direct Memory Access (RDMA) technology reduces memory copy compared to TCP/IP protocol, which effectively reduces the CPU overload and is good for performance improvement.

In this paper, we attempt to optimize Storm with InfiniBand technology. Specifically, we redesigned Storm's communication module with accelio-based JXIO. The performance evaluation on QDR (32 Gbps) shows that our design can achieve 2x and 4x peak speedup, compared with 1 Gigabit Ethernet (1 GigE) and IP-over-InfiniBand (IPoIB). It also significantly reduces the CPU load and communication delay.

1 Introduction

The processing of big data can be performed in batch or stream. Batch-processing framework is suitable for the application scenarios where data is not sensitive to time, such as business statistics. Stream-processing framework is more suitable for real-time application scenarios. Apache Storm is a representative of real-time stream processing framework and has crucial impacts on a wide range of use case, such as social networks, real-time transactions,

weather forecasts, recommendation systems, etc.

The latest version of Apache Storm's communication module implementation is based on the asynchronous event-driven network application framework Netty. Netty is a JAVA NIO-based client-server programming framework. Due to the high latency, limited bandwidth, and frequent memory copying of the TCP/IP protocol stack, the Netty-based communication module has a significant impact on the performance of Storm in distributed scenarios.

Storm performs data processing and transmission separately by creating multiple worker processes. The communication between workers is implemented by Netty, which is based on TCP/IP protocol. Netty implements asynchronous communication through a multi-threaded callback mechanism. Thread waiting and activation will perform context switching to occupy CPU resources.

In Storm's working environment, worker processes communicate frequently. One communication will perform multiple context switches and memory copies. This will occupy a large amount of CPU resources. Therefore, we analyze the structure of the Storm communication module and test communication latency and CPU overload. Finally, we used JXIO, which is a Java API for the Verbs interface to reconstruct the communication module of Storm, and tried to explore whether the high-performance computing communication technology can improve the performance of Storm.

The rest of this paper is organized as follows. In Section 2, we summarize Storm principles and abstract, and analyze the structure of the communication module. Section 3 presents our contribution to Storm’s messaging module and explains how we design and implement the RDMA-based Storm in detail. Section 4 demonstrates the evaluation of our newly designed Storm. Finally, Section5 and 6 concludes our work and describes future directions for RDMA-based Storm.

2 Preliminary

2.1. The abstraction in Storm

Data model:

Storm uses tuples as its data model. A tuple is a named list of values, and a field in a tuple can be an object of any type. Out of the box, Storm supports all the primitive types, strings, and byte arrays as tuple field values. To use an object of another type, you just need to implement for the type.

Streams:

The core abstraction in Storm is the "stream". A stream is an unbounded sequence of tuples. Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way. For example, user can transform a stream of tweets into a stream of trending topics.

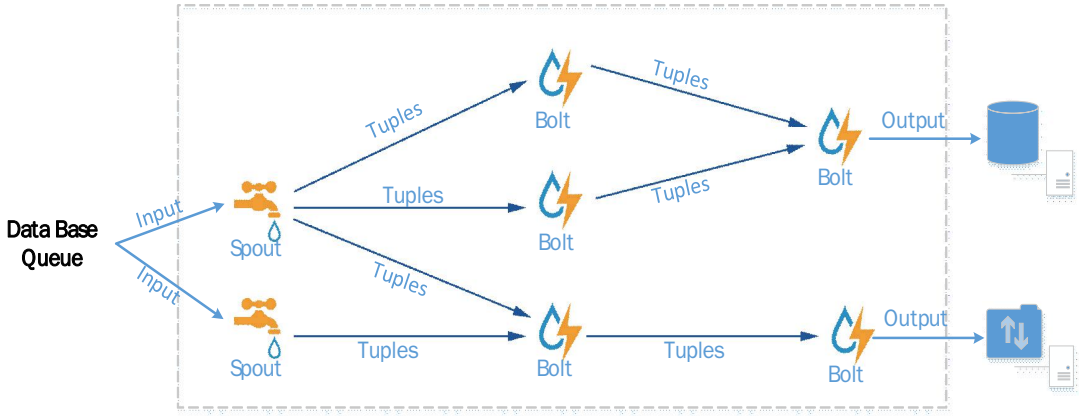
A spout is a source of streams. For example, a spout may read tuples off a queue and emit them as a stream. Alternatively, a spout may connect to the

Twitter API and emit a stream of tweets. A bolt consumes any number of input streams, does some processing, and possibly emits new streams.

Topologies:

To do real-time computation on Storm, user creates "topologies". A topology is a graph of computation. Each node in a topology contains processing logic, and links between nodes indicate how data should be passed around between nodes.

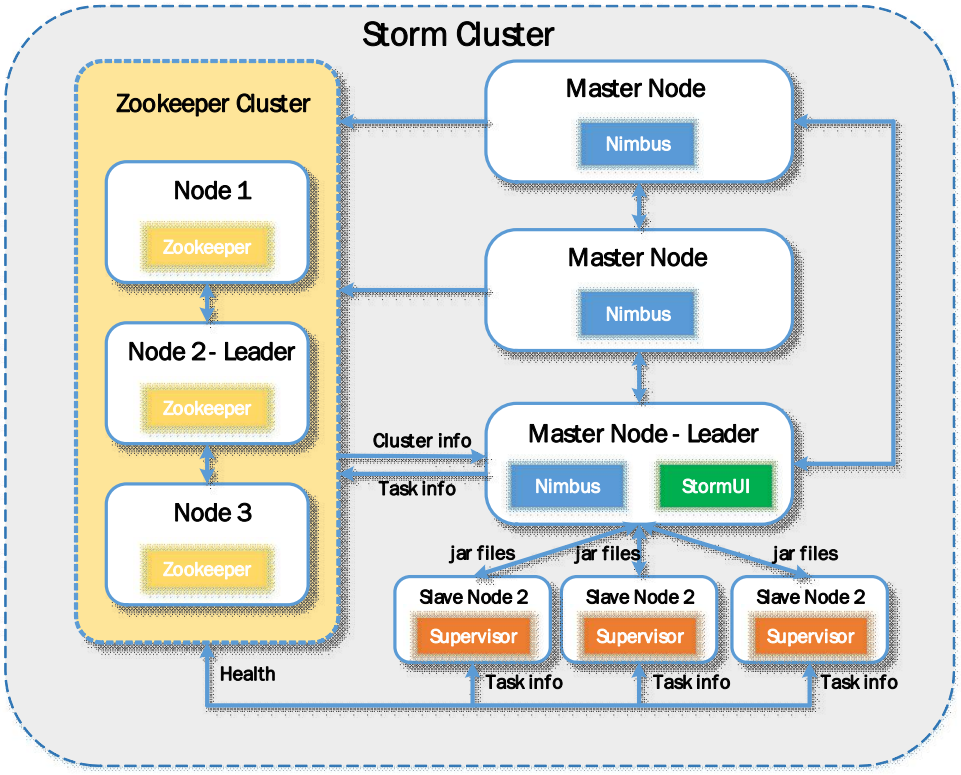
Networks of spouts and bolts are packaged into a "topology" which is the top-level abstraction that user submit to Storm clusters for execution. A topology is a graph of stream transformations where each node is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams. When a spout or bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream.



2.2. Components of a Storm cluster

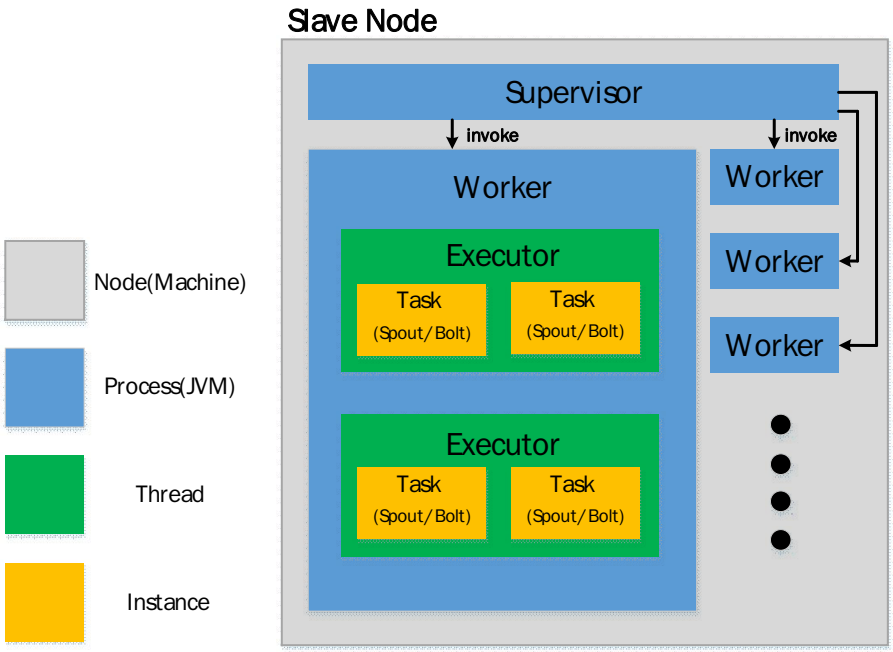
In this section, we briefly discuss our study on Apache Storm, summarize its principles, and abstract.

There are two kinds of nodes on a Storm cluster: the master node and the worker nodes. The master node runs a daemon called "Nimbus" that is similar to Hadoop's "JobTracker". Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures.



Each worker node runs a daemon called the "Supervisor". The supervisor listens for work assigned to its machine, starts, and stops worker processes as necessary based on what Nimbus has assigned to it. Each worker process executes a subset of a topology; a running topology consists of many worker processes spread across many machines.

All coordination between Nimbus and the Supervisors is done through a Zookeeper cluster, which is a distributed application coordination service. Additionally, the Nimbus daemon and Supervisor daemons are fail-fast and stateless; all state is kept in Zookeeper or on local disk. This design leads to Storm clusters being incredibly stable.

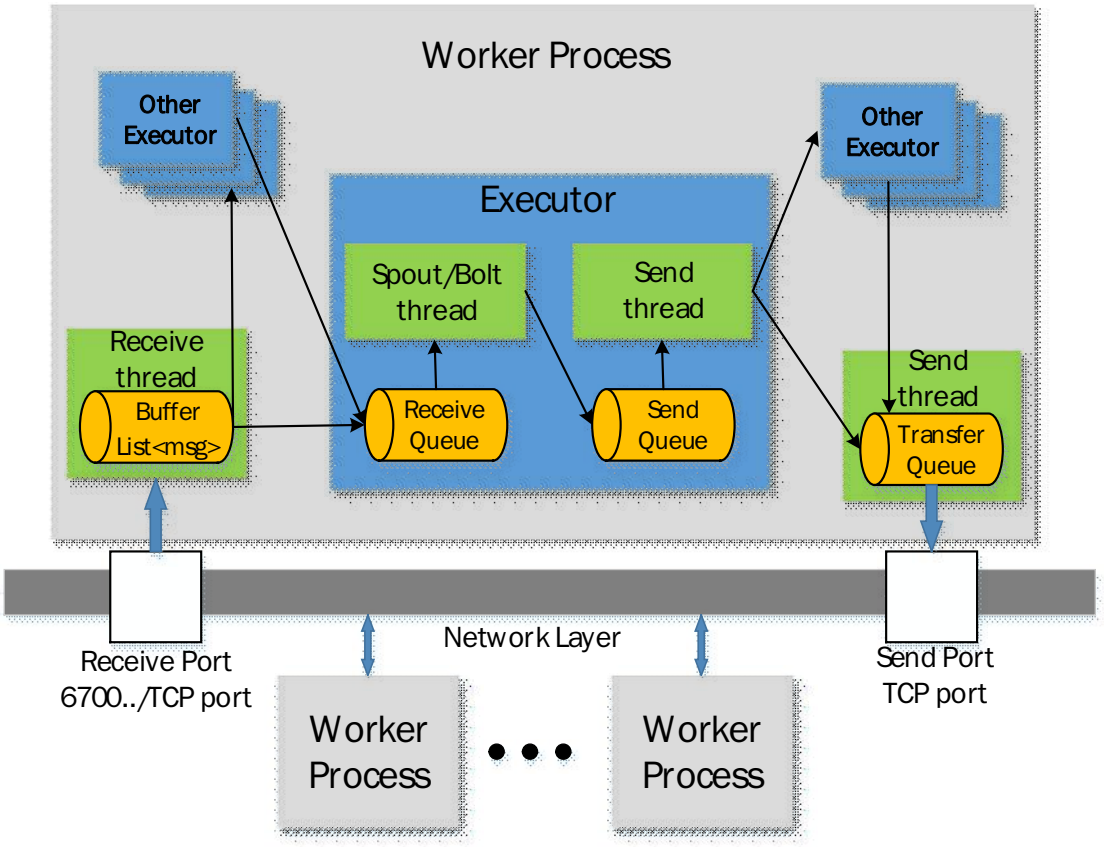


Parallelism is the key to fast streaming of Storm. Storm handles user-defined logic in parallel by creating a large number of Spouts and Bolts. If the abstract definition is mapped to a specific execution unit, the Storm server corresponds to the real physical machine Node, and the worker corresponds to a process. Each process has many Executors, that is, threads; each thread has a different task, such as sending and receiving messages, user-defined logic processing, etc. In order to handle different tasks, each Executor contains many Spout or Bolt instances corresponding to different Tasks.

This is a hierarchical structure of Storm. A Node can execute multiple Worker processes according to the user's definition. Worker, a Java process actually handles user-defined Topology. It processes tuples by executing Executor. Executor handles or emits tuples by instantiating Spout or Bolt.

2.3. Messaging processing structure

Storm has a Worker process for allocating resources, and a Worker process is the smallest unit of resource allocation. Each Worker process contains multiple Executors. The Executor is a component that actually executes the Task. It contains a work thread and a send thread. Each Executor has its own receive queue and send queue. Each worker process has a separate accepting thread listening to the receiving port. The Worker Receive Thread passes the received message to the corresponding Executor (one or more) receive queues via the Task number. The Worker Receive Thread sends each message uploaded from the network to the corresponding Executor's receive queue.



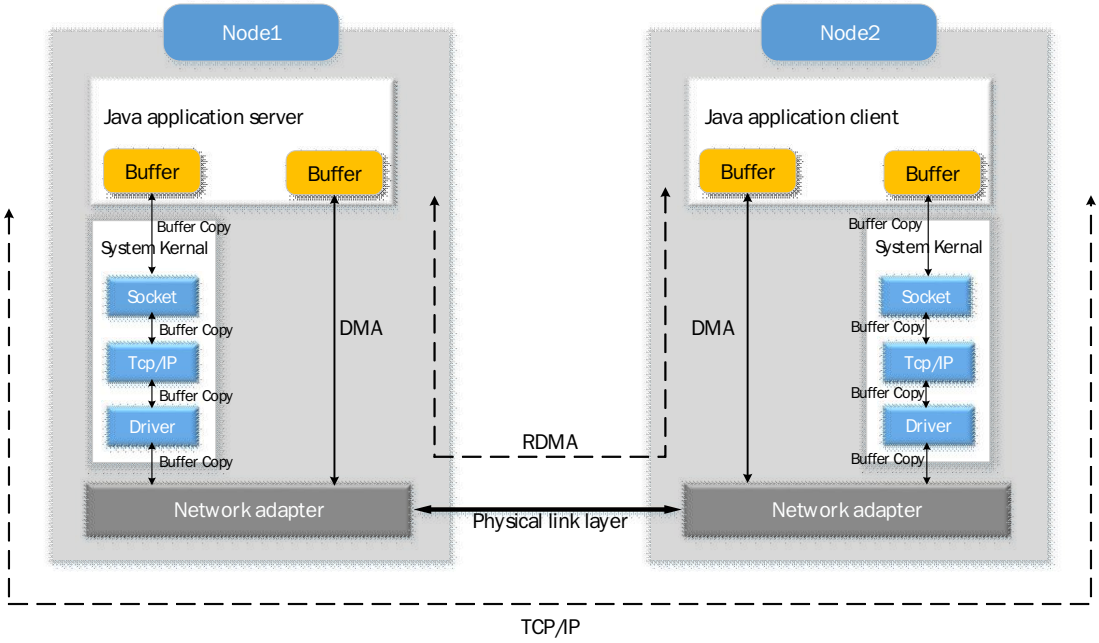
The Executor receives queue stores messages sent by other Executors inside the Worker or other Workers. The Executor worker thread takes the data from the receive queue and then calls the execute method to send the Tuple to the Executor's send queue. The Executor's sending thread gets the message from the send queue and chooses to send it to the Worker's transmission queue or other Executor's receive queue according to the destination address of the message. Finally, the worker's sending thread reads the message from the transmission queue and then sends the Tuple to the network.

2.4. InfiniBand and RDMA

InfinIBand is an industry standard switched fabric that is designed for high-speed, general purpose I/O interconnects nodes in high-end clusters.

One of the main features of InfiniBand is Remote Direct Memory Access (RDMA). This feature allows software to remotely read or update memory contents of another remote process without any software involvement at the remote side. It uses a queue-based model. Each Queue Pair (QP) has a certain number of work queue elements. Upper-level software can place a Work Request (WR) on a QP. Then the Host processes the WR Channel Adapter (HCA). When work is completed, a Work Completion (WC) notification is placed on the Completion Queue (CQ). Upper level software can detect completion by polling the CQ or by asynchronous events.

InfiniBand software stacks also provide a driver for implementing the IP layer, which exposes the InfiniBand device as just another network interface available from the system with an IP address. It is often called “IP over IB” or IPoIB for short.



InfiniBand provides a programming interface called Verbs to use RDMA in application programs. Verbs is a low-level communication interface that enables application to implement RDMA communications. The benefits of RDMA communications as follow.

Zero-copy:

The TCP/IP protocol accepts data from the network card to the kernel buffer and then to the app buffer. The process of sending needs to copy data from the app buffer to the socket buffer and then copy to the network card. The concept of zero replication is to avoid copying data in kernel space and user space. The main purpose is to reduce unnecessary copying and avoid having the CPU do many data copying tasks. Some hardware can complete the work of TCP/IP protocol. Data can be transmitted directly between the app buffer and the hardware without going through the socket buffer.

Kernel bypass:

The application can directly manipulate the hardware to transfer data without the involvement of the system kernel. Without system interruption, frequent context switching is not required, and the complex protocol stack is rid of the hardware.

No CPU involvement:

RDMA (Remote Direct Memory Access) allows computers on a network to exchange data in main memory without involving any computer's processor, cache, or operating system interrupt. Like local-based direct memory access (DMA), RDMA improves throughput and performance because it frees up resources.

2.5. JXIO

JXIO is Java API over AccelIO (C library). AccelIO is a high-performance asynchronous reliable messaging and RPC library optimized for hardware acceleration. RDMA and TCP/IP transport are implemented, and other transports, such as shared-memory can take advantage of efficient and convenient API.

Interface Summary	**Description**
ClientSession.Callbacks	It defines callback processing for events on success/failure of connection request or success/failure of message transmission.
EventQueueHandler.Callbacks	Its callback is called on server Side. If a request from client arrives and there are no more Msg list is empty, this callback is called.
ServerPortal.Callbacks	It defines callback processing for events on client connection requests.
ServerSession.Callbacks	It receives messages from the clients and defines callback processing for events on success/failure of the response message transmission.
Class Summary	**Description**
ClientSession	ClientSession is the object that connects to the Server and transmits messages.
EventQueueHandler	This class receives message in the server or client, and generates an event for the message.
Msg	Msg It is a message object used when Accelio's server and client communicate.
MsgPool	Both client and server use MsgPool (different instance of the same object).
ServerPortal	ServerPortal is the object, which listens to incoming connections.
ServerSession	ServerSession is the object, which receives Msgs from Client and sends responses.

3 Contribution

We analyzed the design pattern of the Storm communication module and the communication logic and flow of Netty, and reconstructed the communication module with JXIO based on the defined communication function interface. Section 3.1 focuses on the design pattern of the communication module, the functions of Storm related important classes and the principles and logic of Netty

communication. Section 3.2 presents the methods and details of our redesign with JXIO.

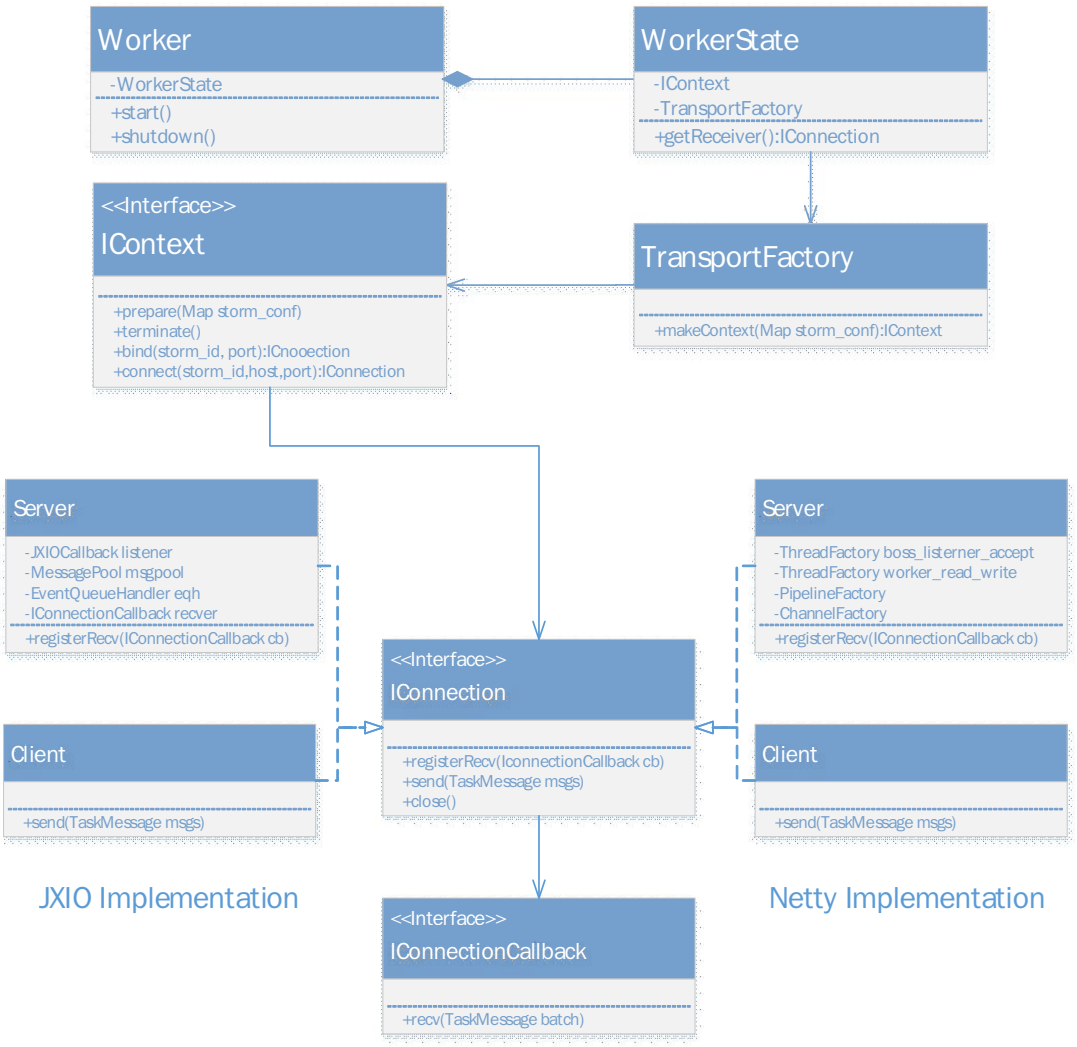
3.1. Messaging Transport Layer Design Analysis

In the part of the message component of Storm's source code, several interfaces are defined. In order to meet the communication requirements between the workers. The relevant interfaces are as follows.

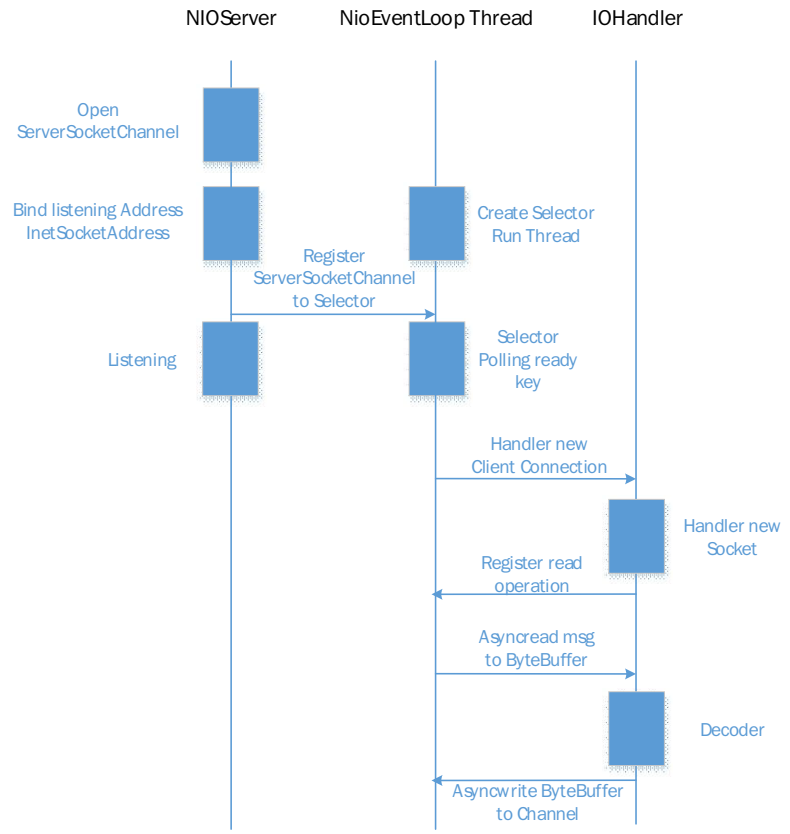
TransportFactory class: The **Workerdaemon** class contains a **WorkerState** class that controls the state transition of the communication and handles communication. **WorkerState** uses the **TransportFactory** class to create the communication context. Apache Storm uses Netty as its communication framework by default. We modified the **TransportFactory** class, which can be used to select the communication component of JXIO or the communication component of Netty through configuration parameters.

Context class: It is an implementation of the **IContext** interface. It contains methods for creating Server and Client. We rebuilt the JXIO Context. The Worker daemon process can get the JXIO server and client, so that RDMA can be used for communication.

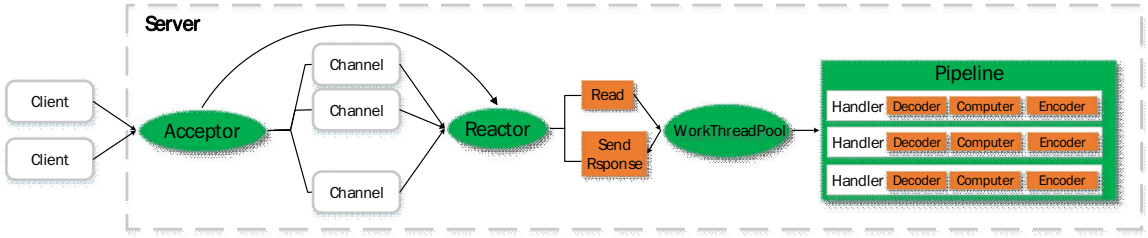
IconnectionCallback: This interface defines the asynchronous method of the receive message. **ServerSession** implements non-blocking asynchronous processing of message received by overriding its Receive method.



The Storm communication module is implemented by Netty framework by default. Netty is based on Java's NIO (Java non-blocking IO) and uses Reactor mode as its threading model. Netty's workflow as shown in the **figure**. **NIOServer** uses a separate NIO thread pool to receive client connections. When the TCP connection request processing is completed, **NIOServer** registers the newly created **SocketChannel** to a thread in the I/O thread pool. In addition, assign an **EventLoop** to this Channel. Each **EventLoop** corresponds to a thread. The IO operation on the Channel will be executed on **EventLoop**. Moreover, the thread in **EventLoop** will execute the method defined in the handler on **ChannelPipeline**.



Netty uses the Reactor threading model. **Reactor** is event-driven and can have multiple concurrent event sources. The **Acceptor** is responsible for listening for events. Whenever an event arrives, the **Acceptor** registers the **Channel** with the **Selector** and enters it into the service handler, i.e. **Reactor** thread. The **Dispatcher** in the **Reactor** thread distributes the event to the corresponding **Pipeline** for processing. **Pipeline** is the sequential combination of **Handlers**, i.e. the message-processing pipeline.



3.2. Detailed Design

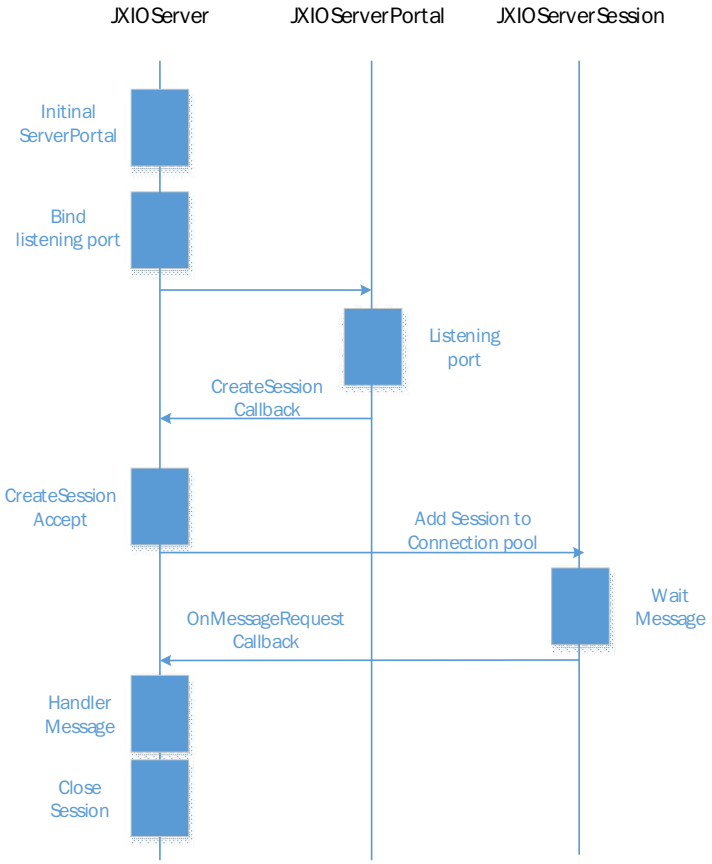
JXIO is Java API over AccelIO, which is a high-performance asynchronous reliable messaging, and RPC library optimized for hardware acceleration. Both RDMA and TCP/IP transport are implemented. The main classes of JXIO are as follows:

JXIOServer: Initialize parameters, bind IP addresses, set listening ports, register memory pools, and initialize event handling queues to handle client

connections and IO events.

JXIOServerPortal: ServerPortal is the object, which listens to incoming connections. It can accept/reject or forward to another portal the new session request and works as a listener to listens on a well-known port and redirects the session to a different thread (it can also accept the session). There are two events on its lifetime, namely **onSessionNew**, which is for a new session arrives from Client and **onSessionEvent,** which is for CLOSED event arrives. On each of them, a method of interface **Callbacks** is invoked.

JXIOServerSession: ServerSession is the object, which receives Msgs from Client and sends responses. ServerSession receives three events on his lifetime, namely **OnRequest**, which is for a request from Client is received, **onSessionEvent** which is for several types of session events and **onMsgError**. On each of them, a method of interface Callbacks is invoked.



Based on the communication logic of Storm, we rebuilds JXIO communication module's Callback class.

ServerPortal.Callback:

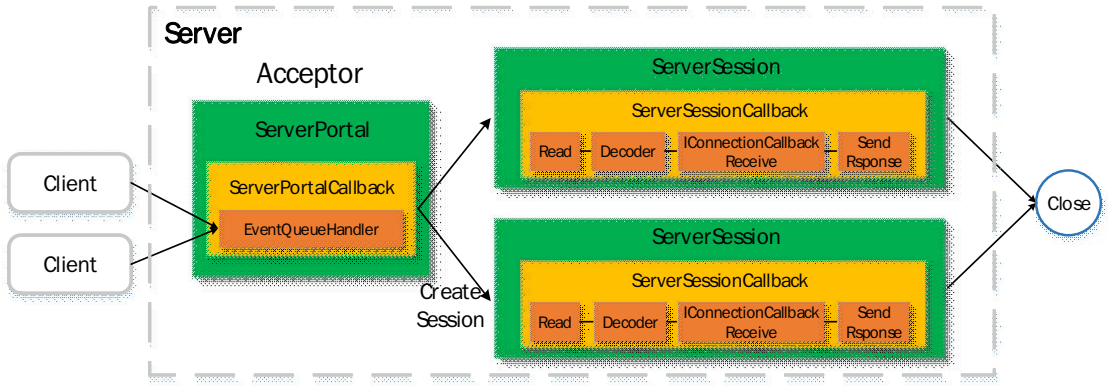
It is used to listen to client events. When there is a client connection, it calls the **CreateSession** callback method, creates a connection session, and puts it into the connection pool. At the same time, it registers the connection to the EventQueueHandler, so that it can send and receive messages.

ServerSession.Callback:

This callback method is used to receive the client's message. It requires the Decoder method to decode the received message into the **TaskMessage** message structure defined by Storm. Then the message is placed in the receive queue Buffer.

ClientSession.Callback:

Since JXIO is in Request-Response mode, the client needs to receive a response message after sending the data. When the connection succeeds or fails, or the message is sent successfully/fails and the client will receive the response message from the server for further processing.



4 Evaluation

4.1. Experimental Setup

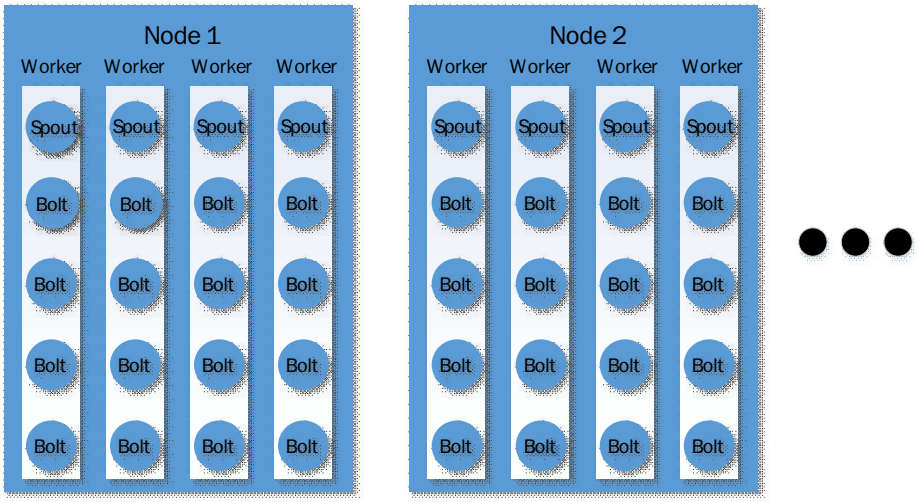
The cluster configuration we use for evaluation is: 8 Sugon node with dual Intel Xeon E5-2660 processors and 128GB RAM. Each node runs Centos Linux release 7.3.1611. The nodes are connected with both 1GigE and Mellanox 40G QDR infiniBand.

We use Storm-2.0.1, Zookeeper-3.4.5 in our experiments. The Zookeeper and Storm are deployed on 8 nodes of the cluster. In detail, the distributed application coordination services Zookeeper is evenly deployed on 8 nodes, so that the nimbus and supervisor daemon can share the state information at minimal cost. In addition, the nimbus daemon is deployed on node1 and supervisor daemon is deployed on the remaining 7 nodes.

The variables in the experiment are the communication mode, the size of the transmitted message buffer, and the size of the cluster. The dependent variable is the processing delay of the Storm, the CPU load of the node, and the throughput of the message. First, we run Storm on 4 nodes, adjust the message size by setting configuration parameter, which is **topology.transfer.buffer.size**, and test the processing delay and CPU load of Storm in TCP, IPoIB and RDMA modes, then find the best buffer size of Storm performance. Second, we test the performance of Storm by setting different cluster sizes. Finally, we built a real streaming system with kafaka, redis, and Storm, and tested the performance of the entire system.

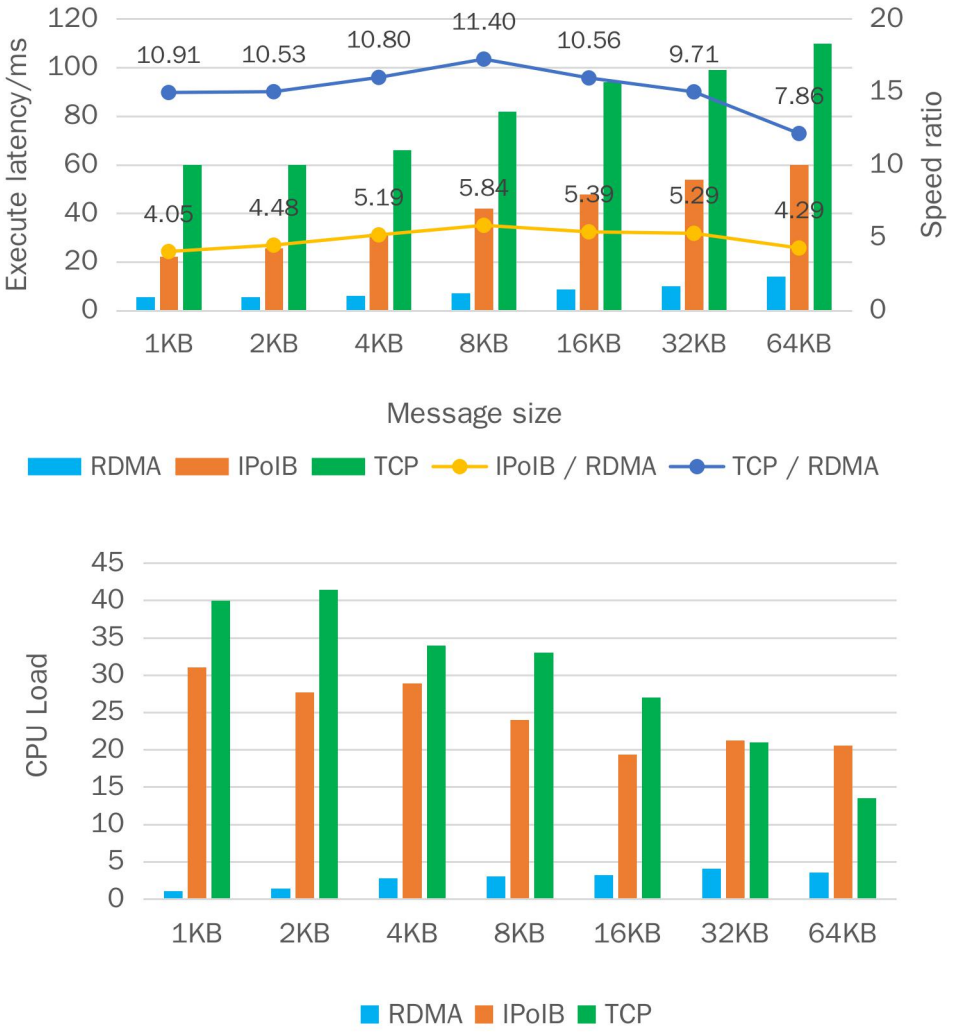
4.2. The effect of message size on performance

We design a Topology specifically for testing. Four worker processes are started on one node, and 16 Worker processes are started on the cluster. Each Worker process contains 5 Executors, and each Executor performs a task in Spout or Bolt. The grouping strategy uses broadcast grouping, i.e. Allgrouping, to ensure maximum traffic between inter-Workers. First, spout initializes some data, generates some tuples, and puts the generated tuple into the send buffer. Then Send thread sends the buffered tuples to the next bolt of each worker process according to the buffer size set in the configuration parameters. Similarly, the bolt that receives the tuples sends the data to each subsequent bolt.



We conclude our experiment in two aspects according to the experimental data:

- The increase in the size of the send buffer will result in an increase in processing delay, which has the most significant impact on TCP and has less impact on RDMA. Both RDMA and IPoIB achieve significant improvement over TCP. Compared with TCP, RDMA can achieve an acceleration ratio of about 10 times. Moreover, Compared with IPoIB, RDMA can achieve an acceleration ratio of about 5 times.
- The size of the send buffer will also affect the CPU load. Because IPoIB takes advantage of hardware, it can process more packets than TCP at the same time. However, this amplifies the shortcomings of TCP/IP protocol, that is, frequent context switching and memory copying, which will increase the CPU load. RDMA significantly reduces CPU load compared to IPoIB because there is no need to interrupt the operating system for memory copying.



4.3. The impact of distributed scale on performance