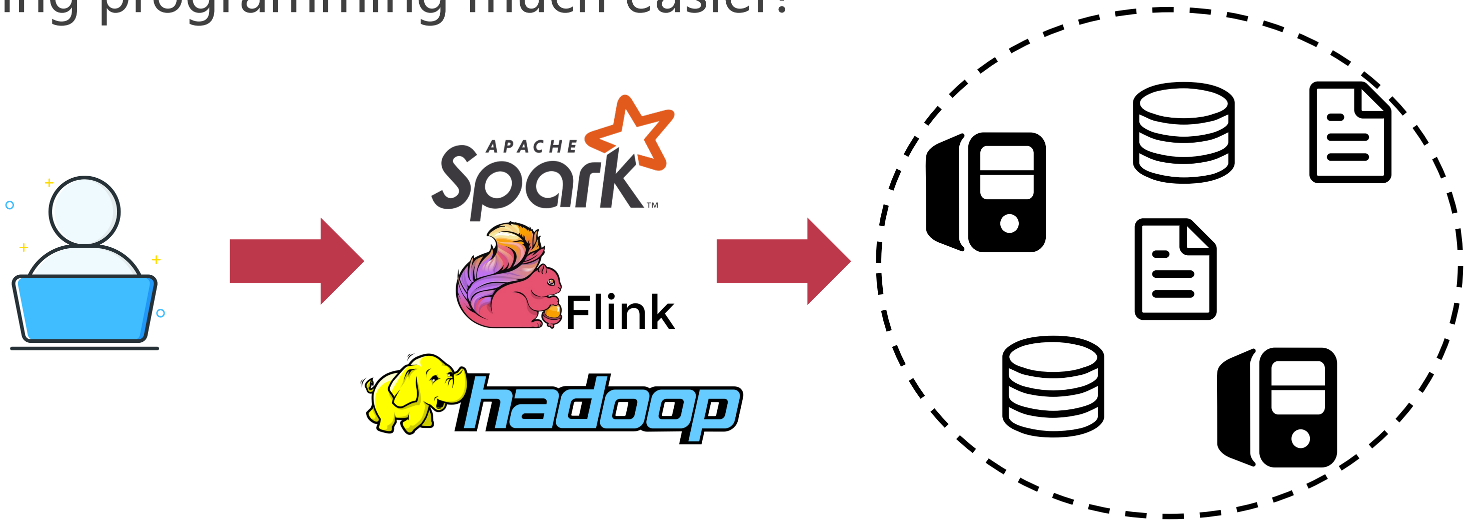
Zero-Change Object Transmission for Distributed Big Data Analytics

Mingyu Wu, Shuaiwei Wang, Haibo Chen, Binyu Zang

Shanghai Jiao Tong University

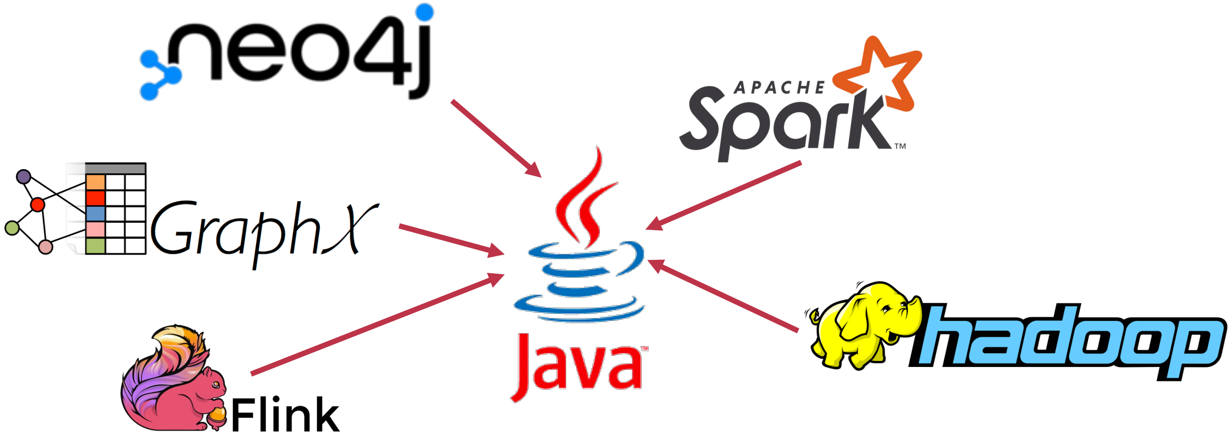
Distributed Big-data Analytics

- **Widely used in many areas**
- **Hiding messy details on distributed data processing**
  - Task scheduling, resource management, fault tolerance…
  - Making programming much easier!



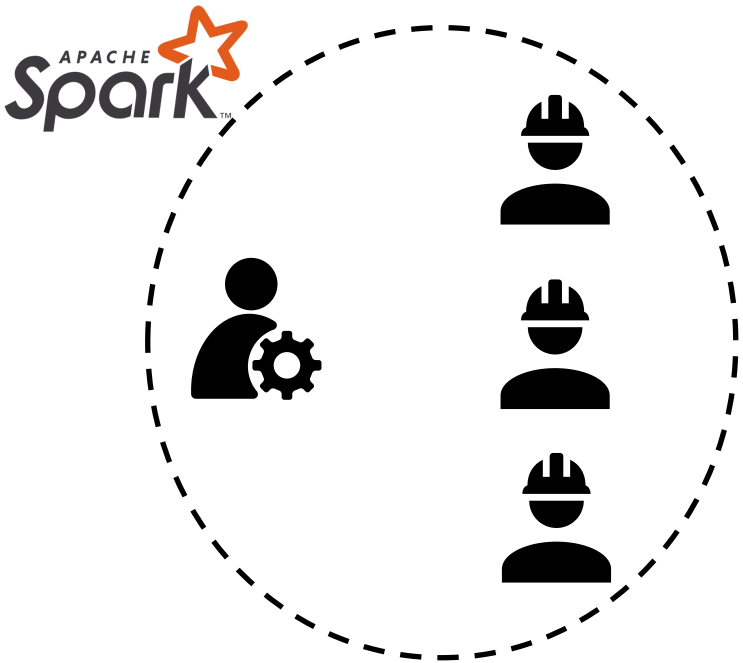
Distributed Big-data Analytics

- **Widely used in many areas**
- **Hiding messy details on distributed data processing**
- **Most are written in languages like Java and Scala**
  - Relying on the runtime environment provided by JVMs



Workflow of Big-data Processing

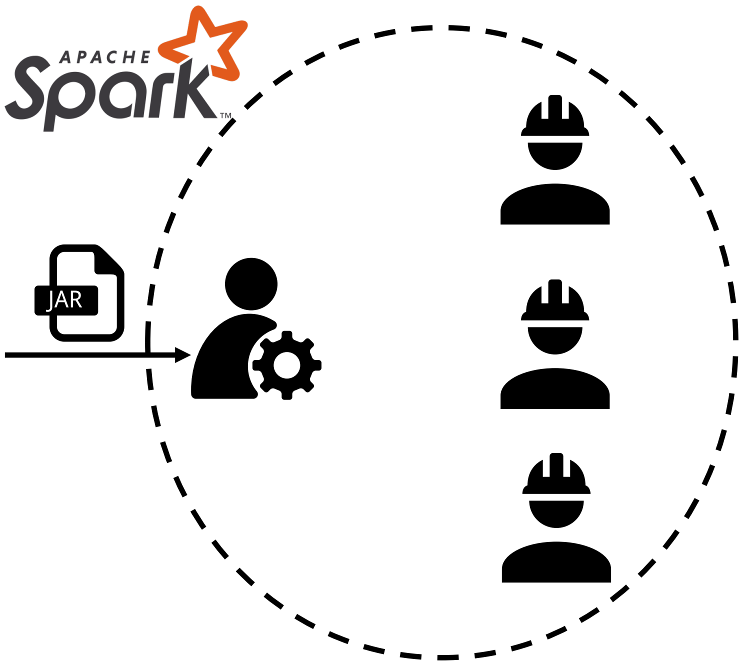
- **Launching managers and workers on various machines**
  - Taking Spark as an example: 1 manager, 3 workers



Workflow of Big-data Processing

- **Launching managers and workers on various machines**
  - Taking Spark as an example: 1 manager, 3 workers

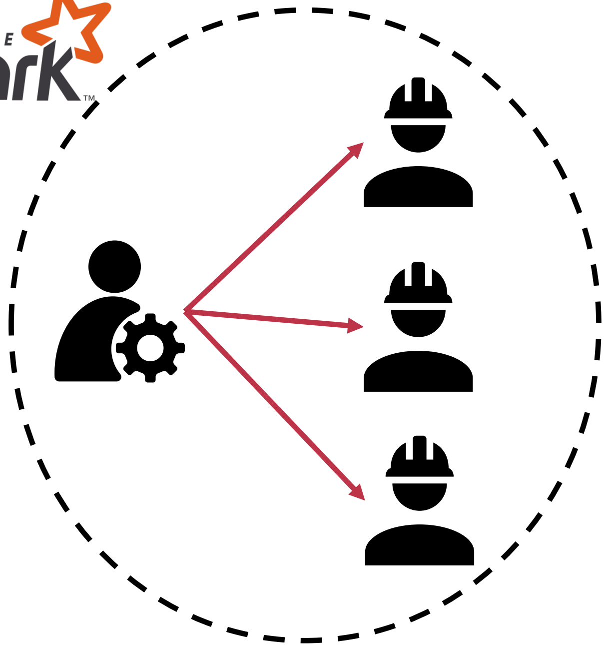
1. upload applications



Workflow of Big-data Processing

- **Launching managers and workers on various machines**
  - Taking Spark as an example: 1 manager, 3 workers

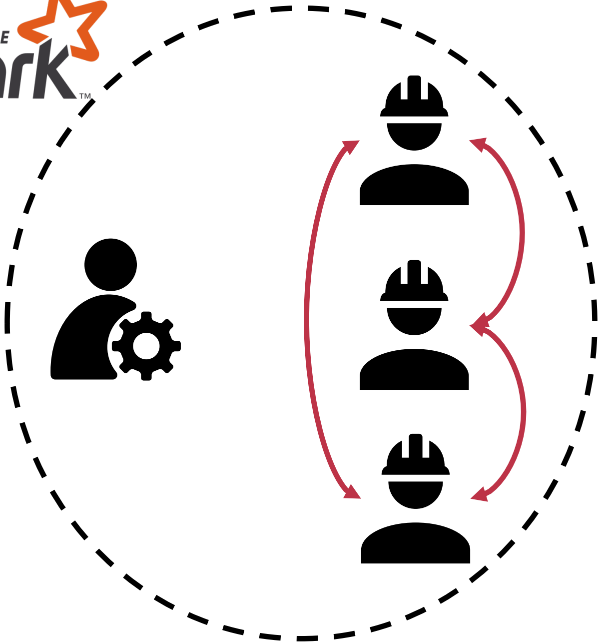
2. Task assignment



Workflow of Big-data Processing

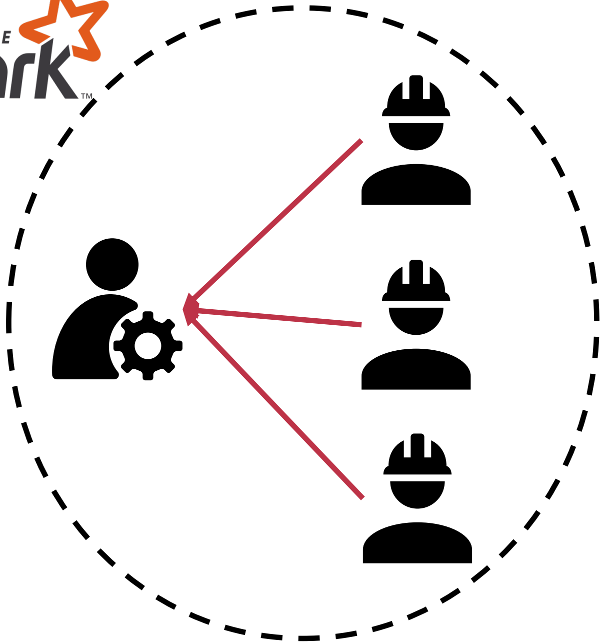
- **Launching managers and workers on various machines**
  - Taking Spark as an example: 1 manager, 3 workers

3. Inter-worker shuffling



Workflow of Big-data Processing

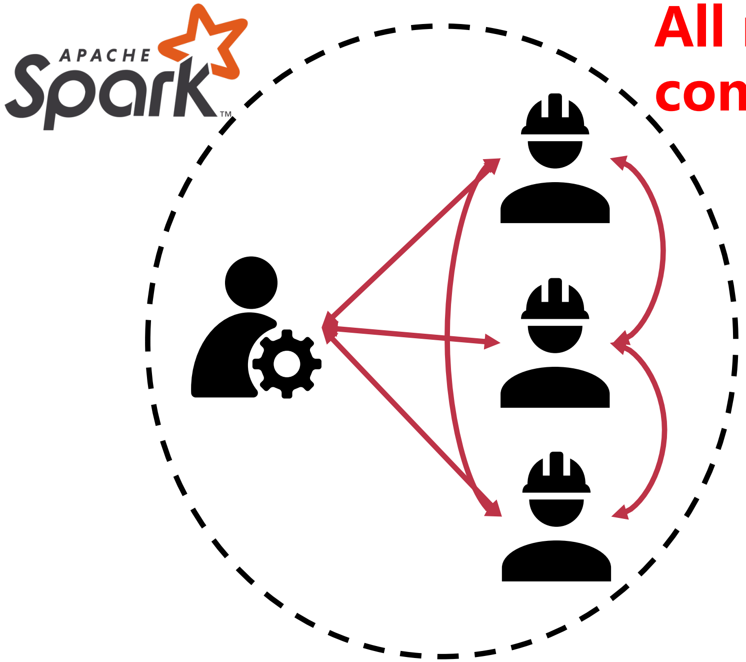
- **Launching managers and workers on various machines**
  – Taking Spark as an example: 1 manager, 3 workers



4. Returning results

Workflow of Big-data Processing

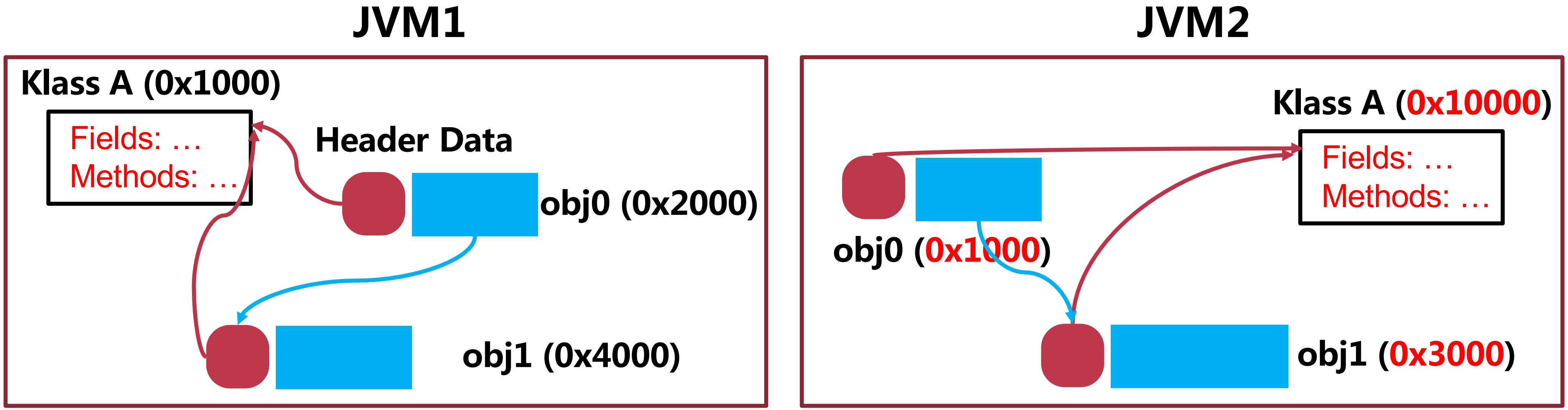
- **Launching managers and workers on various machines**
  - Taking Spark as an example: 1 manager, 3 workers



All nodes frequently communicate with each other!

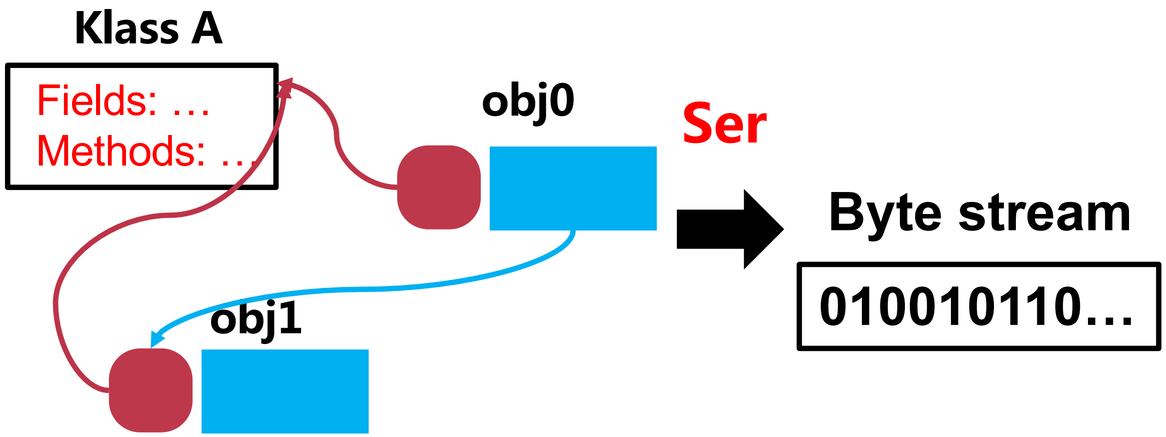
Costly Inter-JVM Communications

- **Each JVM has its own way to represent Java objects**
  - Header: storing an address to its type information (*Klass*)
  - Data: storing absolute addresses of other objects
  - Both are different in different JVMs



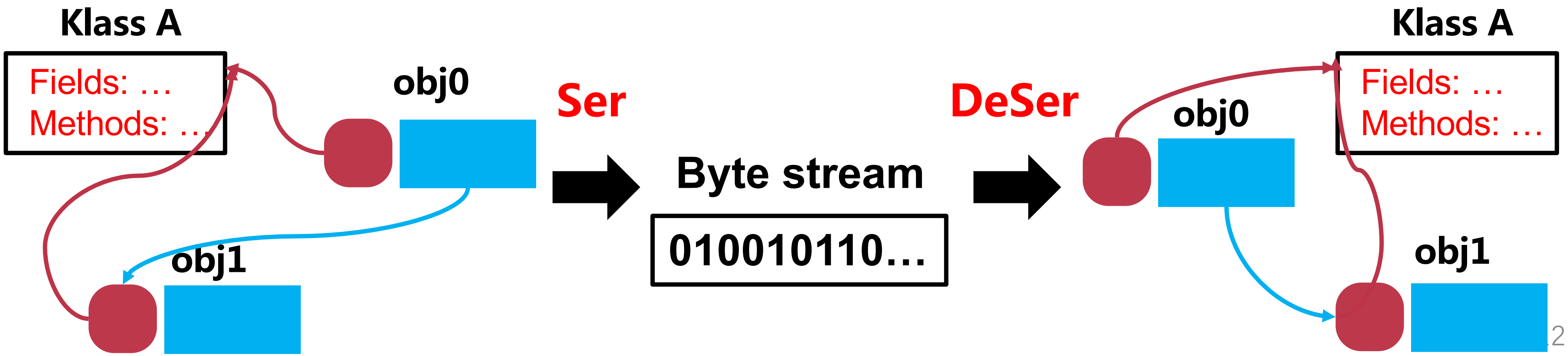
Costly Inter-JVM Communications

- **Java default solution: serialization/deserialization (S/D)**
  - Serialization: objects -> byte stream (general format)



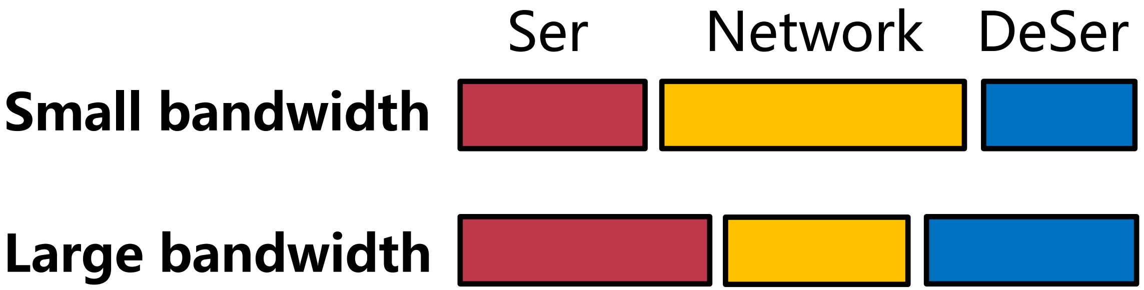
Costly Inter-JVM Communications

- **Java default solution: serialization/deserialization (S/D)**
  - Serialization: objects -> byte stream (general format)
  - Deserialization: byte stream -> objects



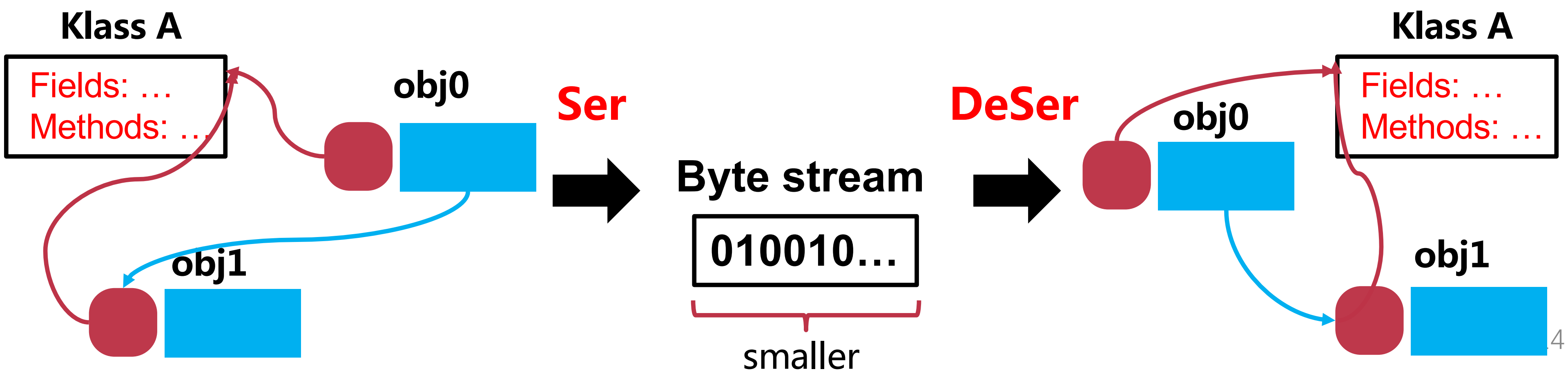
Costly Inter-JVM Communications

- **S/D is quite costly**
  - Ser: traversing all reachable objects and pack them
  - Deser: decoding bytes and allocating new objects
  - Both compute-intensive, cannot be improved by better network
  - S/D can account for more than 50% of the execution time!



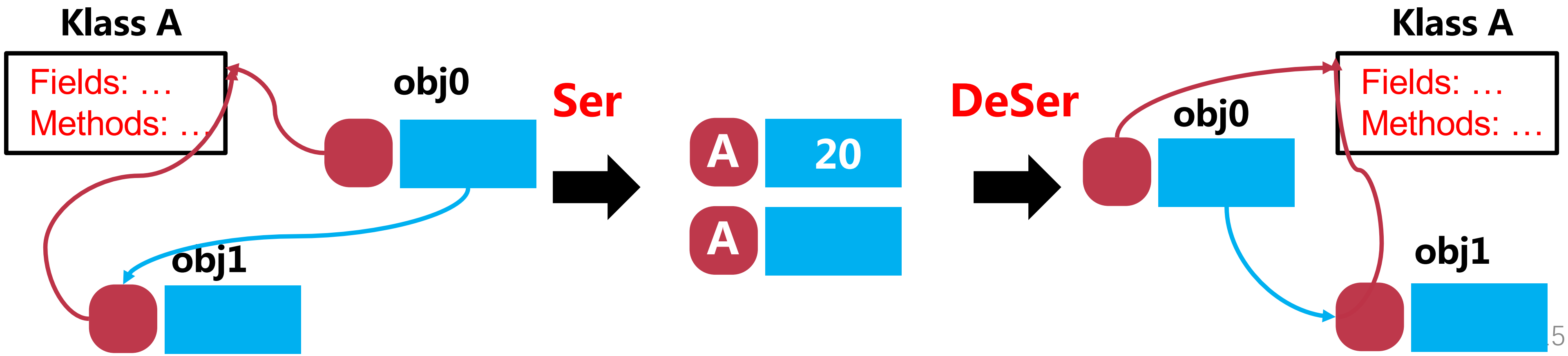
Existing S/D Optimizations

- **Kryo: improving the original (Java built-in) S/D tool**
  - The layout of byte streams becomes more compact
  - The transformation phases still exist



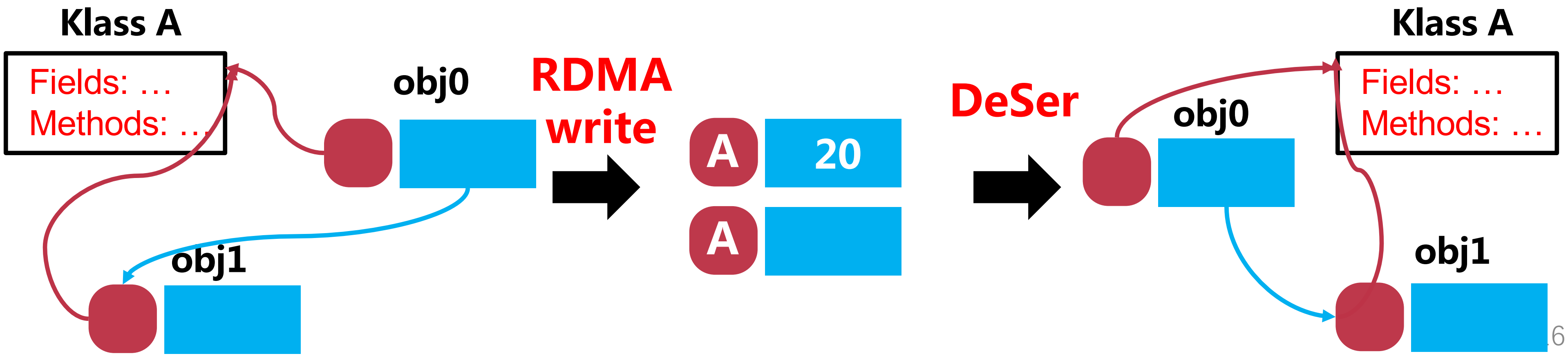
Existing S/D Optimizations

- **Kryo: improving the original (Java built-in) S/D tool**
- **Skyway: directly sending object graphs**
  - Encoding/decoding type information and references during S/D
  - Still require transformation on references and type information



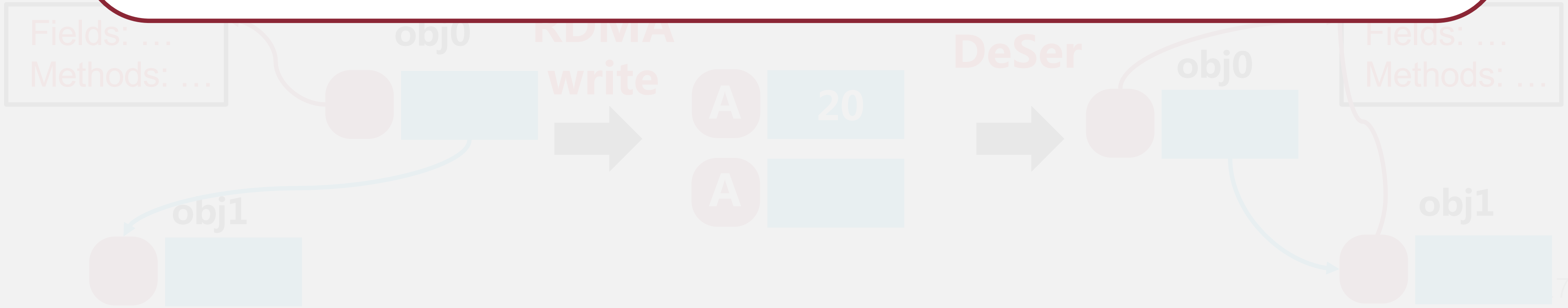
Existing S/D Optimizations

- **Kryo: improving the original (Java built-in) S/D tool**
- **Skyway: directly sending object graphs**
- **Naos: RDMA-friendly object-based transmission**
  - References and type information still requires fixing



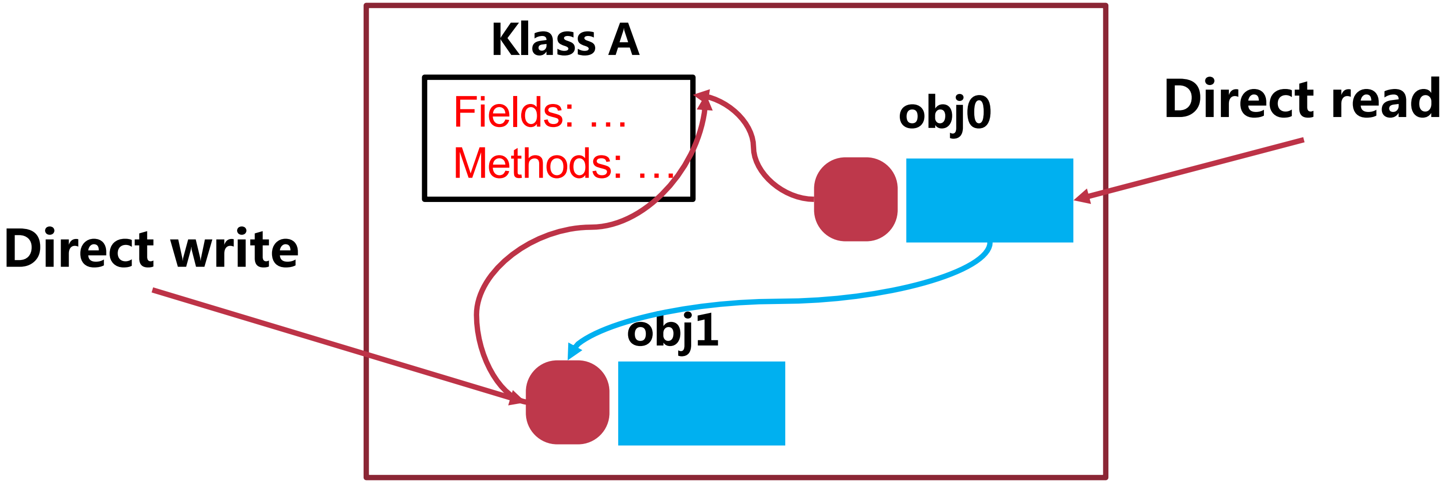
Existing S/D Optimizations

Can we *totally* remove the S/D-related transformation?



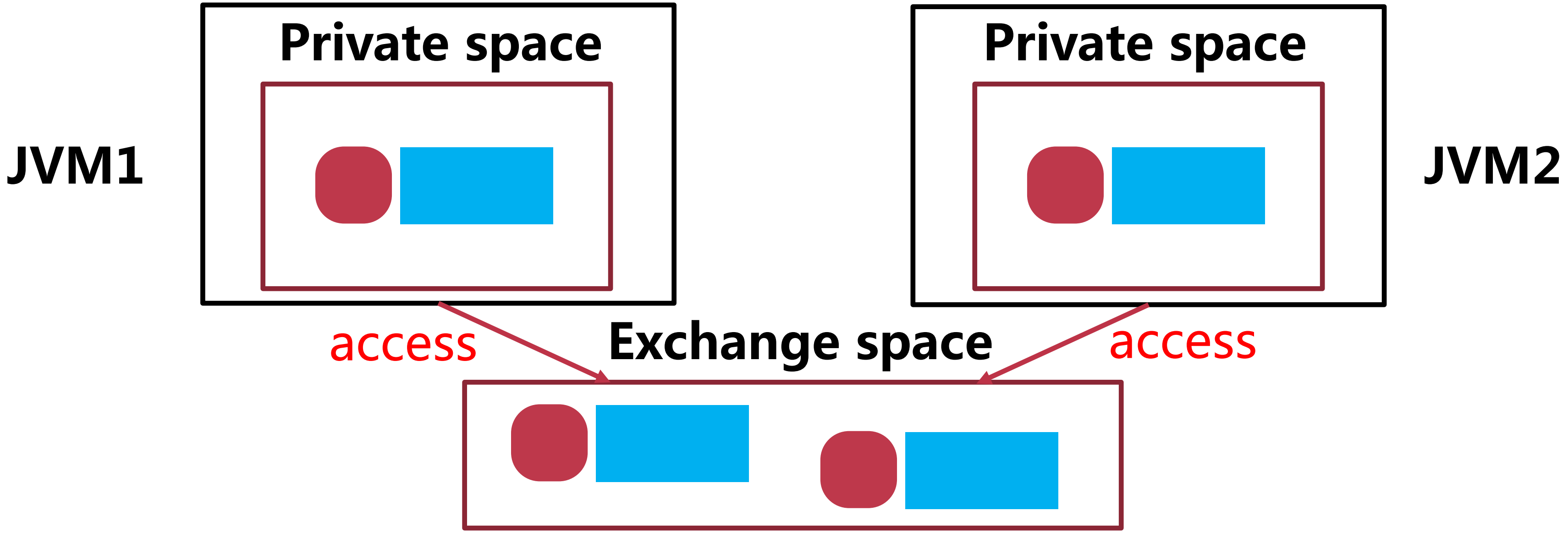
Our Solution: ZCOT

- **Zero-Change Object Transmission**
  - Upon receiving, objects can be directly used without any change
- **With ZCOT, objects can be directly read and written**



How to Achieve This?

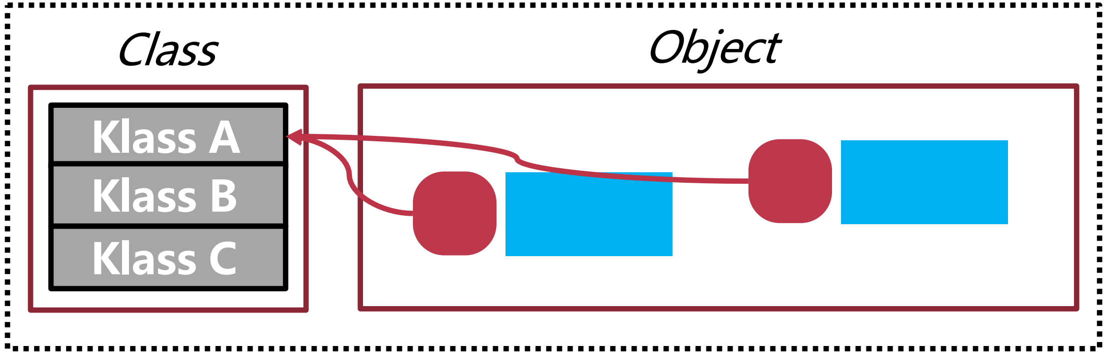
- **Each JVM has a shared space (exchange space)**
  - Objects can be directly accessed without pointer fixing
  - A per-JVM private space is used for normal allocation



How to Achieve This?

- **Each JVM has a shared space (exchange space)**
- **Exchange space contains a class sub-space**
  - Storing type information used by objects in the exchange space
  - No class pointer is required to fix

Exchange space

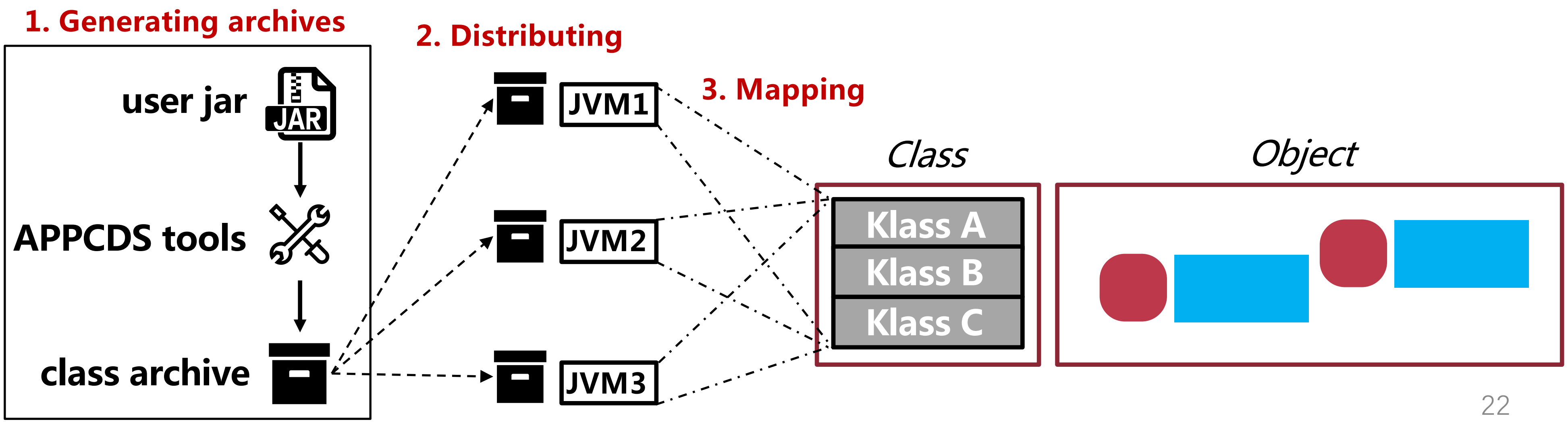


Challenges for ZCOT

- How to construct a shared space for all JVMs?
- How to remain compatible with existing applications?
- How to manage memory resources among JVMs?

Space Construction: DCDS

- **Extending the built-in APPCDS to support distributed sharing**
  - Allowing applications to share classes among JVMs
  - Reusing JDK built-in tools to construct a shared space



Compatibility with Applications

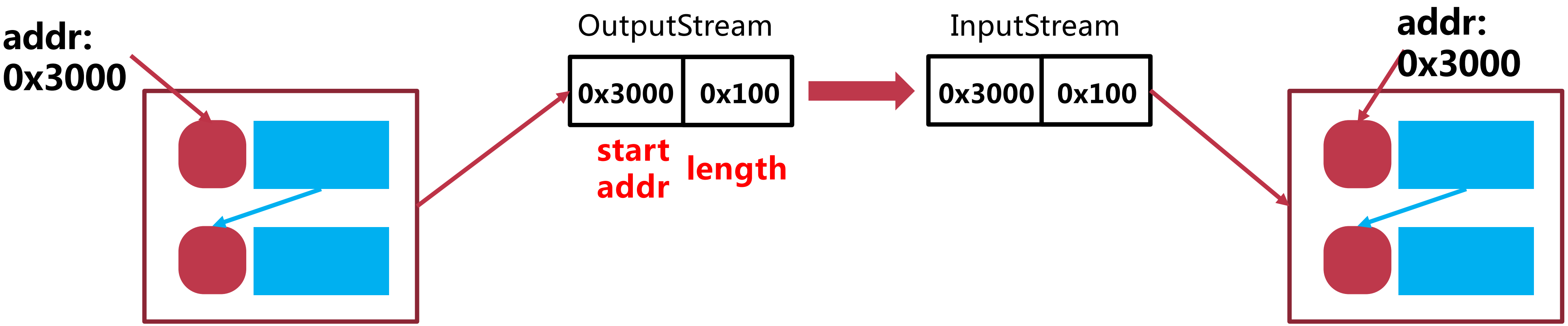
- **ZCOT sends/receives data in an object format**

- **However: existing applications still use S/D interfaces**
  - Ser: writeObject(Object obj) (into a byte **OutputStream**)
  - DeSer: readObject() (from a byte **InputStream**)

How to remain compatible with ZCOT's object-based mechanism?

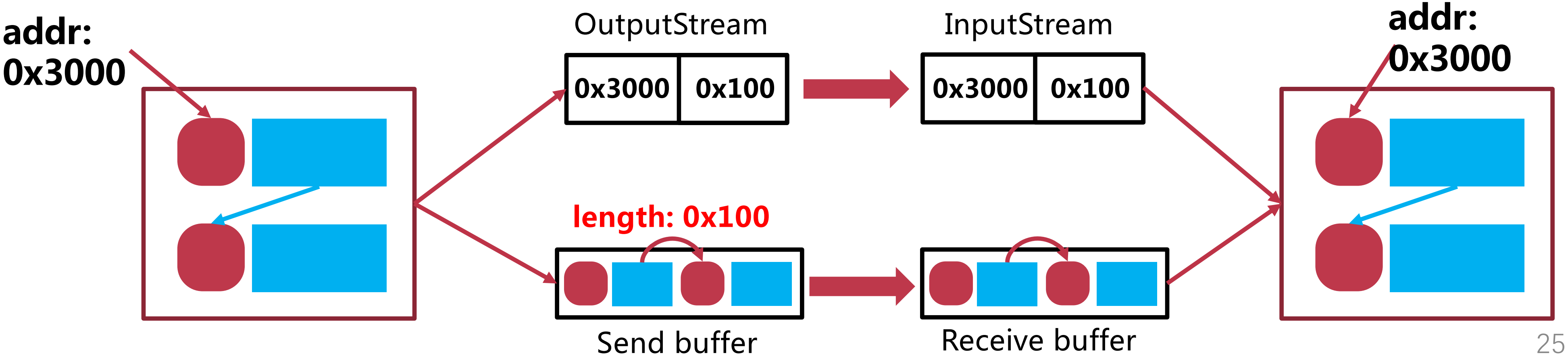
Compatibility with Applications

- **ZCOT's Solution: two-level data transmission**
  - Dividing into frontend and backend
  - Frontend: still remaining compatible with original S/D interfaces



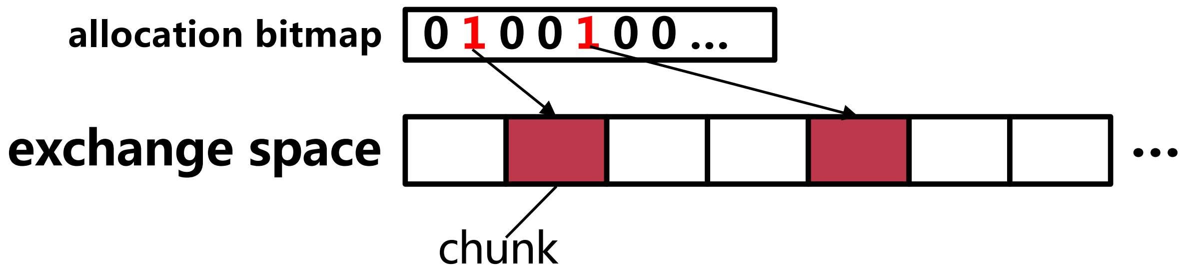
Compatibility with Applications

- **ZCOT's Solution: two-level data transmission**
  - Dividing into frontend and backend
  - Frontend: still remaining compatible with original S/D interfaces
  - Backend: sending and receiving real objects



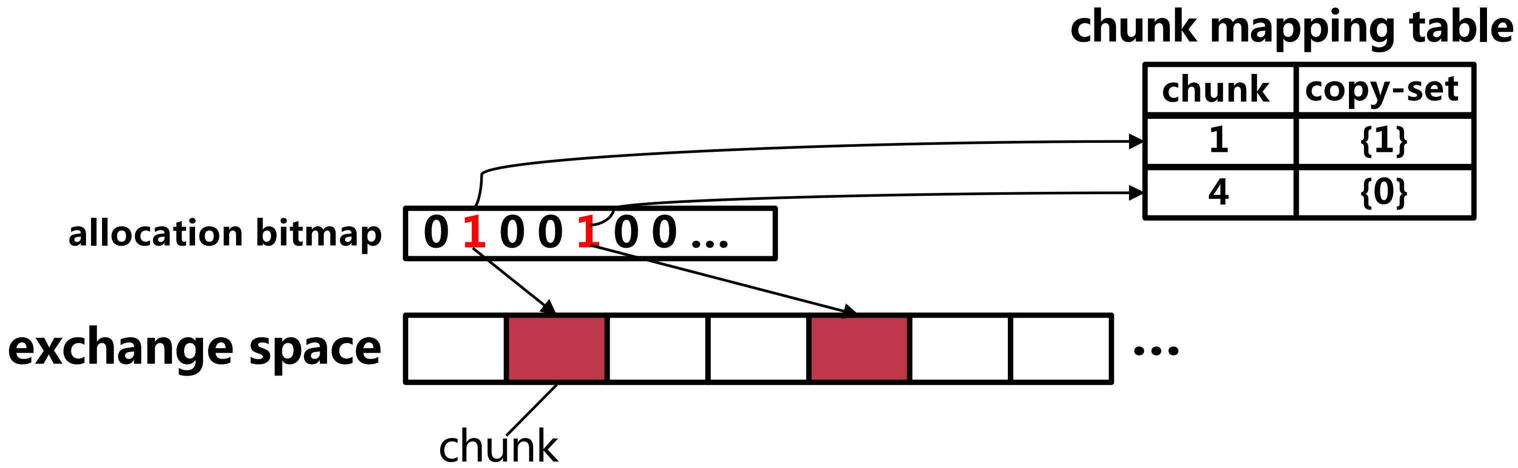
Distributed Memory Management

- **Using a metadata server to manage the exchange space**
  - Basic unit: chunks (default size: 256MB)
  - Allocation bitmap: marking if a chunk has been allocated



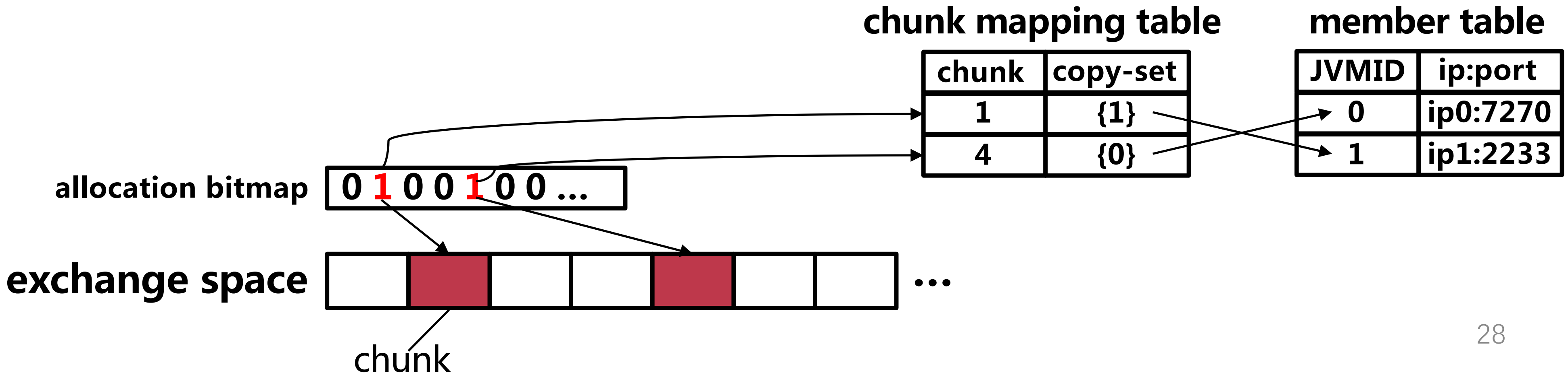
Distributed Memory Management

- **Using a metadata server to manage the exchange space**
  - Basic unit: chunks (default size: 256MB)
  - Allocation bitmap: marking if a chunk has been allocated
  - Chunk mapping table: marking which JVMs has the chunk



Distributed Memory Management

- **Using a metadata server to manage the exchange space**
  - Basic unit: chunks (default size: 256MB)
  - Allocation bitmap: marking if a chunk has been allocated
  - Chunk mapping table: marking which JVMs has the chunk
  - Member table: info for all JVMs

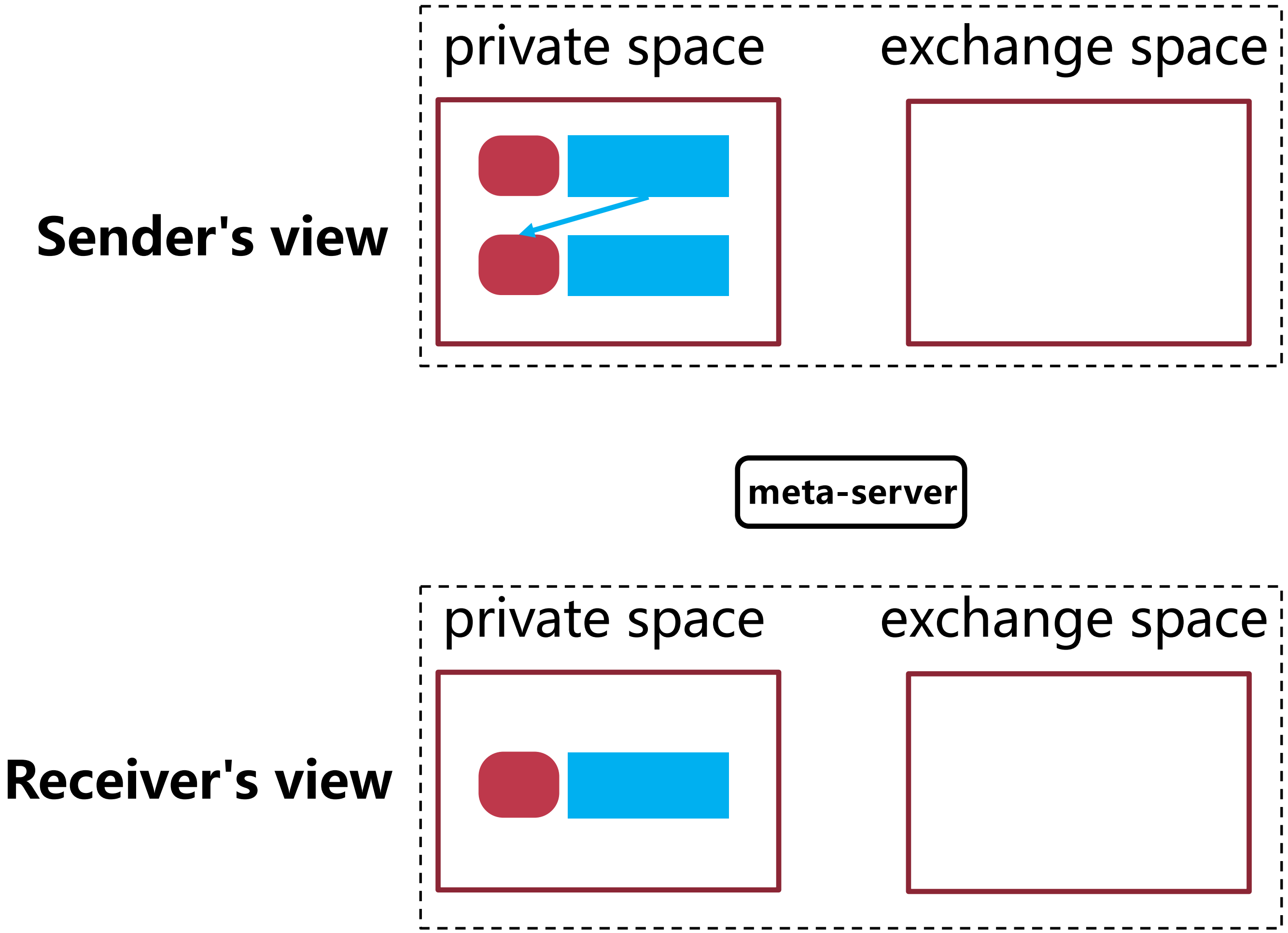


RPC Interfaces

- **The metadata server provides 4 RPC interfaces**
  - **register**: register a JVM into the member table
  - **acquire**: acquire a new chunk from the metadata server
  - **get_remote**: get a chunk from other JVMs
    - Coordinated by the metadata server
  - **release**: release a chunk to the metadata server

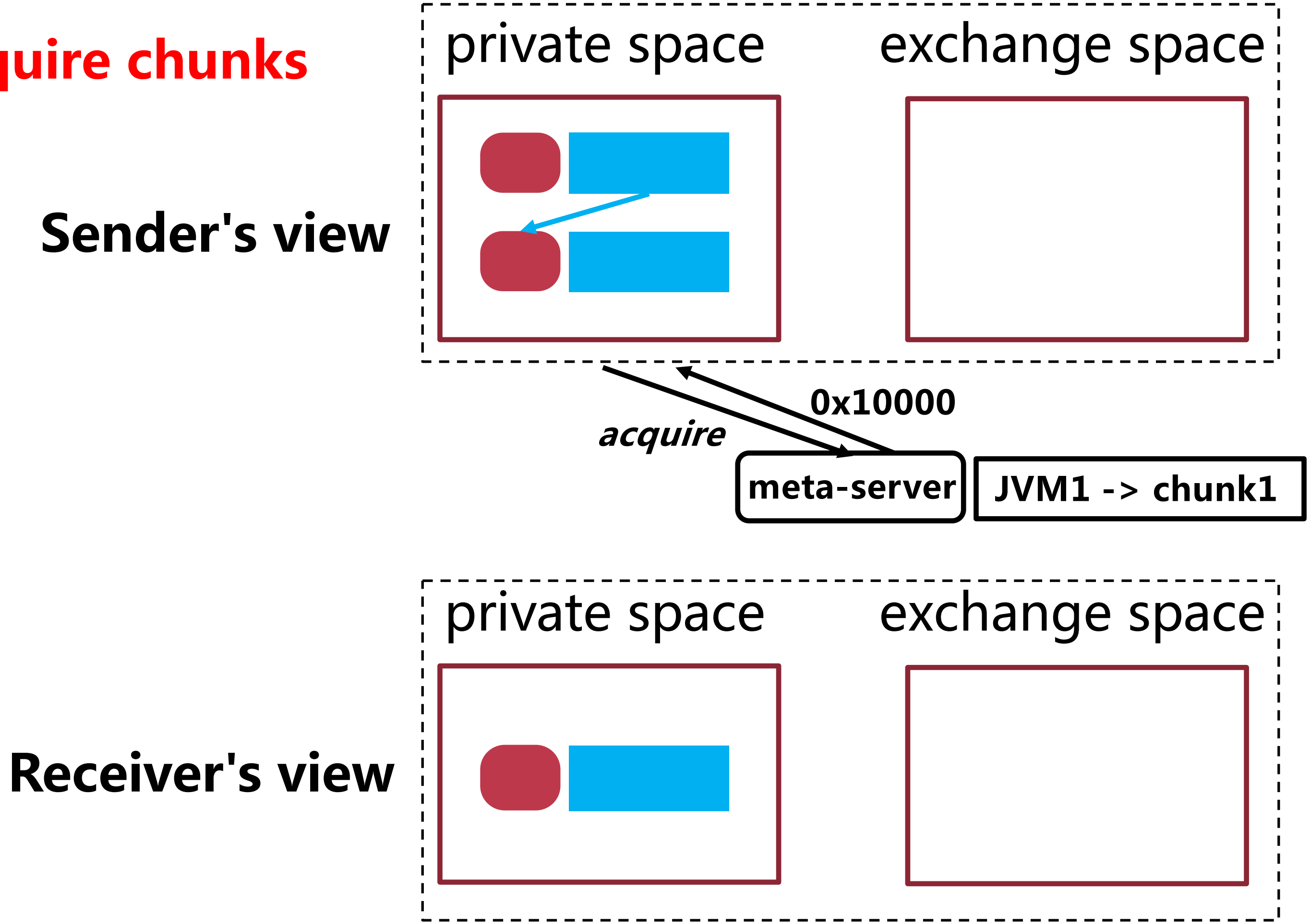
- **Integrated with memory management of JVMs**
  - E.g., GC should invoke the release RPC

The Workflow of ZCOT

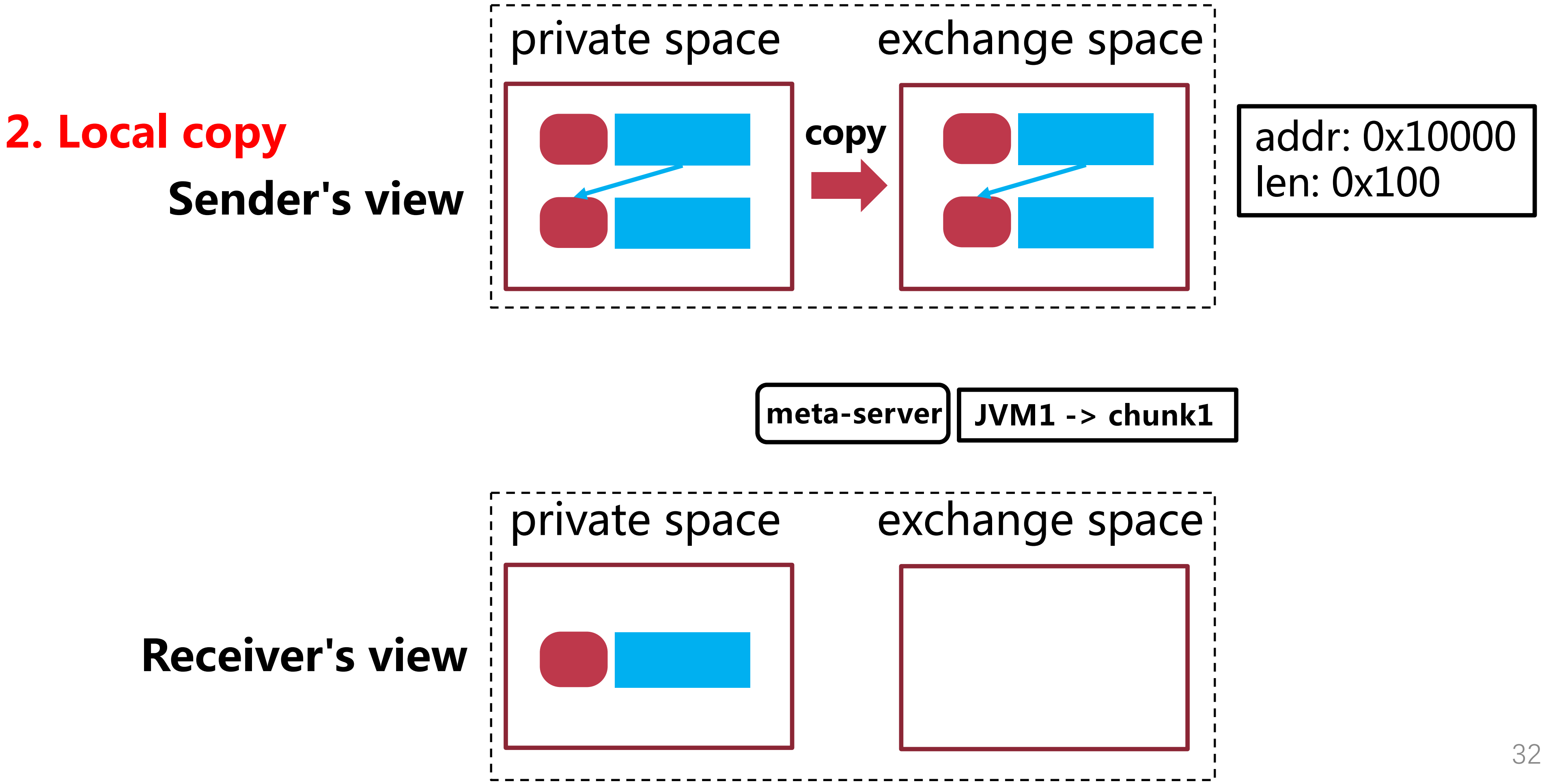


The Workflow of ZCOT

1. Acquire chunks

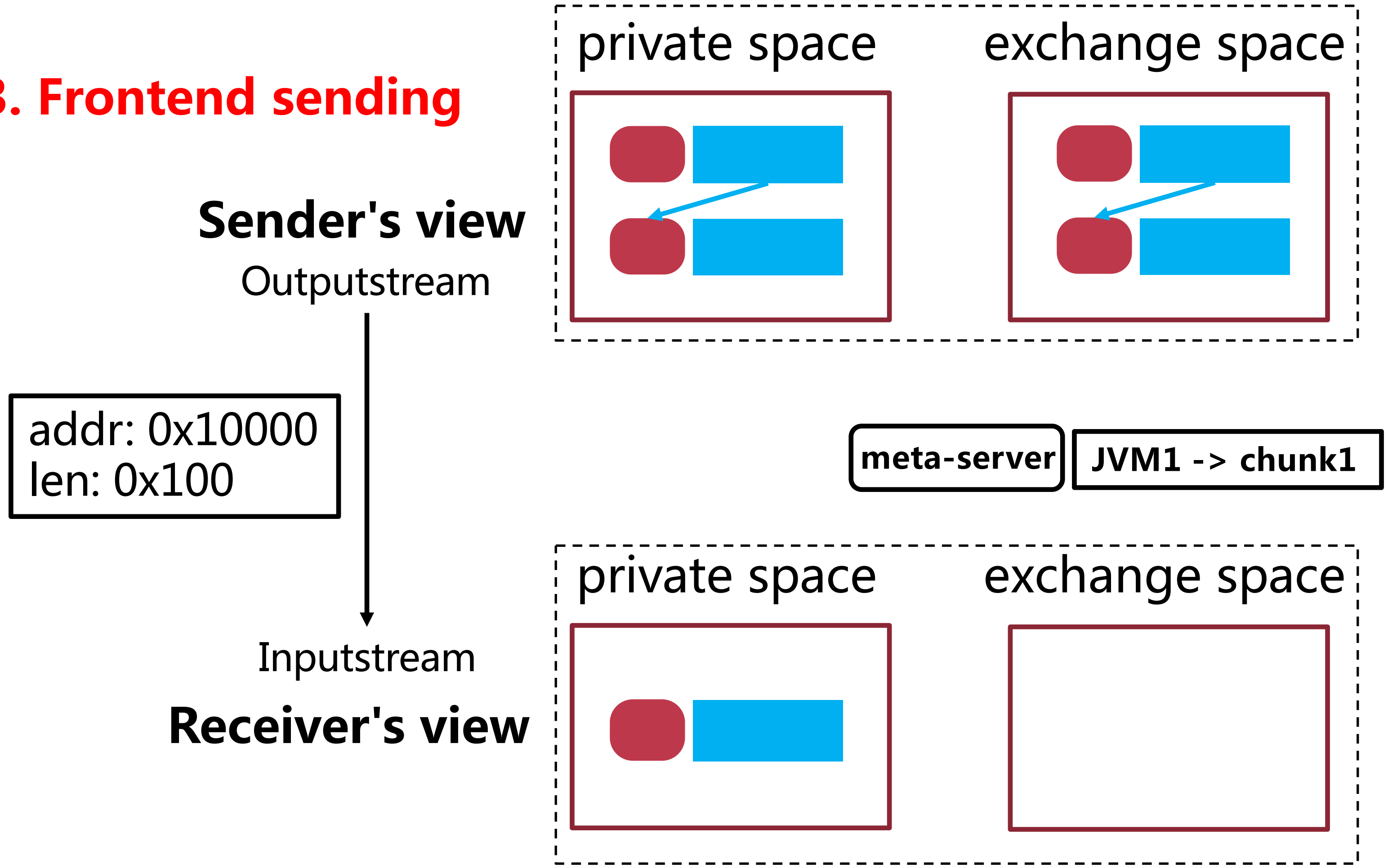


The Workflow of ZCOT



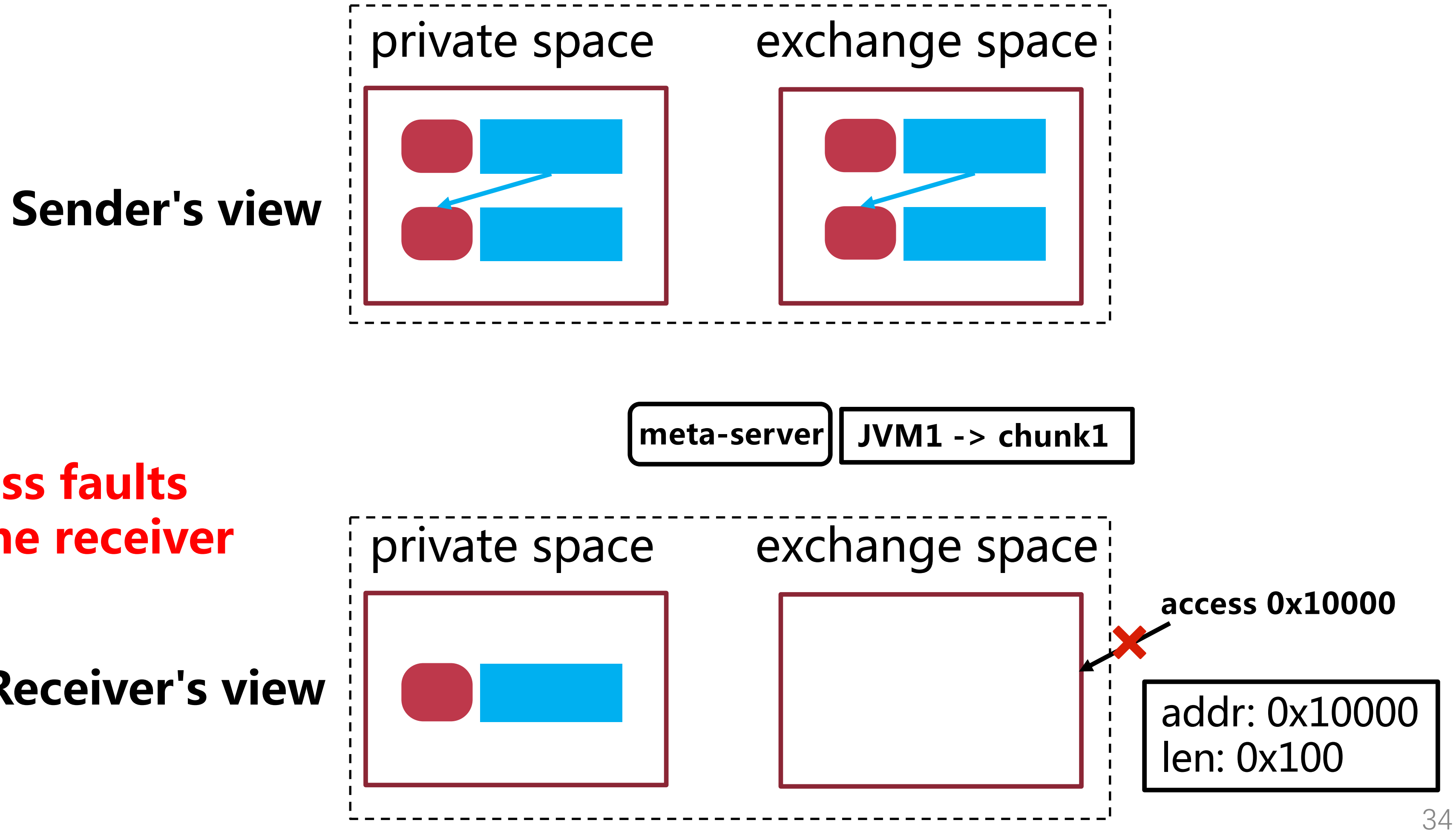
The Workflow of ZCOT

3. Frontend sending



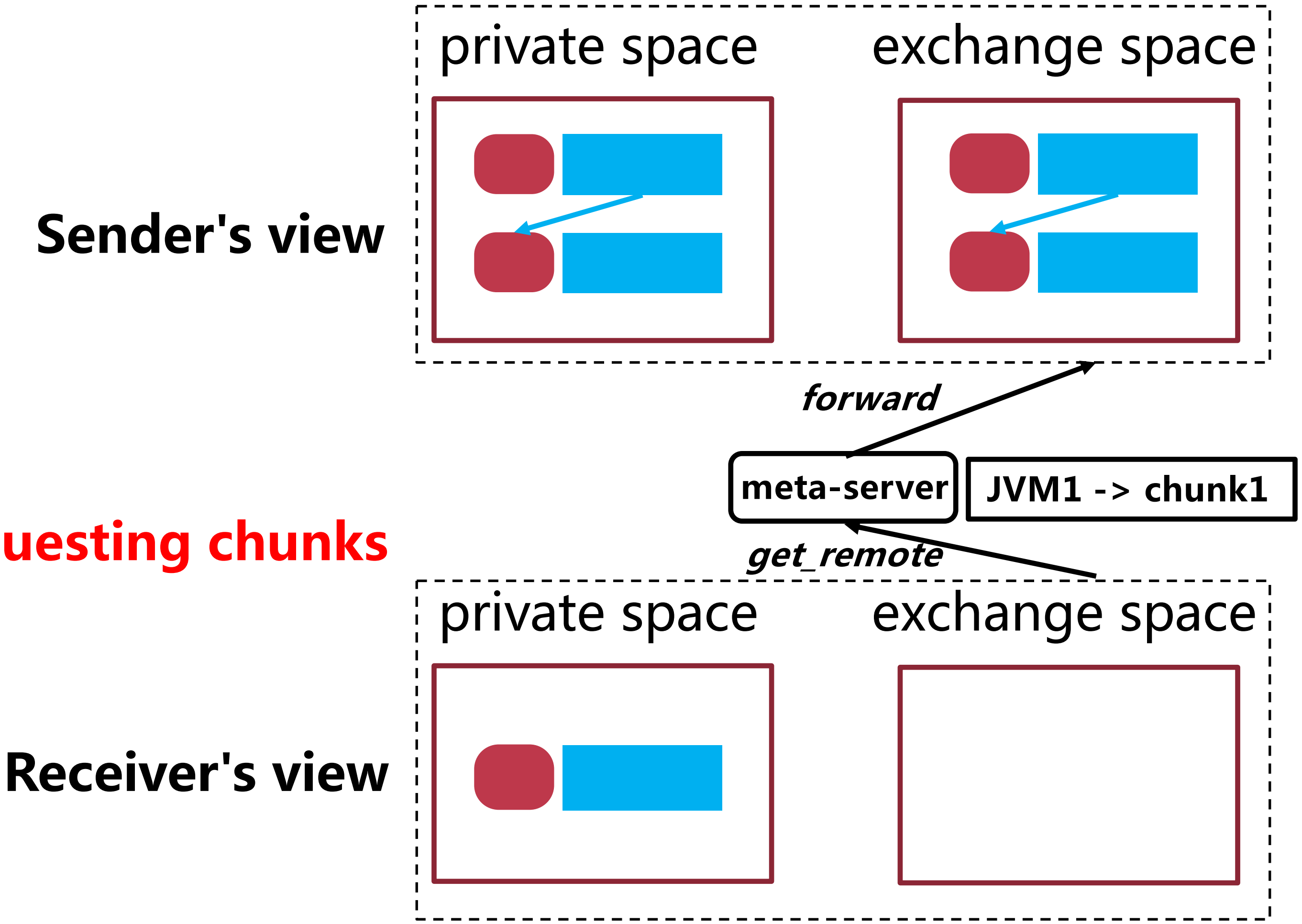
The Workflow of ZCOT

4. Access faults on the receiver



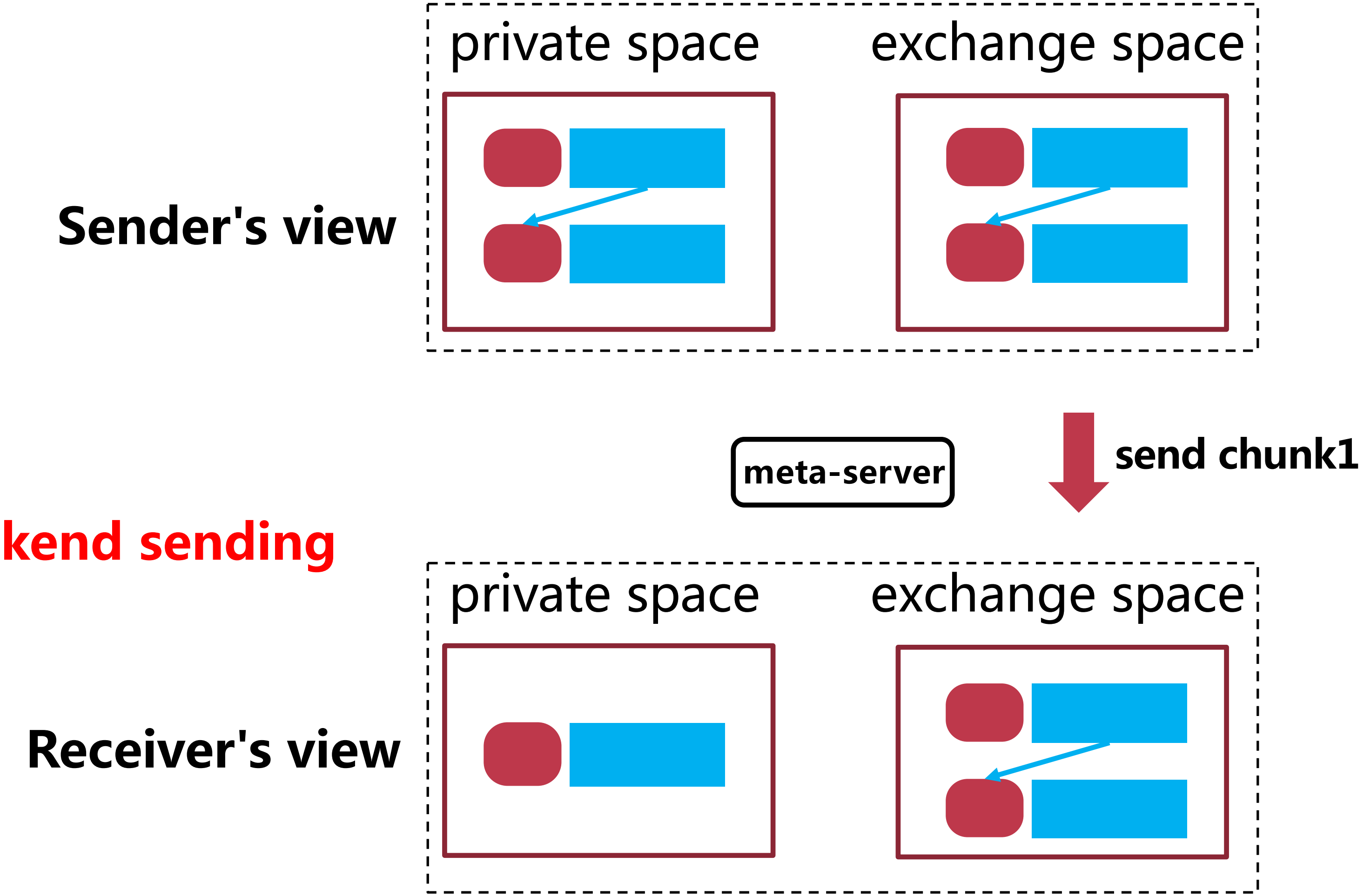
The Workflow of ZCOT

5. Requesting chunks



The Workflow of ZCOT

6. Backend sending



More Details in Our Paper

- **Data persistence**
- **Group-based prefetching**
- **Integrated with GC**
- **Data deduplication among multiple rounds**

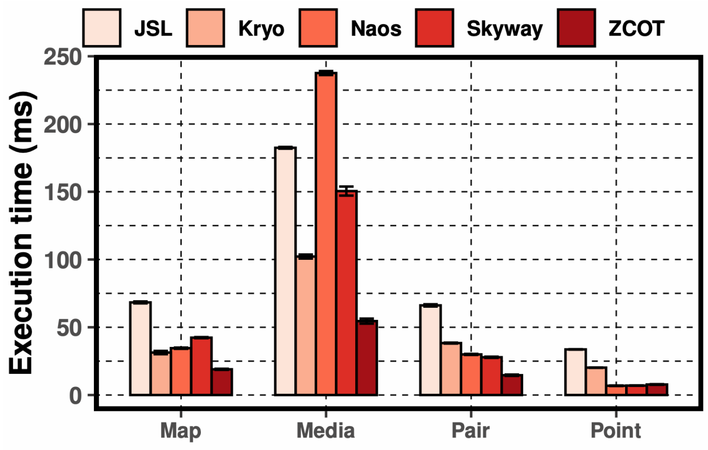
Experimental Setup

- **Hardware: A cluster with four nodes**
  - 100 Gbit/s Mellanox ConnectX-5 NICs
  - Dual Xeon E5-2650 CPUs and 128GB DRAM for each

- **Three evaluated applications**
  - Microbenchmark: data structures used in Naos and Skyway
  - Spark-v3.0.0
  - Flink-v1.14

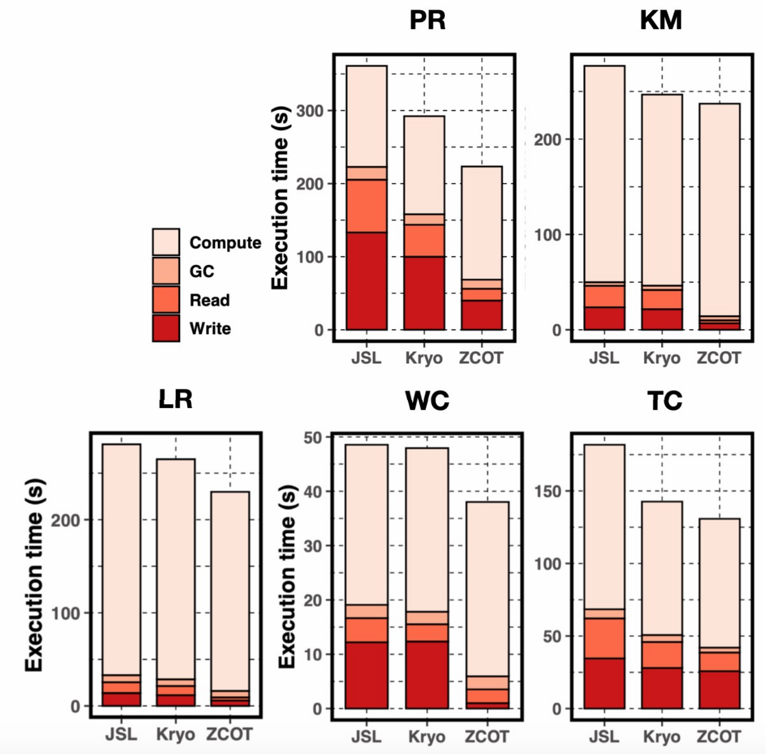
Mircobenchmark

- **Using the *microperf* tester from Naos for evaluation**
- **Evaluated against four aforementioned baselines**
  - Java built-in (JSL), Kryo, Skyway, Naos
- **Improving transmission phases against all baselines**
  - 2.28x compared with Naos



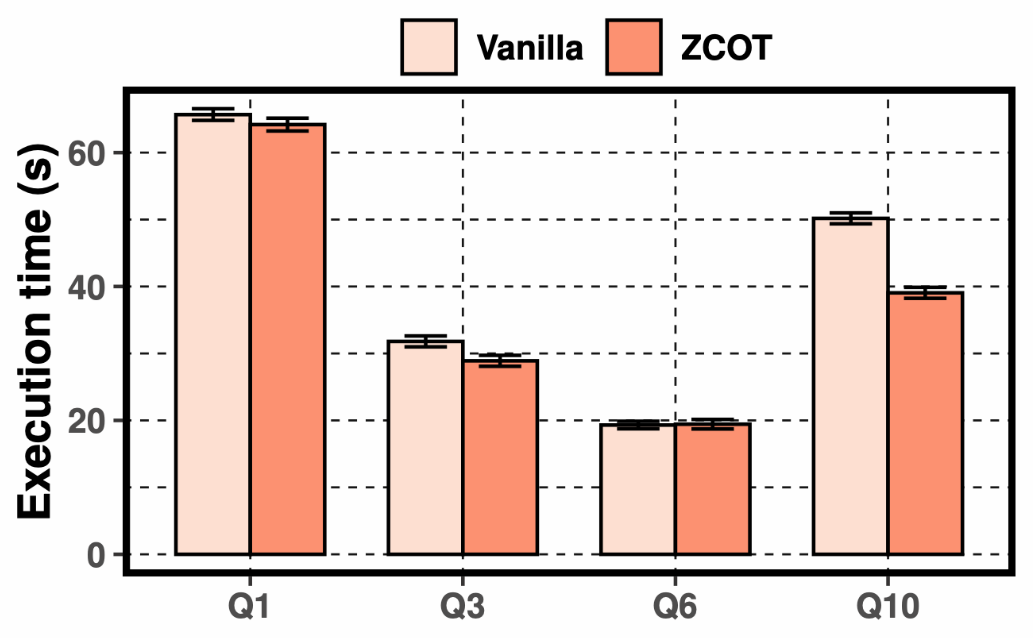
Spark Performance

- **Easy of integration**
  - Implementing a ZCSerializer in place of Kryo and JSL
  - Only contains 70 lines of code
- **Evaluation results**
  - 13.9% improvement against Kryo
  - 4.19x speedup in the write part
  - 2.95x in the read part



Flink Performance

- **Evaluated with four different queries in TPC-H**
  - 22.2% improvement at best (Q10)
  - Less improvement since Flink S/D is manually optimized



Conclusion

- **Data transmission is a costly phase in big-data analytics**
  - More severe in Java due to serialization/deserialization (S/D)

- **ZCOT: Zero-Change Object Transmission**
  - Sending and receiving objects through a shared exchange space
  - Remaining compatible with existing S/D interfaces
  - Significant speedup against S/D libraries

Thanks!

INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS