WHITE PAPER

# Hazelcast IMDG Performance with Serialization


## By Fuad Malikov

Co-founder & Vice President, Technical Operations
Hazelcast

WHITE PAPER

# Hazelcast IMDG Performance with Serialization

## TABLE OF CONTENTS

# Maximizing Hazelcast Performance with Serialization

## WHAT IS SERIALIZATION?

In computer science, object serialization is the process of transforming an object's state into a sequence of bytes, and then rebuilding them into a live object at some future time (deserialization).

In terms of Hazelcast, serialization is conversion of a Java object into a binary representation of the object and deserialization is reverting back that binary data into a Java object. Serialization is key to making your operations faster and also more space efficient.

All the Java objects that you put into Hazelcast get serialized. This is needed since Hazelcast is a distributed system. The data, along with its replicas, is stored in different partitions, on multiple nodes. Often data that you need will not be present on your local machine and Hazelcast will need to fetch that data from another remote machine. This means data will go through the network, and therefore requires serialization.

Also note, that most serialization and deserialization happens on client side. Based on the operation, however, some requests require both processes on the server side. In this case the necessary classes must be available.

## WHERE IS SERIALIZATION USED?

1. **Key / Value** - When you add key/value objects to a Hazelcast map, these key/value objects are serialized.

   `cartMap.put(cart.id, cart);` … serialization + deserialization

   The put method serializes the key and value objects. This method also returns the previous value associated with the key. This requires deserialization of the old value.

2. **Queue / Set / List Items** - If you do a queue/set/list operation, the items that you put or consume in a queue/set/list, will be serialized.

   `taskQueue.offer(task);` … serialization

3. **Executor Service** - While using an executor service that involves sending a runnable, that needs to be serialized.

   `executor.execute(new MyRunnable());` … serialization

4. **Entry Processor** - An entry processor within a map, needs the data to be serialized, since the data may be distributed across several nodes and may need to be transported across these different nodes.

   `cartMap.get(orderId);` … deserialization

   The key here, will be serialized. However, the value returned will need to be deserialized.

5. **Lock** - The binary representation of an object is compared to check if its locked or not. Hence, this requires serialization.

   `lock.lock(accountId);` … serialization

6. **Topic Messages** - When sending a message to a topic, there will be numerous consumers that receive that message, This too, requires serialization.

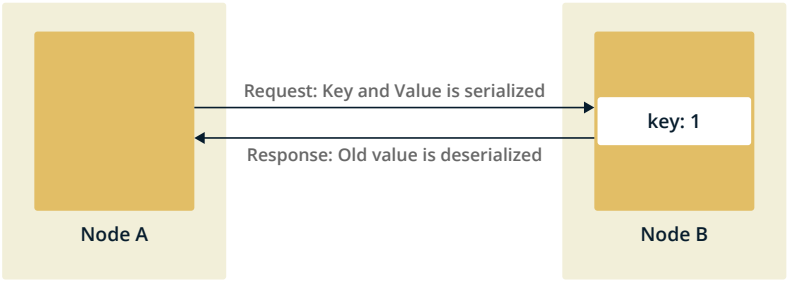
   `topic.publish (myMessageObject);` … serialization

   The message will be deserialized on the receiver side.

## PUT OPERATION SERIALIZATION CYCLE

Let us assume that we have a two node cluster with an application running on both nodes. Let us also assume that key 1 is mapped to Node B. We then call a `cartMap.put(1, cart)` operation from Node A.

```
cartMap.put(1, cart);
```




When the put operation is called, the key 1 along with the value cart will be serialized and sent to Node B, where this binary representation of the key/value objects is stored. Since the put operation returns the previous value of key 1, if present, the binary representation of the old value is sent back to Node A. On Node A, this binary representation is again deserialized back to a Java object.

## OPTIMIZED TYPES

Hazelcast optimizes the serialization for the below types, and the user cannot override this behavior:

| | | |
|---|---|---|
| Byte | byte[] | String |
| Boolean | char[] | Date |
| Character | short[] | BigInteger |
| Short | int[] | BigDecimal |
| Integer | long[] | Class |
| Long | float[] | Enum |
| Float | double[] | |
| Double | | |

Hence, when objects of the these types are added to Hazelcast, the user need not worry about the serialization and deserialization of these objects. However, this is not the case for complex objects.

## SPECIAL TYPES

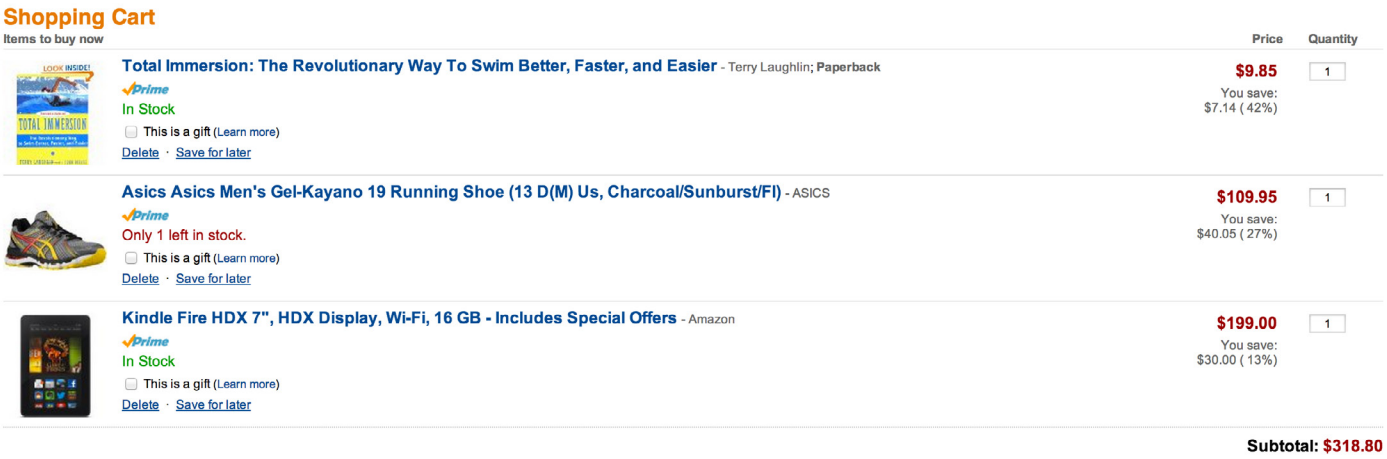
For complex objects, the following interfaces are used, for serialization and deserialization

1. Java
   - java.io.Serializable
   - java.io.Externalizable
2. Hazelcast
   - com.hazelcast.nio.serialization.DataSerializable
   - com.hazelcast.nio.serialization.IdentifiedDataSerializable
   - com.hazelcast.nio.serialization.Portable
3. Custom Serialization

We will discuss these in more detail ahead.

## SAMPLE DOMAIN – SHOPPING CART

For this tutorial, we have built a small test application, along with benchmarks to test the performance of this application. This application is a shopping-cart like e-commerce website. The two main domain objects are `ShoppingCartItem` and `ShoppingCart`. Below is a sample shopping cart containing three items.




The `ShoppingCartItem` and `ShoppingCart` objects have the following attributes.

### ShoppingCartItem Implementation

```
public class ShoppingCartItem {
   public long cost;
   public int quantity;
   public String itemName;
   public boolean inStock;
   public String url;
}
```

### ShoppingCart Implementation

```
public class ShoppingCart {
   public long total = 0;
   public Date date;
   public long id;
   private List<ShoppingCartItem> items = new ArrayList<>();

   public void addItem(ShoppingCartItem item) {
      items.add(item);
      total += item.cost * item.quantity;
   }

   public void removeItem(int index) {
      ShoppingCartItem item = items.remove(index);
      total -= item.cost * item.quantity;
   }

   public int size() {
      return items.size();
   }
}
```

## BENCHMARK – 100,000 TIMES

The benchmark program is a simple one, that we run on our local machine, using a single thread and on a single node.

**ShoppingCartBenchmark Implementation**

```
@Override
public void writePerformance() {
   Random random = new Random();
   for (int k = 0; k < OPERATIONS_PER_INVOCATION; k++) {
      ShoppingCart cart = createNewShoppingCart(random);
      cartMap.set(cart.id, cart);
   }
}

@Override
public void readPerformance() {
   Random random = new Random();
   for (int k = 0; k < OPERATIONS_PER_INVOCATION; k++) {
      long orderId = random.nextInt(maxOrders);
      cartMap.get(orderId);
   }
}

private ShoppingCart createNewShoppingCart(Random random) {
   ShoppingCart cart = new ShoppingCart();
   cart.id = random.nextInt(maxOrders);
   cart.date = new Date();
   int count = random.nextInt(maxCartItems);
   for (int k = 0; k < count; k++) {
      ShoppingCartItem item = createNewShoppingCartItem(random);
      cart.addItem(item);
   }
   return cart;
}

private ShoppingCartItem createNewShoppingCartItem(Random random) {
   int i = random.nextInt(10);
   ShoppingCartItem item = new ShoppingCartItem();
   item.cost = i * 9;
   item.quantity = i % 2;
   item.itemName = "item_" + i;
   item.inStock = (i == 9);
   item.url = "http://www.amazon.com/gp/product/" + i;
   return item;
}
```

We use the `writePerformance()` method to measure the performance of serialization operations. Notice that we have used the `cartMap.set()` method instead of the `cartMap.put()` method. This is because , unlike the `put` method, the `set` method does not return any value. Hence , there is no deserialization operation happening in the background. This allows us to measure the performance of only the serialization that takes place when a `ShoppingCart` object is added to the `cartMap` using the `set` method.

As part of our performance test, we run the `cartMap.set()` method 100,000 times using random `ShoppingCart` objects. We then perform this test 5 times and take the average running time. Finally in the end, we calculate the number of `set` operations per millisecond. This will give us the number of serializations

that take place per millisecond. We run this benchmark test using our different serialization interfaces and find out which interface performs the best.

Similarly, the `readPerformance()` method performs the get operation on the map 100,000 times. We again do this experiment 5 times and take the average running time. This will give us the number of deserializations that can take place in one millisecond, thus reflecting the performance of the corresponding serialization interface used.

The main class is the Benchmark.java class.

```
public class Benchmark {
   public static void main(String[] args) throws IOException, RunnerException {
      // doBenchmark(new SerializableBenchmark());
      // doBenchmark(new ExternalizableBenchmark());
      // doBenchmark(new DataSerializableBenchmark());
      // doBenchmark(new IdentifiedDataSerializableBenchmark(false));
      // doBenchmark("Unsafe ", new IdentifiedDataSerializableBenchmark(true));
      // doBenchmark(new PortableBenchmark(false));
      // doBenchmark("Unsafe ", new PortableBenchmark(true));
      doBenchmark(new KryoBenchmark());
   }

   private static void doBenchmark(ShoppingCartBenchmark sb) {
      doBenchmark("", sb);
   }

   private static void doBenchmark(String prefix, ShoppingCartBenchmark sb) {
      doBenchmark(prefix + sb.getClass().getSimpleName().
         replace("Benchmark", "") + " Write Performance ", sb, () ->
            sb.writePerformance());
      doBenchmark(prefix + sb.getClass().getSimpleName().
         replace("Benchmark", "") + " Read Performance", sb, () ->
            sb.readPerformance());
   }

   private static void doBenchmark(String name, ShoppingCartBenchmark sb, Runnable
runnable) {
      System.out.println(name);
      sb.setUp();
      long total = 0;
      int count = 5;
      for (int i = 1; i <= count; i++) {
         System.out.print(name + " :: " + i + " iteration: ");
         long start = System.currentTimeMillis();
         runnable.run();
         long end = System.currentTimeMillis();
         long ops = (long) ShoppingCartBenchmark.OPERATIONS_PER_INVOCATION / (end -
start);
         total += ops;
         System.out.println(ops + " ops in ms");
      }
      sb.tearDown();
      System.out.println(name + ":: average " + total / count);
   }
}
```

This class runs, one by one, the different serialization interfaces. As shown in the `doBenchmark(String prefix, ShoppingCartBenchmark sb)` method, we will first test the write performance and then the read performance. Each of these tests are run 5 times, as shown in the `for` loop in the `doBenchmark(String name, ShoppingCartBenchmark sb, Runnable runnable)` method. We then calculate the average run time for the operations.

Now let us look at the different serialization methods.

## JAVA.IO.SERIALIZABLE

This is a standard java interface. In the Java world, serializability of a class can be enabled by the class implementing the `java.io.Serializable` interface. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

**Pros**

- Standard and Basic
- Doesn't require any implementation

**Cons**

- Takes more time and cpu
- Occupies more space

The only difference in code is that the `ShoppingCart` and `ShoppingCartItem` objects will now implement the Serializable interface as shown below.

**Serializable - ShoppingCart Implementation**

```
public class ShoppingCart implements Serializable {
   public long total = 0;
   public Date date;
   public long id;
   private List<ShoppingCartItem> items = new ArrayList<>();

   public void addItem(ShoppingCartItem item) {
      items.add(item);
      total += item.cost * item.quantity;
   }

   public void removeItem(int index) {
      ShoppingCartItem item = items.remove(index);
      total -= item.cost * item.quantity;
   }

   public int size() {
      return items.size();
   }
}
```

**Serializable - ShoppingCartItem Implementation**

```
public class ShoppingCartItem implements Serializable {
   public long cost;
   public int quantity;
   public String itemName;
   public boolean inStock;
   public String url;
}
```

When we then ran the benchmark program with the java.io.Serializable interface, we got the following results. You may get different values, but they will be more or less in the same range.

Serializable Write performance - average is 31 operations per millisecond
Serializable Read performance - average is 46 operations per millisecond
Binary Object Size- average is 525 bytes

## JAVA.IO.EXTERNALIZABLE

This is another standard Java interface. Only the identity of the class of an Externalizable instance is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances. The writeExternal and readExternal methods of the Externalizable interface are implemented by a class to give the class complete control over the format and contents of the stream for an object and its supertypes.

**Pros**

- Standard
- Efficient than Serializable in terms of CPU and Memory

**Cons**

- Requires to implement the actual Serialization

**Externalizable - ShoppingCartItem Implementation**

```
   @Override
   public void writeExternal(ObjectOutput out) throws IOException {
      out.writeLong(cost);
      out.writeInt(quantity);
      out.writeUTF(itemName);
      out.writeBoolean(inStock);
      out.writeUTF(url);
   }

   @Override
   public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
      cost = in.readLong();
      quantity = in.readInt();
      itemName = in.readUTF();
      inStock = in.readBoolean();
      url = in.readUTF();
   }
```

**Externalizable - ShoppingCart Implementation**

```
    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
       out.writeLong(total);
       out.writeLong(date.getTime());
       out.writeLong(id);
       out.writeInt(items.size());
       items.forEach(item -> {
          try {
             item.writeExternal(out);
          } catch (IOException e) {
             e.printStackTrace();
          }
       });
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
 ClassNotFoundException {
       total = in.readLong();
       date = new Date(in.readLong());
       id = in.readLong();
       int count = in.readInt();
       items = new ArrayList<>(count);
       for (int i = 0; i < count; i++) {
          ShoppingCartItem item = new ShoppingCartItem();
          item.readExternal(in);
          items.add(item);
       }
    }
```

The `ShoppingCart` and `ShoppingCartItem` objects will now implement the Externalizable interface. Also, the `ShoppingCart` and `ShoppingCartItem` class will need to implement two additional methods, `writeExternal` and `readExternal`.

When we then ran the benchmark program with the `java.io.Externalizable` interface, we got the following results. You may get different values, but they will be more or less in the same range.

Externalizable Write performance - average is 61 operations per millisecond
Externalizable Read performance - average is 60 operations per millisecond
Binary Object Size - average is 235 bytes

## COM.HAZELCAST.NIO.SERIALIZATION.DATASERIALIZABLE

This is an interface that Hazelcast provides. It is very similar to Externalizable. However, it is a bit more optimized. Unlike the Externalizable, it does not store class meta data, along with the class name. Instead of the writeExternal and readExternal methods of Externalizable, this interface requires implementation of the writeData and readData methods.

**Pros**

- Efficient than Serializable in terms of CPU and Memory

**Cons**

- Hazelcast specific
- Requires to implement the actual serialization
- Uses Reflection while deserializing

**DataSerializable - ShoppingCartItem Implementation**

```
@Override
public void writeData(ObjectDataOutput out) throws IOException {
   out.writeLong(cost);
   out.writeInt(quantity);
   out.writeUTF(itemName);
   out.writeBoolean(inStock);
   out.writeUTF(url);
}

@Override
public void readData(ObjectDataInput in) throws IOException {
   cost = in.readLong();
   quantity = in.readInt();
   itemName = in.readUTF();
   inStock = in.readBoolean();
   url = in.readUTF();
}
```

**DataSerializable - ShoppingCart Implementation**

```
@Override
public void writeData(ObjectDataOutput out) throws IOException {
   out.writeLong(total);
   out.writeLong(date.getTime());
   out.writeLong(id);
   out.writeInt(items.size());
   items.forEach(item -> {
      try {
         item.writeData(out);
      } catch (IOException e) {
         e.printStackTrace();
      }
   });
}

@Override
public void readData(ObjectDataInput in) throws IOException {
   total = in.readLong();
   date = new Date(in.readLong());
   id = in.readLong();
   int count = in.readInt();
   items = new ArrayList<>(count);
   for (int i = 0; i < count; i++) {
      ShoppingCartItem item = new ShoppingCartItem();
      item.readData(in);
      items.add(item);
   }
}
```

When we ran the benchmark program with the `com.hazelcast.nio.serialization.DataSerializable` interface and got the following results.

DataSerializable Write performance - average is 64 operations per millisecond
DataSerializable Read performance - average is 59 operations per millisecond
Binary Object Size - average is 231 bytes

## COM.HAZELCAST.NIO.SERIALIZATION.IDENTIFIEDDATASERIALIZABLE

This interface was introduced in Hazelcast 3. It extends the DataSerializable interface. IdentifiedDataSerializable eliminates the reflection that is used by DataSerializable during deserialization. It contains two more methods, `getFactoryId` and `getId`. Instead of Reflection, we have a factory implementation where we pass an id and this implementation will create and return an object having that id.

**Pros**

- Efficient than Serializable in terms of CPU and Memory
- Doesn't use Reflection while deserializing

**Cons**

- Hazelcast specific
- Requires to implement the actual serialization
- Requires to implement a Factory and configuration

**IdentifiedDataSerializable - ShoppingCartItem Implementation**

```
@Override
public void writeData(ObjectDataOutput out) throws IOException {
   out.writeLong(cost);
   out.writeInt(quantity);
   out.writeUTF(itemName);
   out.writeBoolean(inStock);
   out.writeUTF(url);
}

@Override
public void readData(ObjectDataInput in) throws IOException {
   cost = in.readLong();
   quantity = in.readInt();
   itemName = in.readUTF();
   inStock = in.readBoolean();
   url = in.readUTF();
}

@Override
public int getFactoryId() {
   return 1;
}

@Override
public int getId() {
   return 1;
}
```

**IdentifiedDataSerializable - ShoppingCart Implementation**

```
@Override
public void writeData(ObjectDataOutput out) throws IOException {
   out.writeLong(total);
   out.writeLong(date.getTime());
   out.writeLong(id);
   out.writeInt(items.size());
   items.forEach(item -> {
      try {
         item.writeData(out);
      } catch (IOException e) {
         e.printStackTrace();
      }
   });
}

@Override
public void readData(ObjectDataInput in) throws IOException {
   total = in.readLong();
   date = new Date(in.readLong());
   id = in.readLong();
   int count = in.readInt();
   items = new ArrayList<>(count);
   for (int i = 0; i < count; i++) {
      ShoppingCartItem item = new ShoppingCartItem();
      item.readData(in);
      items.add(item);
   }
}

@Override
public int getFactoryId() {
   return 1;
}

@Override
public int getId() {
   return 2;
}
```

In addition to the `writeData` and `readData` methods, the `ShoppingCartItem` class contains two other methods, `getFactoryId` and `getId`. Similar is the case with the `ShoppingCart` class. Since both `ShoppingCart` and `ShoppingCartItem` use the same factory, the `getFactoryId` methods in both implementations will return the same id, 1. However, since these are two different objects the `getId` method returns 1 for `ShoppingCartItem` and 2 for `ShoppingCart`.

**IdentifiedDataSerializable - ShoppingCartDSFactory Implementation**

```
public class ShoppingCartDSFactory implements DataSerializableFactory {

   @Override
   public IdentifiedDataSerializable create(int i) {
      switch (i) {
      case 1:
         return new ShoppingCartItem();
      case 2:
         return new ShoppingCart();
      default:
         return null;
      }
   }
}

Config config = new Config();
config.getSerializationConfig().addDataSerializableFactory(1, new
ShoppingCartDSFactory());
hz = Hazelcast.newHazelcastInstance(config);
```

There is one extra step here - the implementation of the factory itself. The `ShoppingCartDSFactory` implementation creates and returns the object corresponding to the id. The factory also needs to be added to the serialization config.

We ran the benchmark program with the `com.hazelcast.nio.serialization.IdentifiedDataSerializable` interface and got the following results.

IdentifiedDataSerializable Write performance - average is 68 operations per millisecond
IdentifiedDataSerializable Read performance - average is 60 operations per millisecond
Binary Object Size - average is 186 bytes

## COM.HAZELCAST.NIO.SERIALIZATION.PORTABLE

This is another interface provided by Hazelcast. This supports versioning of the serialized object. Also, it allows for partial deserialization of an object to answer a query, without deserializing the whole object.

**Pros**

- Efficient than Serializable in terms of CPU and Memory
- Doesn't use Reflection while deserializing
- Supports versioning
- Supports partial deserialization during Queries

**Cons**

- Hazelcast specific
- Requires to implement the actual serialization
- Requires to implement a Factory and Class Definition
- Class definition is also sent together with Data but stored only once per class

The `writeData` and `readData` methods of IdentifiedDataSerializable are replaced by the `writePortable` and `readPortable` methods in Portable. The advantage that this interface has over others is that since we attach the field names along with the field data while serializing , we can deserialize only a specific field instead of deserializing the entire object. Also the write and read of different fields within an object need not be in the same order, as is the case with all the previous serialization methods. Notice, how cost is written first but is not read first.

**Portable - ShoppingCartItem Implementation**

```
@Override
public int getFactoryId() {
   return 2;
}

@Override
public int getClassId() {
   return 1;
}

@Override
public void writePortable(PortableWriter out) throws IOException {
   out.writeLong("cost", cost);
   out.writeInt("quantity", quantity);
   out.writeUTF("name", itemName);
   out.writeBoolean("stock", inStock);
   out.writeUTF("url", url);
}

@Override
public void readPortable(PortableReader in) throws IOException {
   url = in.readUTF("url");
   quantity = in.readInt("quantity");
   cost = in.readLong("cost");
   inStock = in.readBoolean("stock");
   itemName = in.readUTF("name");
}
```

**Portable - ShoppingCart Implementation**

```
    @Override
    public int getFactoryId() {
       return 2;
    }

    @Override
    public int getClassId() {
       return 2;
    }

    @Override
    public void writePortable(PortableWriter out) throws IOException {
       out.writeLong("total", total);
       out.writeLong("date", date.getTime());
       out.writeLong("id", id);
       Portable[] portables = items.toArray(new Portable[] {});
       out.writePortableArray("items", portables);
    }

    @Override
    public void readPortable(PortableReader in) throws IOException {
       Portable[] portables = in.readPortableArray("items");
       items = new ArrayList<>(portables.length);
       for (Portable portable : portables) {
          items.add((ShoppingCartItem) portable);
       }
       id = in.readLong("id");
       total = in.readLong("total");
       date = new Date(in.readLong("date"));
    }
```

The Factory implementation class is similar to the one in IdentifiedDataSerializable. In Portable, we can additionally provide the class definition through the config class. For objects containing primitive type data with no nullable version, this class definition is not required, since Hazelcast will internally generate this definition. However, for all other cases, we can explicitly provide class definition of objects through the configuration. In the code below, we can see the class definition of `ShoppingCartItem` and `ShoppingCart` added to the config class.

**Portable - ShoppingCartPortableFactory Implementation**

```
public class ShoppingCartPortableFactory implements PortableFactory {
   @Override
   public Portable create(int i) {
      switch (i) {
         case 1: return new ShoppingCartItem();
         case 2: return new ShoppingCart();
         default: return null;
      }
   }
}
```

**Portable - Class Definition Implementation**

```
   Config config = new Config();
      config.getSerializationConfig().addPortableFactory(2, new
ShoppingCartPortableFactory());

   ClassDefinitionBuilder builder0 = new ClassDefinitionBuilder(2, 1);
   builder0.addIntField("quantity").addLongField("cost").addUTFField("name").
   addBooleanField("stock").addUTFField("url");
   ClassDefinition shoppingCartItemClassDef = builder0.build();

   ClassDefinitionBuilder builder1 = new ClassDefinitionBuilder(2, 2);
   builder1.addLongField("total").addLongField("date").addLongField("id")
   .addPortableArrayField("items", shoppingCartItemClassDef);
   ClassDefinition shoppingCartClassDef = builder1.build();

   config.getSerializationConfig().addClassDefinition(shoppingCartClassDef);
   config.getSerializationConfig().addClassDefinition(shoppingCartItemClassDef);
```

We ran the benchmark program with the `com.hazelcast.nio.serialization.Portable` interface and got the following results.

Portable Write performance - average is 65 operations per millisecond
Portable Read performance - average is 54 operations per millisecond
Binary Object Size - average is 386 bytes

Here the binary object size is much larger. This is because we are passing the class definition along with the object data to the parent node. However, on the parent node side, the class definition for each class will be stored only once irrespective of the number of objects of that class at that node.

## PLUGGABLE/CUSTOM SERIALIZATION (EG. KRYO)

Hazelcast also allows you to plug in any custom serialization framework as well. You can have different objects using different types of serializations. This doesn't require class to implement an interface. Because of this feature, you can even implement serialization for class files whose code you are not allowed to change.

**Pros**

- Doesn't require class to implement an interface
- Very convenient and flexible
- Can be stream based or byte array based

**Cons**

- Requires to implement the actual serialization
- Requires to plug and configure

You can implement custom serializations using any of two interfaces, ByteArraySerializer or StreamSerializer. The `ShoppingCart` and `ShoppingCartItem` classes do not need to implement any interfaces in this case. For the sake of this demo, we have used a StreamSerializer. The configurations will be as shown below.

```
public class ShoppingCartKryoSerializer implements StreamSerializer<ShoppingCart> {
   private static final ThreadLocal<Kryo> kryoThreadLocal
         = new ThreadLocal<Kryo>() {

      @Override
      protected Kryo initialValue() {
         Kryo kryo = new Kryo();
         kryo.register(AllTest.Customer.class);
         return kryo;
      }
   };

   @Override
   public int getTypeId() {
      return 0;
   }

   @Override
   public void destroy() {
   }

   @Override
   public void write(ObjectDataOutput objectDataOutput, ShoppingCart shoppingCart)
throws IOException {
      Kryo kryo = kryoThreadLocal.get();
      Output output = new Output((OutputStream) objectDataOutput);
      kryo.writeObject(output, shoppingCart);
      output.flush();
   }

   @Override
   public ShoppingCart read(ObjectDataInput objectDataInput) throws IOException {
      InputStream in = (InputStream) objectDataInput;
      Input input = new Input(in);
      Kryo kryo = kryoThreadLocal.get();
      return kryo.readObject(input, ShoppingCart.class);
   }
}
```

We then need to add this Serializer to the config class as shown below.

```
Config config = new Config();
config.getSerializationConfig().getSerializerConfigs().add(
      new SerializerConfig().
      setTypeClass(ShoppingCart.class).
      setImplementation(new ShoppingCartKryoSerializer()));
```

We ran the benchmark program with the Kryo Serializer interface and got the following results.

Kryo Serializer Write performance - average is 60 operations per millisecond
Kryo Serializer Read performance - average is 51 operations per millisecond
Binary Object Size - average is 210 bytes

## NATIVE BYTE ORDER AND UNSAFE

We now move onto discussing extra config options that are available for serialization.

Setting the native byte array and unsafe options to true, as shown below, enables fast copy of primitive arrays like byte[], long[], etc. in your object.

```
config.getSerializationConfig().setAllowUnsafe(true).setUseNativeByteOrder(true);
```

## COMPRESSION

Compression is supported only by Serializable and Externalizable. It has not been applied to other serializable methods because it is extremely slow (~1000 times) and consumes a lot of CPU. However, it can reduce binary object size from 525 bytes to 26 bytes.

```
config.getSerializationConfig().setEnableCompression(true);
```

## SHAREDOBJECT

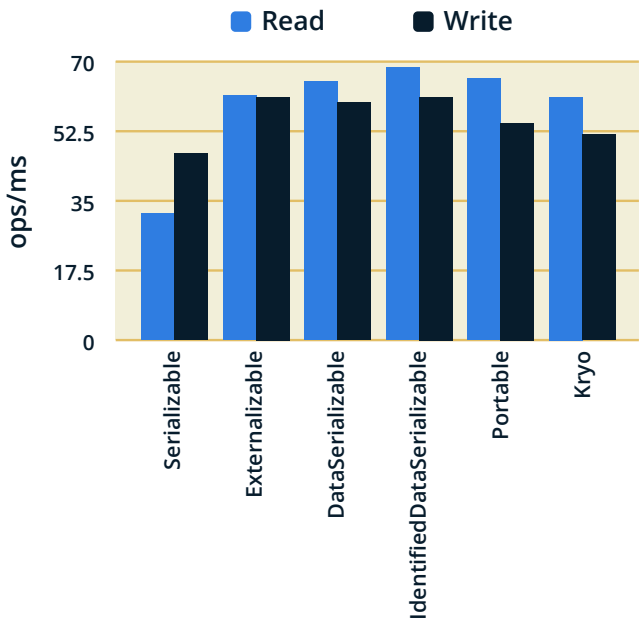
If set true, java serializer will back reference an object pointing to a previously serialized instance. With unshared, every instance is thought as unique and copied separately even if they point to the same instance. Default is false.

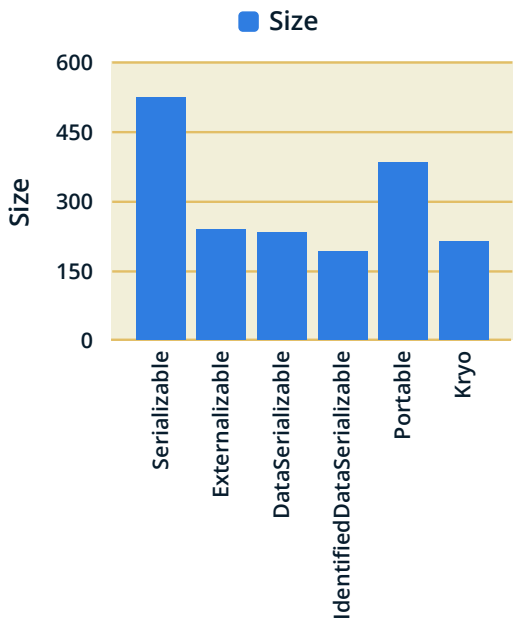
```
config.getSerializationConfig().setEnableSharedObject(true);
```

## COMPARISON

Comparison of all the serialization methods shows that the IdentifiedDataSerializable is best in terms of performance. It has the most number of operations per milliseconds. Also, it generates the smallest size binary objects.

Please note that these benchmark tests are performed on a a single node on a laptop. This is not a full scale benchmark test. However, these tests do give us an idea of how Hazelcast performs with different serialization methods.




hazelcast

350 Cambridge Ave, Suite 100, Palo Alto, CA 94306 USA
Email: sales@hazelcast.com Phone: +1 (650) 521-5453
Visit us at www.hazelcast.com