Naos: Serialization-free RDMA networking in Java

Konstantin Taranov[1], Rodrigo Bruno[2†], Gustavo Alonso[1], and Torsten Hoefler[1]
[1]*Department of Computer Science, ETH Zurich* [2]*INESC-ID / Técnico, ULisboa*

Abstract

Managed languages such as Java and Scala do not allow developers to directly access heap objects. As a result, to send on-heap data over the network, it has to be explicitly converted to byte streams before sending and converted back to objects after receiving. The technique, also known as object serialization/deserialization, is an expensive procedure limiting the performance of JVM-based distributed systems as it induces additional memory copies and requires data transformation resulting in high CPU and memory bandwidth consumption. This paper presents Naos, a JVM-based technique bypassing heap serialization boundaries that allows objects to be directly sent from a local heap to a remote one with minimal CPU involvement and over RDMA networks. As Naos eliminates the need to copy and transform objects, and enables asynchronous communication, it offers significant speedups compared to state-of-the-art serialization libraries. Naos exposes a simple high level API hiding the complexity of the RDMA protocol that transparently allows JVM-based systems to take advantage of offloaded RDMA networking.

1 Introduction

Managed programming languages, such as Java and Scala, are a common vehicle for developing distributed platforms such as Spark [36], Flink [6], or Zookeeper [11]. However, the high level abstractions available in managed languages often cause significant performance overheads. In particular, to exchange data over the network, Java applications are currently forced to transform structured data via serialization, causing a high CPU overhead and requiring copying the data multiple times. While less of an issue in single-node applications, the overhead is substantial in distributed settings, especially in big data applications. Serialization already accounts for 6% of total CPU cycles at Google datacenters [15].

Data transfer with object serialization/deserialization (OSD) is a complex process involving five steps: **graph traversal** to identify all objects that should be serialized; **data transformation** to convert the objects into a byte stream (network-friendly format); **transmission** to send the serialized data over the network; **data traversal** at the receiver to decode the received data; and **object construction** that involves allocating memory and object re-initialization.

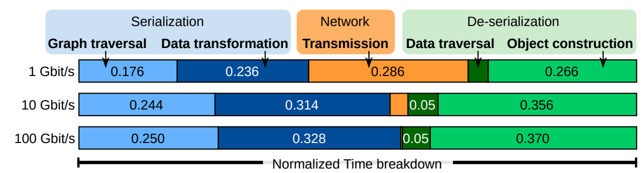


Figure 1: Impact of network bandwidth on OSD time.

To illustrate the CPU overhead caused by OSD, we benchmarked the Kryo [26] serializer and measured its CPU utilization while sending objects over different networks. Figure 1 shows the fraction of time spent on each OSD step for the transfer of an array of 1.28M objects, all of the same exact type. Each object has two fields, each encapsulating a primitive type. Results show that the time spent in OSD increases as networks get faster. For a 10 Gbit/s network, it takes less than 3% of the time to send data over the network, but it takes more than 31%/35% of the time in data transformation/object construction. This discrepancy is even more evident in 100 Gbit/s networks in which the network time drops to less than 0.01% and the time spent on the CPU performing OSD accounts for almost 100% of the transfer time. While networks are getting faster, the pressure is moving away from the network and into the CPU (and memory bandwidth), further aggravating the already well-known CPU-bottleneck problems encountered in distributed applications [22, 32]. Furthermore, existing distributed platforms that heavily rely on OSD are not able to take advantage of faster networks such as RDMA.

With the widespread use of Java in large scale data processing and the increased availability of RDMA, it is time to rethink current OSD techniques so that part of the load of object shifts from the CPU back to the network. In this paper, we aim to develop native runtime support for *serialization-free*

[†]This author was at ETH Zurich during this project.

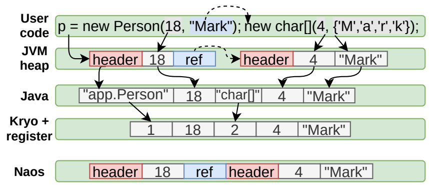
Sender with Kryo

```
buffer = ByteBuffer.allocate(512);
person = new Person(18, "Mike");
kryo = new Kryo();
kryo.register(Person.class);
out = new Output(buffer);
kryo.writeObject(out, person);
connection.write(buffer)
```

Sender with Naos

```
person = new Person(18,"Mike");
connection.writeObject(person);
```

Data format



Receiver with Kryo

```
buffer = ByteBuffer.allocate(512);
connection.read(buffer);
kryo = new Kryo();
kryo.register(Person.class);
in = new Input(buffer);
obj = kryo.readObject(in,Person.class);
person = (Person)obj;
```

Receiver with Naos

```
object = connection.readObject();
person = (Person)object;
```

Figure 2: Serialization, Data format, Deserialization for Kryo and Naos

networking that avoids superfluous memory copies and data transformation by sending objects directly from the source heap into the remote heap. Sending and receiving data without data marshalling enables the use of zero-copy RDMA networking, bypassing not only serialization but the need to copy data. Such a design significantly reduces the pressure on the CPU at the cost of higher data volumes to be transferred, since objects are sent in their uncompressed memory format.

To test and evaluate these ideas, we have developed Naos (Naos stands for Not Another Object Serializer), a library and runtime plugin for OpenJDK 11 HotSpot JVM that allows objects in the source heap to be directly written into a remote heap, avoiding data transformations and excessive data copies. Naos is designed to accelerate object transfers in distributed applications by taking advantage of RDMA communication (although it also supports conventional TCP sockets).

Naos allows applications to directly send objects without employing serialization libraries. Its API requires no type registration nor serialization snippets, guaranteeing developers a close to zero effort when building systems using Naos. Finally, Naos is the first (to the best of our knowledge) library integrating RDMA into JVM allowing the user to communicate on-heap objects transparently, thereby easing the adoption of RDMA networking by JVM-based distributed applications. Our evaluation shows that Naos provides a 2x throughput speedup over serialization approaches for transferring contiguous objects and for moderately sparse object graphs. Naos improves latency-sensitive applications such as RPCs by providing a 2.2x reduction in latency.

Contributions. Naos is the first *serialization-free* communication library for JVM that allows applications to send objects directly through RDMA or TCP connections. Naos unlocks efficient asynchronous RDMA networking to JVM users hiding all the burden of low-level RDMA programming from the users, thereby facilitating the adoption of RDMA. For that, Naos solves several complex design issues such as sending unmodified memory segments across Java heaps without employing intermediate buffers, and interacting with concurrent garbage collection without compromising JVM's memory safety. For the first issue, Naos proposes a novel algorithm that writes objects directly to the remote heap and makes them valid on the receiver's address space (§3.3). For the second one, Naos proposes techniques preventing a concurrent JVM garbage collector from moving unsent objects that may be accessed by RNIC and from accessing unrecovered received objects (§3.2). Finally, Naos enables pipelining communication and serialization, which was previously impossible with the OSD approach (§3.4).

2 Object Serialization

Overview. Many third-party libraries [12, 16, 26] have been developed to perform OSD in Java. Some of them provide Java bindings for popular cross-language OSD approaches (e.g., Protobuf [12]), allowing serializing arbitrary data structures into well-defined messages that can then be exchanged using any network protocol. While remaining independent of programming languages or operating systems, such libraries suffer from low performance [21]. Therefore, JVM-based big-data applications (e.g., Spark, Flink) rely on specialized libraries such as Kryo [26], designed specifically for JVMs.

Figure 2 presents a serialization example of a Java object and its data formats: memory layout (JVM heap), and serialization formats (Java, Kryo). All Java objects start with a JVM-specific header (red) followed by a number of primitive (gray) or reference fields (blue). The object of type `Person` has one primitive `int` field followed by a reference field to a character array (`char[]`). The character array starts with the `length` of the array followed by all characters.

Serializing an object involves traversing the object graph starting from that object and, upon visiting each reachable object, copying all primitive fields into the pre-allocated byte buffer. During native Java serialization, headers are replaced by class descriptors in textual format (`app.Person`) and field references are replaced by the contents of the pointed object. Deserialization follows a similar logic; upon visiting a serialized object, a new object must be allocated, and all primitive fields are copied out of the buffer into the allocated object.

Kryo. Kryo [26], one of the most widely used OSD libraries, addresses some limitations of native Java serialization by requiring manual registration of classes to achieve a more compact representation of the serialized data. Figure 2 shows a serialized data format in Kryo with class registration (Kryo+register). Kryo can represent all primitive types and classes using integer identifiers, thereby reducing the amount of space needed for storing type names. Although the class registration is trivial in this example, this task is cumbersome

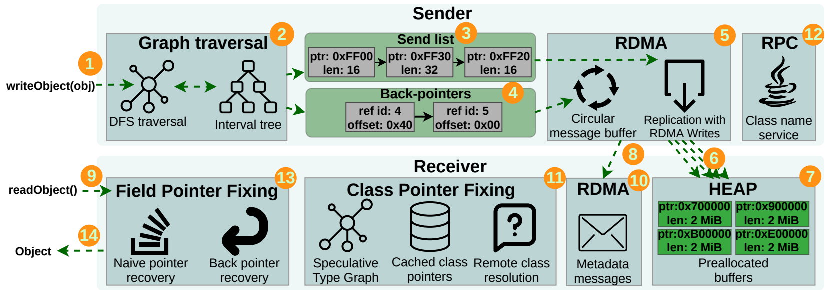


Figure 3: Naos' workflow for sending and receiving a Java object.

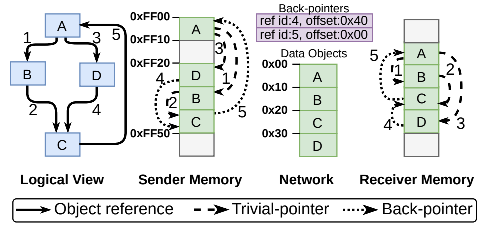


Figure 4: Object views of Naos' graph traversal and pointer recovery.

for applications with hundreds of data types. Compared to Kryo, Naos provides a cleaner interface (see Figure 2) with no need for developer involvement. To send a Java object, one can directly write it (`writeObject`) to the network. The receiver can directly read the object with `readObject`.

Accelerated OSD. To address the overhead of having to transform the data, Cereal [13] and Optimus Prime [24] resort to dedicated hardware accelerators for OSD. These accelerators are co-designed with the serialization format to parallelize the OSD process. Even though their data formats are not portable across different JVMs, their simulation results promise 15x speedup in serialization throughput on average over Kryo at the expense of requiring specialized hardware.

Zero-transformation OSD. The trade-off portability vs. performance is also exploited by the serialization library Skyway [21]. By dropping portability, Skyway manages to partially avoid data transformations and object construction by serializing Java objects in their JVM formats, i.e., the objects are written to communication buffers in the same binary format they are stored in the heap. Like Skyway, Naos sends objects in the JVM heap format, assuming that communicating parties run on the same JVM software. Unlike Naos, however, Skyway is a *serialization library* requiring to copy objects to and from communication buffers. Naos, on the other hand, completely removes the need to explicitly serialize and deserialize objects to send objects between Java heaps even with RDMA. What is more, Skyway's memory management prevents the use of RDMA networking (§4).

Naos integration and applicability. *Naos is not a serialization library*. Naos only covers end-to-end transfers (see Table 1) and cannot replace OSD in systems that do not use it for communication (e.g., for writing objects to disks). Naos has been primarily designed for future systems that want to take advantage of serialization-free zero-copy RDMA networking.

In several existing Java frameworks the main obstacle to using Naos is that some of these systems do not consider the possibility to send objects without serialization. For example, Spark and Hadoop completely decouple serialization from communication: their serialization modules are designed to serialize objects only to files, and their shuffle modules are designed to communicate only files. Such file-centric design simplifies inter-node communication, as processes can share file descriptors instead of sending data, and helps to reduce memory usage by dumping data to disks. However, it makes integrating Naos very difficult. For such use-cases, conventional OSD libraries are a better fit than Naos if a redesign for true zero-copy is infeasible.

Table 1: APIs of Naos RDMA.

API	Description
`void writeObject(Object)`	Blocking send of a single object
`Object readObject()`	Blocking read of an object from heap
`boolean isReadable()`	Check whether an object can be read
`long writeObjectAsync(Object)`	Nonblocking send of a single object
`int waitHandle(long)`	Wait for a send request to complete
`int testHandle(long)`	Tests completion of a send request

3 System Overview

Naos allows Java applications to send/receive objects directly through RDMA or TCP connections. Naos uses a collection of algorithms and data structures to efficiently transmit large complex data structures. Figure 3 presents a graphical overview of Naos' workflow, including the main algorithms and data structures. An object transfer starts with a `writeObject` 1 triggering a DFS graph traversal 2 (§3.1). During the traversal, pointers to already visited objects are detected using an interval tree. After the traversal, both the objects 3 and metadata 4 are sent over the network using RDMA 5 (§3.2). Naos uses a circular message buffer to send metadata 8 and writes objects directly to the remote heap 6. Upon reception of the data and metadata, the receiver starts recovering (§3.3) the object graph by fixing class pointers 11 and field pointers 13. Once pointers are fixed, the head of the object graph is returned 14 to the caller of `readObject` 9.

The `writeObject` call in Naos is blocking, that is, the call returns once the object transmission is completed. It ensures that the object is received by the destination. In contrast to the classical TCP/IP semantics, all RDMA operations are executed asynchronously by design, allowing overlapping computation with communication. Naos also provides a nonblocking `writeObjectAsync` call enabling asynchronous communication for RDMA connections (§3.2). The nonblocking call initiates the send operation but does not fully complete it.

Instead, it returns a request handle, that is used by a user to wait for the completion using `waitHandle` call or to verify whether the request is completed using `testHandle` call.

	The length of the send list				Traversal time (us)			
Structure	(1-0-0)	(1-1-0)	(1-2-0)	(1-1-1)	(1-0-0)	(1-1-0)	(1-2-0)	(1-1-1)
BFS	1	2048	3072	3072	42	194	315	271
DFS	1	2	2	1	42	57	74	76

Table 2: Graph traversal of the object *array*.

3.1 Object Graph Traversal

Java objects can contain reference fields pointing to other Java objects and therefore, when an object is passed as an argument to `writeObject` 1, all objects reachable from it need to be sent. To find all objects reachable from a particular object, Naos traverses the object graph 2 in Depth-First-Search (DFS) order. Figure 4 illustrates a simple example of an object graph's (*Logical View*), sender memory layout, format sent over the network, and receiver memory layout. The *sender memory* starts at address `0xFF00` and all objects occupy 16 bytes. Edges are numbered according to DFS order.

When an object is visited for the first time, it is included in the *Send list* 3: a list of memory blocks that will be sent over the network. Each memory block has two elements: the starting virtual address, and the length. The send list contains objects ordered according to DFS order, and the objects are sent in this order over the network. Naos also merges the memory blocks that are adjacent in the send list to reduce its length. For that, during traversal Naos checks whether a new visited memory block is a continuation of the last block of the send list: if yes, then Naos increases the length of the last block, otherwise, Naos adds a new block to the list. The resulting send list is presented in 3, which contains three elements: for object *A*, for objects *B* and *C* as they are adjacent in memory and in DFS order, and for object *D*.

DFS vs BFS traversal. Even though Skyway [21] uses BFS traversal for serialization, Naos exploits DFS due to the fact that Java objects are constructed in DFS order (i.e., a JVM first allocates memory for an object and then recursively for all its fields). Thus, DFS traversal has better memory locality that can be illustrated by traversing an object *array* from the following code snippet. Let us consider a class *Person* that has different graph structures denoted as (L0-L1-L2), where L*i* is the number of objects on the level *i* of the object graph (e.g., the object in Figure 4 has structure (1-2-1)).

```
Person[] array = new Person[1024];
for(int i=0; i<1024; i++)
  array[i] = new Person();
```

Table 2 reports the length of the send list after BFS and DFS traversals and corresponding traversal time for several object graphs. The data shows that for complex graph structures DFS provides much shorter send lists and faster traversal time.

Back-pointers. Naos sends objects directly from one heap to another. As a result, objects are sent containing pointers that are valid only in the sender address space, but not in the receiver's. Naos addresses this problem by sending extra metadata along with data objects, which is used by the receiver to efficiently recover the pointers (§3.3).

Naos is designed to send as little metadata as possible. The metadata contains a 24-byte header with object and metadata sizes and, if present, pointers to already visited objects 4. These pointers are redundant edges after building a spanning tree over the object graph using DFS. We call them *back-pointers* since they always point to already visited objects in the send list (see Figure 4). For each back-pointer, a reference identifier representing the order by which the reference was visited in DFS order, and an offset within the send list where this reference should point to are sent to the receiver as metadata. In our example in Figure 4, only references 4 ($D \rightarrow C$) and 5 ($C \rightarrow A$) are sent.

All edges of the spanning tree (we call them *trivial-pointers* for simplicity) can be automatically inferred during a DFS traversal in the receiver (§3.3). This allows Naos to send no information about *trivial-pointers* resulting in a massive reduction of metadata sent over the network. Note the graphs without cycles do not contain back-pointers, which covers the vast majority of the most popular Java data structures.

Back-pointer/Cycle detection. To detect pointers to already visited objects (i.e., back-pointers), Naos uses a *memory interval tree* that keeps tracks of all visited memory intervals during DFS traversal. The interval tree is implemented using a red-black tree, which is selected over a hashtable (as Java and Kryo do) for two reasons. First, for large data structures, the hashtable grows (one entry per visited object) to large sizes and will lead to expensive lookups due to hash collision. Second, references to already visited objects are very rare and references pointing to objects in nearby memory positions are common in most Java popular data structures. Therefore, an interval tree, in most cases, contains a few large memory intervals, thereby ensuring fast lookups. We further optimize our interval tree by providing different fast paths.

Algorithm 1 presents how Naos decides whether a particular object o has been already visited. Two helper variables are used: *curr* points to the last node inserted into the tree; *next* points to the tree node that follows *curr* in the tree. All

Algorithm 1 Was object o already visited?

if $o.addr = curr.addr + curr.len \wedge o.addr \neq next.addr$ **then**
  $curr.len \leftarrow curr.len + o.size$ ▷ hot-path
  return $false$
if $o.addr > curr.addr \wedge o.addr < next.addr$ **then**
  $curr \leftarrow tree.insert\_before(next, o)$ ▷ warm-path
  return $false$
$node, success \leftarrow tree.insert(o)$ ▷ cold-path
if $success$ **then**
  $curr \leftarrow node$
  $next \leftarrow curr.next()$
  return $false$
return $true$ ▷ is a back-pointer

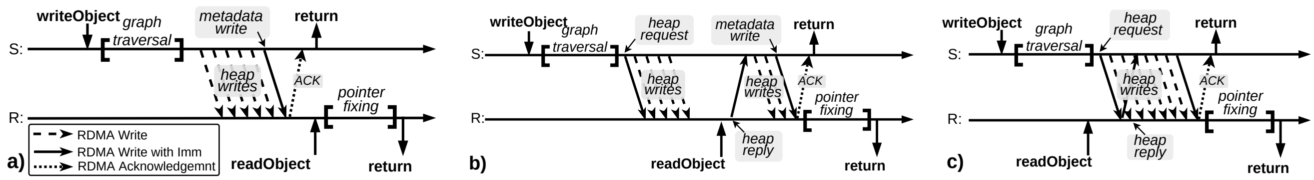


Figure 5: Blocking communication mechanism for three scenarios: **(a)** the sender can fit data to the pre-allocated receiver heap; **(b,c)** the sender needs to request extra heap memory. The receiver was not ready to receive data in **(b)**, and it was ready in **(c)**.

tree nodes keep an initial address *addr* and its length *length*. If the object's address is adjacent to the last memory interval inserted into the tree, the insertion is performed in *O(1)* time (*hot-path*). If the memory pointer is higher than the current tree node and lower than the *next* tree node, then insertion is performed in *O(1)* (*warm-path*), unless the tree needs to be re-balanced, taking *O(log(n))* time . Otherwise, the memory pointer is inserted in the tree in *O(log(n))* time (*cold-path*).

As a comparison, Skyway does not use complex structures for cycle detection and simply extends the JVM header of Java objects by 8 bytes. Even though it ensures that the newness of an object can be be checked in *O(1)*, it results in a 15.4% increase in memory usage [21].

3.2 Network exchange of on-Heap Objects

Naos adds native RDMA communication to JVM without compromising JVM's memory safety. Naos' interface does not expose explicit RDMA access to the remote or local heap memory. Instead, its API allows only sending and receiving Java objects, hiding all the burden of low-level RDMA programming from the user. Internally, though, Naos fully relies on efficient one-sided RDMA communication to completely avoid redundant data copies. Naos also supports TCP for sending objects directly from its heap, but the use of RDMA requires overcoming peculiarities of managed languages such as concurrent garbage collection.

Blocking RDMA protocol. This section describes the *blocking* RDMA protocol for a single connection. All connections are handled independently and do not share resources. The core idea of Naos RDMA is that the receiver pre-registers buffers of fixed size in its heap and registers them for RDMA Write access. The sender uses RDMA Writes 6 to write the objects from its local heap directly to the known reserved buffers in the remote heap 7. The metadata is sent separately using a circular buffer 8 for RDMA messaging [8, 23].

The protocol allows the sender to start writing memory to the remote heap even if the receiver did not call `readObject`, as illustrated in Figure 5(a). The sender can continue writing the data while it has enough free remote memory. Once the sender completes writing all objects to the remote heap using RDMA, it sends a separate completion message with metadata via the circular message buffer 8. The remote circular buffer is filled using *RDMA Write with immediate data*, which generates a completion event on the receiver after the write completes. The sender can unblock from sending once it receives an acknowledgment from the network indicating that all data has been written to the receiver. The acknowledgment is generated by the network and does not require the receiver's interaction. The receiver fetches the received object when it calls `readObject`, after all pointers are recovered (§3.3).

Sender's heap management. Naos utilizes *object pinning* to prevent a JVM garbage collector (GC) from moving objects until they are fully transmitted by the RNIC. Object pinning is already offered by some garbage collectors, such as Shenandoah [9]. Shenandoah is a high-performance GC that is supported by upstream OpenJDK. Besides object pinning, Naos also utilizes Shenandoah's memory allocator, that maintains the heap as the collection of fixed size Shenandoah regions. To pin and unpin objects efficiently, Naos pins whole Shenandoah regions containing the affected objects instead of pinning individual objects. During a send request, Naos pins and remembers all affected Shenandoah memory regions. Once the request completes, Naos unpins the regions associated with the request. Shenandoah allows pinning a region multiple times, and each region needs to be unpinned as many times as it has been pinned, thereby successfully preventing Naos from accidentally unlocking the GC for unsent objects.

RNICs cannot simply send data from any buffer and communication buffers must be registered at the RNIC*. Thus, the sender must register the memory addresses of all objects it needs to send. However, RDMA memory registration is an expensive process that may take hundreds of microseconds for a single buffer [14, 20, 30]. Therefore, naive registration of all objects from the send list may completely cancel all performance advantages of RDMA. Naos addresses this issue by registering large fixed-size memory regions (i.e., Shenandoah regions) where the objects are allocated. It enables reusing a single memory registration for all objects stored in it, exploiting spatial locality. Naos also caches memory registrations to reuse them later for future sends, exploiting temporal locality.

Receiver's heap management. When the sender runs out of the remote buffers for writing, it sends a request to the receiver to register more on-heap memory, as illustrated in Figure 5(b,c). Thus, the sender can block until the receiver

*Modern RNICs support implicit on-demand paging (ODP) [17] that removes the need to register buffers. In our preliminary experiments, however, ODP performed worse than conventional explicit memory registration.

replies with new heap buffers, as in Figure 5(b). However, when the receiver is ready to receive data it can immediately reply to the heap request and do not obstruct the sender as in Figure 5(c). The receiver can reply to heap requests when it calls `readObject` or `isReadable`. During these calls, Naos checks for received requests by polling completion events from the RNIC. The process of handling requests is invisible to the caller, which hides the complexity of the underlying protocol from the user.

Upon receiving a heap request, the receiver allocates a new Java byte array buffer of fixed size inside the Java heap and *registers its payload* for RDMA Write access and replies with the RDMA address of the registered buffer. To prevent the GC from moving the reserved on-heap buffers, Naos utilizes *object pinning* offered by Shenandoah [9]. Importantly, the sender writes data to the *payload* of the pre-allocated byte array as it prevents the GC from reading invalid data. The main reason for that is that the unrecovered received objects have invalid class and object pointers (§3.3). Thanks to this enclosure, the GC observes only the array and skips reading objects stored in the payload.

The sender fills the remote buffers in the order it received them from the receiver, constituting a queue of remote heap buffers. Since pre-registered RDMA heap buffers are of fixed size, the sender is not always capable of fully utilizing them. To address this issue, the sender informs the receiver about how many bytes were unused in each *finalized* heap buffer by sending *heap truncate request*. A buffer becomes finalized when the sender jumps to the next buffer in the queue. After receiving the data, the receiver revokes RDMA access to finalized buffers and then unpins them to enable the GC for received objects. It also deallocates unused memory of the finalized buffer and removes the array header to make all received objects visible to the GC.

Nonblocking object sending. The main difference between the blocking `writeObject` and the nonblocking `writeObjectAsync` is that the later returns right after the dispatching *metadata write* request to the device. The nonblocking call submits all communication requests to the RNIC but does not wait for a network acknowledgment. Instead, Naos returns a request handle that can be used by an application to confirm the delivery of the object using `testHandle` call. Compared to the blocking call, Naos prevents the GC from moving affected objects even after the call returns. Naos *pins* the affected objects before exiting the JVM, and unpins them later once the corresponding acknowledgment is received.

Naos TCP. Naos supports sending objects directly from the heap using TCP as well. Unlike RDMA connections, a traditional TCP socket connection has a single datapath. Thus, to send the objects to the remote heap, the TCP sender first writes the metadata to the socket and then all elements of the send list. The receiver first reads metadata to a temporal buffer from its socket, then, to avoid redundant data copies, it directly reads the data from the socket to the heap. For that, it allocates a byte array buffer of the required size inside the Java heap, and then reads the data from the socket to the payload of the allocated buffer.

Network buffering. Naos is designed to send data directly from the heap without intermediate buffering. However, the size of a JVM object can be as small as 24 bytes. Thus, a highly sparse object graph can result in a lot of small writes to the network, which can significantly reduce the network performance. To address this issue, Naos may buffer small objects before sending them to the remote heap. Large objects are still sent directly from the heap. Naos sends buffered objects once it batches enough bytes to utilize the network, or when a large object needs to be flushed to preserve DFS object order (§3.1).

An alternative approach is to use scatter-gather capability of RNICs [18] for RDMA networking and scatter-gather I/O for TCP sockets. The scatter-gather networking enables building a network message from multiple buffers without intermediate buffering. The current version of Naos does not implement it, but it is an interesting direction for future research.

Memory safety of Naos. Naos uses reliable transport to ensure the delivery of transmitted data. Naos materializes only fully received objects, which prevents returning partially received objects from a faulty sender. Faulty sends can be detected during graph recovery from the network errors provided by the reliable transport. If an error is detected, the receiver revokes RDMA access to pre-allocated buffers and deallocates the unused memory.

Naos' implementation follows all security advice related to RDMA networking [25, 31], therefore, we believe that Naos does not open security breaches. In particular, the pre-allocated heap buffers are not shared between connections preventing remote JVMs to access buffers of each other. In addition, each sender registers its heap only for local read access preventing other remote JVMs to access it. Finally, remote read access is always disabled, and Naos only temporarily enables write access to pre-allocated in-heap buffers, which are private for each sender. Once the in-heap buffer is full, the write access is revoked.

For compatibility between communicating applications, Naos requires that communicating JVMs have the same memory layout of in-heap objects. This can be achieved by running the same JVM with the same settings including GC.

3.3 Object Graph Recovery

Naos sends unmodified memory segments from one heap to another. As a result, objects are sent containing pointers that are valid only on the sender address space, but not on the receiver's. Naos' graph recovery algorithm overwrites these pointers making them valid on the receiver's address space. Java objects have two types of pointers: **class pointers** and **object pointers**. Class pointers point to JVM-internal data structures that describe Java types. Object pointers are

Algorithm 2 Object Graph Recovery

```
1: buffer                                   ▷ the buffer with received objects
2: refid ← 0                                ▷ the number of traversed references
3: offset ← 0                               ▷ current offset in the receive buffer
4: stack.push(new field(), new hint())      ▷ push dummy field and hint
5: while stack.is_not_empty() do
6:     field, hint ← stack.pop()
7:     FIX_FIELD_POINTER(field, hint)
8:     refid ← refid + 1
```

Phase 1 – Fix Field Reference

```
9: procedure FIX_FIELD_POINTER(field, hint)
10:    if refid = cur_back_pointer.id then                ▷ a back-pointer
11:        field.ptr ← buffer + cur_back_pointer.offset
12:        cur_back_pointer ← get_next_back_pointer()
13:    else                                                ▷ a trivial-pointer
14:        obj ← (obj)(buffer + offset)
15:        field.ptr ← obj
16:        FIX_CLASS_POINTER(obj, hint)
17:        ITERATE_FIELDS(obj, hint)
18:        offset ← offset + obj.size
```

Phase 2 – Fix Class

```
19: procedure FIX_CLASS_POINTER(obj, hint)
20:    if hint.rem_class = obj.class then
21:        // hint is correct, do nothing                 ▷ hot-path
22:    else
23:        if class_cache.contains(obj.class) then
24:            new_hint ← class_cache.get(obj.class)      ▷ warm-path
25:            hint.update(new_hint)
26:        else
27:            new_hint ← class_service(obj.class)        ▷ cold-path
28:            class_cache.put(obj.class, new_hint)
29:            hint.update(new_hint)
30:    obj.class ← hint.loc_class
```

Phase 3 – Iterate Fields

```
31: procedure ITERATE_FIELDS(obj, hint)
32:    for field, field_hint in hint.fields do
33:        stack.push({obj + field.offset, field_hint})
```

reference fields that point to other on-heap Java objects.

Naos uses a recovery approach different from the one used in Skyway [21]. Since Skyway copies objects to communication buffers, it can afford modifying data before sending. Thus, Skyway simply replaces class pointers with integers (as Kryo does) and object pointers with their relative offsets within the communication buffer. Such design allows the receiver to simply replace class integers with corresponding class pointers and relative object offsets with corresponding absolute addresses. Unlike Skyway, Naos sends objects directly from the heap using RDMA requiring more sophisticated algorithm for pointer fixing in return for not requiring data copying.

Algorithm 2 describes the Naos' graph recovery approach that starts with a DFS traversal of the object fields (lines 5-8). The traversal is initialized by pushing a *dummy* field pointing to the first received object. The graph recovery terminates when the DFS *stack* is empty. At that point, all pointers are valid in the receiver's heap and the first object can be safely returned to the user.

Fixing Field References. For every object field, the algorithm applies *FIX_FIELD_POINTER* procedure, which investigates whether the tested reference is a *back-pointer* or a *trivial-pointer* by checking whether the received metadata contains the current reference ID (line 10). For *back-pointers*, the *offset* associated with the current pointer is used to fix the reference. If the reference is a *trivial-pointer*, the new memory address can be determined by just using the current *offset* in the *receive* buffer (line 14). For a trivial-pointer, the next step is to fix the class field of the pointed unvisited object (line 16). Note that Naos sends no metadata for trivial pointers, since the sender and the receiver traverse the graph in the same DFS order, providing a significant reduction in metadata size.

Fixing Class References. Updating class pointers is a particularly expensive operation if not designed carefully, since the class pointer needs to be fixed for every object. To achieve high performance, Naos proposes a 3-way approach:

Class Service (*cold-path*) is an RPC service 12 that is started upon creation of a Naos connection. Once a receiver needs to determine the class of a particular sender's class pointer, it issues an RPC request to the sender to translate the pointer to the full class name. The full class name can be used locally to query local JVM internal data structures.

Class Map (*warm-path*) is a per-connection table that caches all class translations. However, accessing a table for every object reference still produces a large overhead, especially in large graphs. To overcome this limitation, Naos proposes the use of Speculative Type Graphs (STG), a type of polymorphic cache inspired by [10].

STG (*hot-path*) is a data structure that dynamically captures type relations in the object graph, providing a translation *hint* for each class pointer. Each STG hint caches: i) a translation between a local and remote class pointer; ii) class description including object fields; iii) pointers to other hints for each field allowing to build hints recursively (lines 32-33). Using STG, Naos can speculate on the type of a particular object using a hint. If the hint is correct the class translation and retrieval of a class descriptor takes *O(1)* time (line 20). Speculation might fail due to type polymorphism in Java (line 22) and, in that case, the cache is used for resolving the class pointer and the STG is updated (line 25) with the new translation hint. In practice, however, most data structures have very regular type graphs allowing the STG to guess correctly most of the times.

After the class pointer of an object is fixed, Naos iterates all its reference fields (line 17). Naos utilizes object's class pointer translation hint to create translation hints for its reference fields (lines 32-33), which are then pushed into the *stack* together with the corresponding object reference.

3.4 Overlapping network and graph traversal

An important disadvantage of conventional serialization approach is that it does not support overlapping serialization and communication: an object must be fully serialized before sending it over a network. Similarly, the receiver cannot start

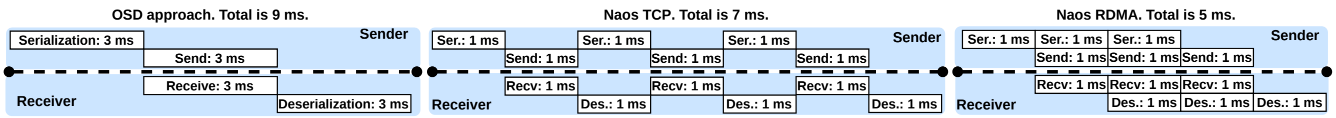


Figure 6: The communication benefits of Naos' pipelining compared to the conventional OSD approach.

deserialization unless it receives all the data (see Figure 6). As a result, applications can suffer from high end-to-end latency for large object graphs.

Naos supports pipelining graph traversal with communication on the sender and pointer fixing with object receiving on the receiver. Both Naos TCP and Naos RDMA benefit from pipelining as it allows the receiver to start pointer fixing of *partially received object graphs*, thereby reducing end-to-end latency. Using offloaded RDMA communication, Naos RDMA can continue traversing the graph after submitting write requests to the RNIC, thereby overlapping communication and graph traversal on both the sender and the receiver.

Pipelining in Naos is implemented by pausing the object traversal and sending partial graphs to the remote heap. A partial graph contains only objects and back-pointers found at a given traversal stage. The receiver can read the partial graph and start pointer fixing. Once all received objects are traversed, the receiver reads the next fragment of the graph.

Figure 6 illustrates how Naos with pipelining improves communication latency of large object graphs compared to the OSD approach. The OSD approach cannot break serialization of a single graph, which results in 9 ms latency. Naos TCP can send partially traversed graphs reducing the latency by 2 ms, but cannot overlap computation with communication. Naos RDMA enables overlapping communication and graph traversal, which reduces the latency by another 2 ms.

4 Evaluation

We evaluate the performance of Naos[†] and compare it with Java, Kryo, and Skyway[‡] serialization engines using four different classes of workloads. First, the performance of Naos is studied by transferring data structures that are commonly used in distributed applications. The goal is to measure the performance benefits of the different techniques proposed in Naos and the trade-offs involved depending on the shape of object graph. In addition, it also shows the impact of using RDMA instead of TCP. Second, we study the role of data streaming and pipelining in OSD performance. Then, we show results for integrating the Naos library into Apache Dubbo [2], a high-performance RPC framework developed in Java, to show the impact of Naos on RPC workloads. Lastly, we use a map-reduce implementation of PageRank to measure the performance of Naos for data processing workloads.

Experimental setup. All experiments were performed on a cluster of 4 nodes interconnected by 100 Gbit/s Mellanox ConnectX-5 NICs. Each node is equipped with an Intel(R) Xeon(R) CPU 6154 @ 3.00 GHz and 384 GB of RAM.

Implementation details. Naos is implemented and tested for OpenJDK HotSpot 11.0.6 [1], a widely-used production JVM. Naos does not require changes to the internals of the JVM and is implemented as a JNI plugin and a Java-level library that allows users to write objects directly to TCP and RDMA connections. Naos TCP provides constructors to create a Naos connection from TCP connections of various network libraries (e.g., `java.net.Socket`). Naos RDMA does not rely on existing JVM RDMA libraries and fully implements a specialized RDMA network library including an API to create and connect RDMA endpoints. Our plugin is implemented in Java and C++ and depends on: *libibverbs*, an implementation of the RDMA verbs, and *librdmacm*, an implementation of the RDMA connection manager.

RDMA communicators for Java and Kryo serializers have been implemented using Disni [27] RDMA library, a high-performance Java RDMA library that encapsulates native C RDMA verbs API. The Disni library is used by Java applications such as Spark [19], Crail [29], and DaRPC [28]. Note that Skyway cannot be used with existing RDMA libraries, including Disni, as these libraries can only work with specialized off-heap memory residing outside of the Java heap memory, whereas Skyway requires the memory buffers reside inside the heap memory to deserialize objects. These limitation stems from the fact that garbage collection can move on-heap buffers while they are being accessed by the RNIC.

In all experiments, the JVM was configured with default parameters and enabled Shenandoah garbage collector as it is the only collector that is currently supported by Naos. Shenandoah was configured with 32 MiB memory regions. Naos was configured with 20 MiB receive buffers. If not stated differently, Naos and all serialization algorithms were deployed without graph cycle detection and with no pipelining (§3.4).

4.1 Serializing Java Data Structures

The performance of OSD approaches is measured using three data structures that are among the most common serialized data structures in real-world workloads deployed in platforms such as Spark, Hadoop, and Flink: a) an array of `float` primitive types, which is common for machine learning workloads;

[†]The source code is available at https://github.com/spcl/naos/.

[‡]We could not compare with the original Skyway as it is not open-source. Therefore, we re-implemented Skyway following the instruction provided in the paper [21]. Note that we did not extend object headers by 8 bytes for cycle detection and simply evaluated Skyway without cycle detection.

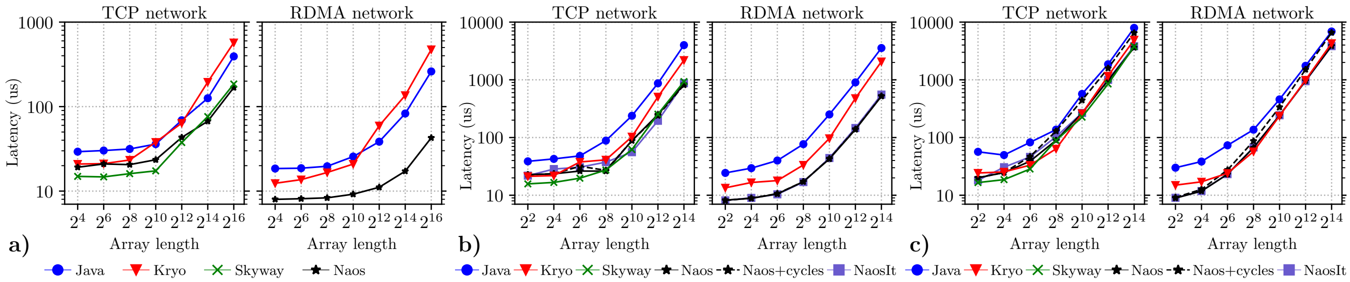


Figure 7: Latency in us for **a)** an array of floats **b)** an array of Points **c)** an array of Pairs. The y-axis is in log scale.

b) an array of `class Point` containing only two primitive types, which represents a 2D Euclidean point; c) an array of `class Pair` containing an integer and a char array, which represents a key-value pair, in many algorithms such as Word Count. In our experiments the char array had length 5, the average word length in the English language.

Benchmarks are carefully designed to guarantee the optimal configuration of all serializers. In particular, for Java, Kryo, and Skyway, all buffers are pre-allocated with the correct size to avoid re-allocation and memory copies during the serialization process. Besides, all types were pre-registered in Kryo to guarantee maximum data format compression. Measurements are taken after a JVM warmup (of at least 100 ms) until convergence of the JIT compiler to achieve maximum performance. All experiments run in complete isolation for several seconds and the aggregated statistics are reported.

Latency. Figure 7 shows the average latency of transferring the aforementioned data types with increasing their size.

Naos performs excellently for contiguous data structures such as the array of float, as it can send them from the heap without making extra copies and using fewer RDMA requests. For comparison, Kryo, Java, and Skyway must first serialize objects to a dedicated send buffer. RDMA-Naos' latency can be as small as 8 us, which is at least a 2x and a 2.4x improvements over Kryo and Java serializers, respectively, for small arrays, and at least a 4.5x for large arrays. For example, Naos RDMA needs only 42 us to send 2^{16} floats, whereas serialization approaches need at least 190 us.

Naos RDMA has lower latency than Skyway, however, Skyway performs better than TCP-Naos for small arrays because of two reasons. First, Naos buffers small objects (less than 256B) to better utilize the network (§3.2). Second, Naos TCP allocates on-heap memory after data arrives, whereas Skyway has all buffers preallocated in our experiments. Both reasons give an advantage to Skyway over Naos TCP for small arrays. For large arrays, Naos TCP provides a 9.1% reduction in latency over Skyway as it incurs fewer data copies.

An array of float is the simplest object graph for graph traversal as it contains a single contiguous object. An array of Point, however, is non-contiguous in memory as this array contains references to objects of class Point, which are 32 bytes each. Nonetheless, Naos provides a 2x and a 4x improvements on average over Java and Kryo for RDMA networks, even with cycle detection enabled (*+cycles*). *Naos+cycles* benefits from our hot-paths of Algorithm 1 as the JVM tends to collocate objects in memory even for the potentially sparse object graphs. The experiment shows that moderately sparse graphs with small objects are not an issue for Naos.

An array of Pair is even sparser graph than the array of Point, as the class Pair has more references than the class Point. Naos RDMA still achieves the lowest latencies for all sizes. However, with cycle detection, Naos' traversal is slower for long arrays is slower compared to Kryo. The main problem is that Naos sends more data than conventional serializers since it needs to send a JVM header of 16 bytes for each Java object. We conclude that Naos does not always provide lower latency compared to conventional OSD approaches and that its performance depends on sparsity and the number of traversed objects.

A shortcoming of Skyway's and Naos' data format is that they do not compress arrays with references and are forced to send long arrays with (invalid) references, whereas Kryo can encode this information in few bytes. To address this issue, we designed a specialized send call for Naos, namely *NaosIt*, that sends only objects stored in an array. The receiver of such compressed message creates a new array and then fills it with received objects. *NaosIt* reduces the size of communicated data, but requires extra memory allocation on the receiver. Overall, NaosIt provides a small improvement over Naos, as the experiments are performed on 100 Gb/s network. Such compression would be more beneficial for slower networks.

CPU and network costs. To show the key differences between Naos networking and the traditional OSD approaches, Table 3 shows the time breakdown of transferring various data structures and their network cost. Naos as a serialization-free approach always has zero cost for serialization and deserialization. Naos' graph traversal time is included in the send time. The OSD approaches with RDMA has zero receive cost as the data delivered directly to pre-allocated receive buffers by the RNIC. Naos, on the other hand, has non-zero cost as the receive time includes the graph recovery.

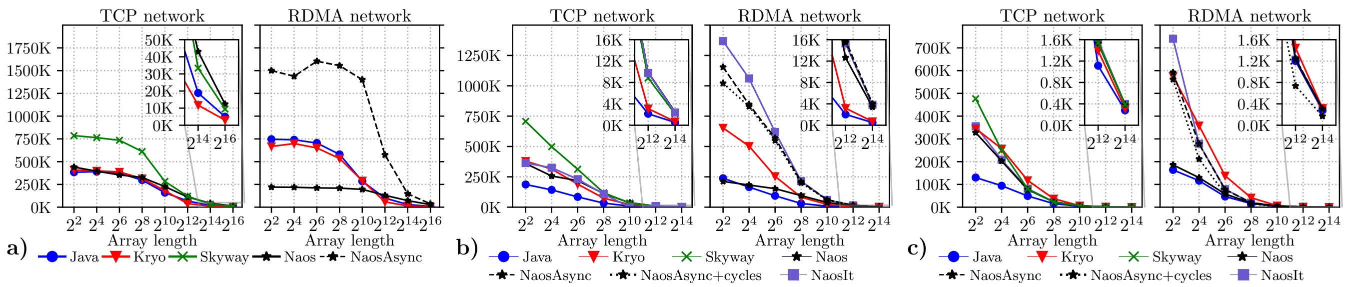


Figure 8: Throughput in objects/sec for **a)** an array of floats **b)** an array of Points **c)** an array of Pairs.

	TCP				RDMA				
Test	Ser.	Send	Receive	Deser.	Ser.	Send	Receive	Deser.	Size (B)
Array of native float with 8192 elements									
Java	15-18	6-9	6-8	17-24	14-17	0-1	0	15-20	32795
Kryo	24-29	7-10	6-8	26-31	24-27	0-1	0	25-86	32772
Skyway	2-3	7-9	7-8	0-1	NA	NA	NA	NA	32792
Naos	0	7-9	16-52	0	0	10-11	0-1	0	32792
Array of class Point with 1024 elements									
Java	109-116	5-7	4-6	94-102	112-119	0-1	0	113-121	14469
Kryo	44-47	4-6	2-3	36-39	43-47	0-1	0	38-41	8132
Skyway	23-26	6-8	6-8	10-11	NA	NA	NA	NA	28696
Naos	0	22-26	27-76	0	0	25-26	13-15	0	28696
NaosIt	0	22-23	28-39	0	0	24-25	15-17	0	24576
Array of class Pair with 1024 elements									
Java	216-664	7-19	10-12	208-222	211-223	0-1	0	217-230	30864
Kryo	135-231	5-10	6-8	79-83	135-148	0-1	0	80-83	18436
Skyway	149-154	9-13	15-31	23-24	NA	NA	NA	NA	61464
Naos	0	161-168	84-137	0	0	199-206	34-39	0	61464
NaosIt	0	159-164	109-138	0	0	200-206	36-40	0	57344
	CPU sender		CPU receiver		CPU sender		CPU receiver		Network

Table 3: CPU time breakdown (in us) and Network cost for transferring arrays. Percentiles 5 and 95 are reported.

Object serialization in TCP experiments takes longer than for RDMA. The difference comes from the fact that in TCP experiments the data is serialized to on-heap buffers, which can be affected by the GC, whereas RDMA requires data to be serialized to off-heap buffers, that are invisible to the GC.

Java and Kryo for RDMA have the same send cost which is the cost of submitting offloaded RDMA request to RNIC. Blocking Naos RDMA has higher cost to send as it needs to wait for a network acknowledgment to finish sending.

For all data types, Naos RDMA shows at least a 2x reduction in CPU time for receiver over Kryo and Java. The main reason is that conventional serialization libraries need to allocate and initialize memory for each received object. Naos does not construct objects and only fixes pointers in the received data. For senders, however, Naos is better at reducing CPU cost for simple graphs such as arrays of floats and points. Note that Naos TCP has a longer receive time than Skyway as it needs to allocate receive memory, whereas Skyway worked with pre-allocated buffers in our experiments.

The network cost of Naos and Skyway increases with the number of transmitted Java objects. For an array of floats, therefore, the size of the transmitted data is approximately the same for all approaches. On the other hand, for an array of Points or Pairs, the network cost of Naos is about 2x higher in comparison with Java and about 3.5x over Kryo. Kryo has the lowest network costs as it replaces the class descriptors with integer identifiers significantly compressing object graphs. Naos and Skyway have the same network cost as they have the same data format, but our NaosIt provides a reduction in the network size for array containers.

Throughput. In this experiment, senders continuously send objects to the receiver. For RDMA approaches with serializers, we provide at the sender and the receiver a large number of send and receive buffers to enable asynchronous communication so that the sender can start serializing and sending the next object without the need to wait for the completion of the previous requests.

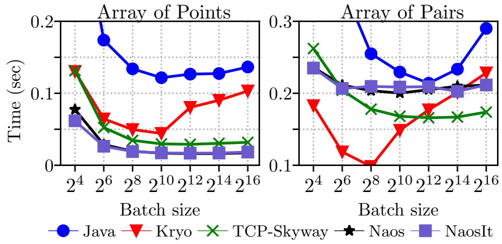
Figure 8(**a**) shows that Naos TCP was not able to significantly outperform Skyway for small arrays, as the throughput of Naos was mostly limited by the receive buffer allocation, whereas Skyway, with pre-allocated memory, achieved 750K req/sec. For arrays larger than 2^{12} elements, however, Naos TCP outperforms Skyway as the cost of data copies at the sender overwhelms the cost of memory allocation at the receiver, showing the advantage of our zero-copy design.

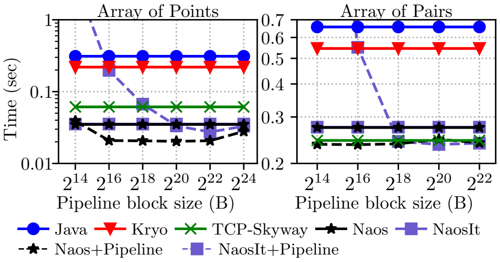
The performance of blocking Naos RDMA is bound by the network latency, which prevented the application to send requests at a higher rate. The NaosAsync RDMA, which avoids waiting for an acknowledgment, achieves the highest performance showing the importance of asynchronous communication. For the array of 512 floats, Naos achieves 1600 Kreq/sec, which is a 2x speedup over existing serialization approaches.

Figures 8(**b,c**) show that the throughput of Naos RDMA was limited by the network bandwidth since NaosIt, that communicates less data, outperforms NaosAsync RDMA. This observation indicates the benefit of our data compression.

The cycle detection decreases the throughput of Naos by less than 3% for moderately sparse graphs. For sparser graphs such as an array of Pairs the slowdown increases to 19%, which is explained by the growth of the Naos' interval tree for cycle detection. Therefore, Naos has lower performance than Kryo, but still outperforms the Java serializer. We think that, in real systems, Naos can be used together with traditional OSD libraries depending on the sparsity of the object graph.

Streaming data transfers. Data processing frameworks such as Spark and Flink rely on data streaming to enable processing of continuous streams of data. The continuous data stream is generated by sending small chunks of data to the





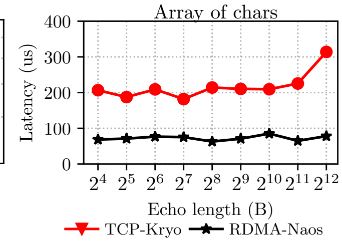


Figure 9: Streaming an array of 2^{20} elements. Figure 10: Pipelining an array of 2^{20} elements. Figure 11: Dubbo RPC latency.

processing nodes. To represent this use-case we implemented the streaming of long data arrays over RDMA networks. Figure 9 shows the streaming time of an array of Points and Pairs with increasing the chunk size.

For the array of Points, Naos RDMA outperforms all serializers for all chunk sizes, and decreases the streaming time of Kryo by 2.1x. For the array of Pairs, Kryo has the highest performance, by sending 256 objects at a time, due to its ability to compress the objects efficiently. For larger chunks, Naos and Skyway take less time than Kryo, since Kryo starts suffering from longer object construction for larger chunks, whereas Skyway and Naos do not need to construct objects.

Even though Skyway and Naos have the same data format, Skyway streamed the array of Pairs faster than Naos. The difference comes from the complexity of Naos' communication algorithm, leading to the higher CPU cost at the sender (see Table 3). Skyway's serialization code only copies traversed objects to send buffers, whereas Naos as a communication library needs to take more factors into account: building send lists, RDMA memory registration, and triggering multiple RDMA requests. Naos could employ various modern RDMA techniques for optimized memory accesses [18], which are interesting directions for future research.

Pipelining data transfers. Naos supports pipelining graph traversal with communication on the sender, and pointer fixing with communication on the receiver. Unlike Naos, conventional OSD approaches require an object to be fully serialized before sending it over the network. In this experiment, we show the effect of pipelining for large object graphs by measuring the time of transferring arrays with 2^{20} elements.

The latencies of Java, Kryo, Skyway, and Naos with no pipelining are depicted as straight lines in Figure 10 as they are independent of the pipeline size. Naos with pipelining provides a 20% reduction in latency in comparison with a non-pipelined variant, since the receiver can start pointer recovery earlier. Note that in the previous experiments with streaming large sparse object graphs, Kryo outperformed Naos as it could split the graph into chunks. For inseparable large graphs, however, Naos takes less time even for highly sparse graphs.

Workload	(50:50)	(95:5)	(100:0)
TCP-Kryo	14-24 Kreq/sec	16-24 Kreq/sec	23-25 Kreq/sec
RDMA-Naos	194-266 Kreq/sec	196-266 Kreq/sec	219-281 Kreq/sec

Table 4: Throughput under YCSB workloads with various (read:write) ratios. Percentiles 5 and 95 are reported.

4.2 Accelerating applications with Naos

Naos provides a simple programming interface (see Table 1) hiding all the burden of low-level RDMA communication. In particular, RDMA benchmarks from the previous experiments take only 10 lines for Naos and over 300 lines for the Disni RDMA library. Thus, we believe that it is simple to build systems using Naos. As proof, we have extended Apache Dubbo with Naos communicator, and implemented a Naos-enabled map-reduce framework.

Zero-copy RPC messages with Dubbo. To show that Naos is easy to use, we extended an RPC library Apache Dubbo with the Naos communicator. For that we added a new Naos-enabled communication module that has no serialization module.

In the first experiment we measure the latency of an RPC function that echoes back a Java String. Naos' performance was compared with the default TCP network library, Mina [3], with Kryo serializer. Naos was deployed with cycle detection. Note that Dubbo besides an RPC arguments also sends an RPC metadata resulting in sending several Java objects. Figure 11 shows that employing Naos RDMA decreases the latency by at least 55% for all tested sizes.

To understand the performance of Naos under a realistic throughput workload, we built a key-value store (KVS) using a Java concurrent hashtable and Dubbo library for communication. We populated the KVS with one million entries of 1 KiB each. We benchmark it under different YCSB [7] workloads. Table 4 shows that Naos RDMA achieves an average speedup of 11x over TCP-Kryo. The speedup comes from the fact that TCP-Kryo was bottlenecked by the CPU, whereas Naos consumes less CPU time to send a KVS request. The experiment shows that Naos can be utilized for KVS workloads as KVS requests and responses have low sparsity.

In comparison with microbenchmarks (§4.1), the performance difference between Naos and Kryo is much higher for the current workload than for the microbenchmarks, where Kryo's performance was measured after JIT compilation that

	LiveJournal [4] (2 nodes)				Orkut [35] (3 nodes)			
	TCP		RDMA		TCP		RDMA	
Test	Total	Stage	Total	Stage	Total	Stage	Total	Stage
Java	413.46	3.67-4.04	406.87	3.53-3.99	364.53	3.20-3.38	365.29	3.10-3.44
Kryo	410.94	3.62-4.06	411.33	3.63-3.99	357.58	2.98-3.47	354.06	2.91-3.42
Skyway	394.32	3.52-3.77	NA	NA	350.48	3.07-3.24	NA	NA
Naos	393.05	3.54-3.76	395.05	3.59-3.70	342.06	2.85-3.32	343.00	2.84-3.35
NaosIt	394.49	3.56-3.77	386.01	3.48-3.69	345.94	2.82-3.45	333.16	2.72-3.24
Skyway†	386.31	3.54-3.78	NA	NA	340.82	2.95-3.30	NA	NA
Naos†	373.04	3.27-3.72	369.10	3.16-3.63	331.31	2.86-3.22	335.50	2.85-3.22

† PageRank with sparsity-aware implementation.

Table 5: Total and per stage processing times in seconds for 100 iterations of PageRank algorithm. Percentiles 5 and 95 are reported for PageRank iterations.

significantly improved its performance for repetitive sending of the same object. Since Naos does not depend on Java runtime optimizations, it can achieve much higher performance than Kryo for dynamic workloads.

Improving Data Processing Applications. We could not integrate Naos into Spark as its shuffle module is designed to communicate files with serialized objects. Integration of Naos would require a substantial redesign of Spark's code base. Therefore, we implemented our own map-reduce framework that takes advantage of Naos. Our framework supports all discussed serializers including Skyway and also offers RDMA networking with Disni. It was designed to resemble Spark but perform shuffle completely in-memory.

We evaluate the OSD approaches by running PageRank on real-world graphs as input: LiveJournal [4] and Orkut [35]. The LiveJournal dataset was processed with two shuffle workers and Orkut with three shuffle workers. Naos was deployed with 256 KiB pipelining and without cycle detection. We report total runtime including data loading and 5 and 95 percentiles for processing a single Pagerank iteration. We also provide two implementations of PageRank: the first one follows conventional design where each score update is a class of 32 bytes; the second implementation was designed to communicate dense contiguous score updates, thereby reducing sparsity of communicated shuffle blocks.

Table 5 shows the lowest runtime was achieved by Naos and Skyway for the first implementation. A side effect of Naos and Skyway is that, after receiving, objects are always contiguous in memory, thereby improving data locality. As a result, an application can process such contiguous objects faster as fewer memory pages need to be fetched. Overall, Naos TCP performs approximately as Skyway, but NaosIt RDMA provides 2.1% and 4.8% improvement over Skyway for LiveJournal and Orkut, respectively. The experiment shows that zero-transformation approaches for OSD can reduce processing time for data-processing workloads.

The sparsity-aware implementation provides an additional 4% reduction in runtimes, showing that applications need to take Naos' limitations into consideration to achieve the highest performance. Thus, Naos could be used in combination with works on data sparsity reduction for JVMs [5, 33, 34].

5 Discussion and Future work

The role of RDMA. RDMA helps Naos to remove potential copies induced by the TCP stack. Application-wise, Naos is zero-copy for both TCP and RDMA networks, unlike Skyway. On the other hand, for trivial graphs, Skyway and Naos TCP use almost identical algorithms for the receiver, as they both receive objects with zero-copy and only fix the class reference. However, since Naos TCP does not pre-allocate memory for receiving, its performance could be bound by memory allocation. It is possible to modify Naos TCP to pre-allocated buffers as Skyway and RDMA Naos do, removing the bottleneck. In this work, however, we focus on the RDMA implementation of Naos.

SmartNICs. In Naos, a sender cannot modify its on-heap memory before sending. Therefore, a receiver has to employ a complex pointer recovery algorithm, whereas Skyway can pre-process buffers before sending them to help the receiver to recover objects faster. We believe that such a feature is better implemented at the SmartNIC level that would fix the pointers on the fly before writing the data to DRAM. For example, since a Naos' RDMA sender already knows the destination addresses of the objects, either the SmartNIC at the sender or at the receiver could fix the object pointers. The class pointers could be fixed by storing class translation tables in the SmartNIC.

6 Conclusions

We have presented Naos, a JVM communication library that enables transferring objects directly from one heap to another over the network with minimal CPU involvement and zero-copy . We demonstrated that existing OSD techniques are bound to CPU and that, as networks get faster, they will become the bottleneck of distributed systems. Naos completely avoids the need to serialize and deserialize objects for data transfers, with the corresponding performance advantages. Naos provides a simple API that simplifies the use of RDMA from JVM-based applications. Our evaluation shows that Naos outperforms all existing OSD approaches for moderately sparse object graphs.

7 Acknowledgement.

We would like to thank our shepherd, Khanh Nguyen, and the anonymous reviewers for their suggestions and help to improve the paper. This research was supported by ETH Zurich, and by Microsoft Research through its Swiss Joint Research Centre. The project also received funding from the European Research Council under the European Union Horizon 2020 programme (grant agreement DAPP, No. 678880). Rodrigo Bruno's research was supported in part by grants from Oracle Labs and SBB.

References

[1] Oracle Corporation and/or its affiliates. JDK 11, 2019. https://openjdk.java.net/projects/jdk/11/.

[2] Dubbo Apache. Apache Dubbo Project, 2018. https://dubbo.apache.org.

[3] MINA Apache. Apache MINA Project, 2009. https://mina.apache.org.

[4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'06, pages 44–54. Association for Computing Machinery, 2006.

[5] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. Compiler-Assisted Object Inlining with Value Fields. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'21. Association for Computing Machinery, 2021.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154. Association for Computing Machinery, 2010.

[8] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'14, pages 401–414. USENIX Association, 2014.

[9] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ'16. Association for Computing Machinery, 2016.

[10] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[11] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference*, USENIX ATC'10. USENIX Association, 2010.

[12] Google Inc. Protocol buffers: Google's data interchange format, 2008. https://github.com/protocolbuffers/protobuf.

[13] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. A specialized architecture for object serialization with applications to big data analytics. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA'20, page 322–334. IEEE Press, 2020.

[14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference*, USENIX ATC'16, pages 437–450. USENIX Association, 2016.

[15] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA'15, pages 158–169. Association for Computing Machinery, 2015.

[16] Sean Leary. JSON-java, 2015. https://github.com/stleary/JSON-java.

[17] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page Fault Support for Network Controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 449–466. Association for Computing Machinery, 2017.

[18] Mellanox Technologies Ltd. Optimized memory access, 2020. https://docs.mellanox.com/display/MLNXOFEDv492240/Optimized+Memory+Access.

[19] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda. High-performance design of Apache Spark with RDMA and its benefits on various workloads. In *2016 IEEE International Conference on Big Data*, BigData'16, pages 253–262, 2016.

[20] Frank Mietke, R. Baumgartl, R. Rex, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In *Proceedings of Euro-Par 2006*

Parallel Processing, pages 124–133. Springer-Verlag Berlin, 2006.

[21] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'18, pages 56–69. Association for Computing Machinery, 2018.

[22] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'15, pages 293–307. USENIX Association, 2015.

[23] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'15, pages 107–118. Association for Computing Machinery, 2015.

[24] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 1203–1216. Association for Computing Machinery, 2020.

[25] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium*, USENIX Security'21. USENIX Association, 2021.

[26] Esoteric Software. Kryo - object graph serialization framework for Java, 2008. https://github.com/EsotericSoftware/kryo.

[27] Patrick Stuedi, Bernard Metzler, and Animesh Trivedi. JVerbs: Ultra-Low Latency for Data Center Applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC'13. Association for Computing Machinery, 2013.

[28] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. DaRPC: Data Center RPC. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'14, pages 1–13. Association for Computing Machinery, 2014.

[29] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of Temporary Storage in the NodeKernel Architecture. In *2019 USENIX Annual Technical Conference*, USENIX ATC'19, pages 767–782. USENIX Association, 2019.

[30] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, SIGMOD'21. Association for Computing Machinery, 2021.

[31] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA – Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access. In *2020 USENIX Annual Technical Conference*, USENIX ATC'20, pages 691–704. USENIX Association, 2020.

[32] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA'16, pages 53–65. IEEE Press, 2016.

[33] Christian Wimmer and Hanspeter Mössenböck. Automatic Feedback-Directed Object Inlining in the Java Hotspot™ Virtual Machine. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE'07, pages 12–21. Association for Computing Machinery, 2007.

[34] Christian Wimmer and Hanspeter Mössenböck. Automatic Array Inlining in Java Virtual Machines. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'08, pages 14–23, New York, NY, USA, 2008. Association for Computing Machinery.

[35] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities Based on Ground-Truth. *Knowl. Inf. Syst.*, 42(1):181–213, 2015.

[36] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10. USENIX Association, 2010.