

# Efficient Message Serialization for Inter-Service Communication in dCache

Evaluating a Replacement for Java Serialization in *dCache*  
Lea Morschel for dCache Team, November 7 2019



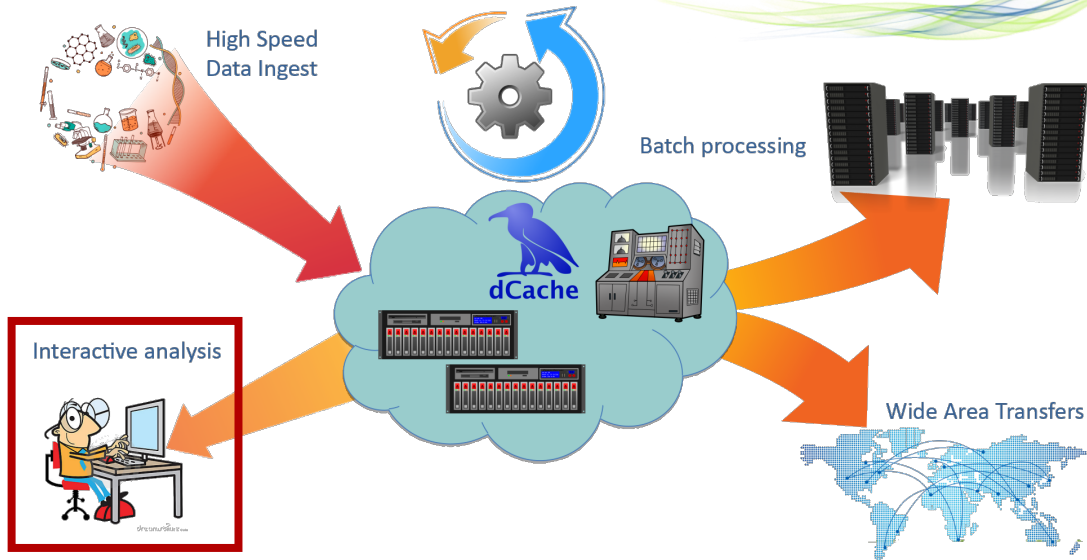
HELMHOLTZ

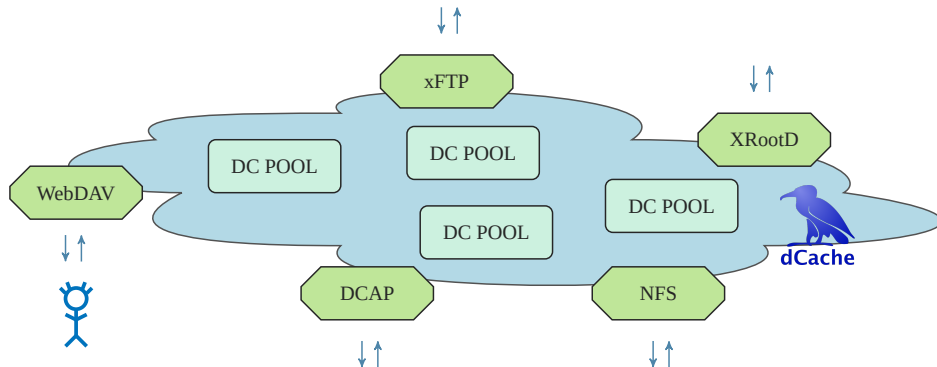
RESEARCH FOR  
GRAND CHALLENGES

- A distributed petabyte-scale storage system for scientific data
- Joint effort between DESY(2000), FNAL(2001) and NDGF(2006)
- Supports standard and HEP specific access protocols and authentication mechanisms
- Developed for HERA and Tevatron, used for LHC and others:
  - Belle II, LOFAR, CTA, IceCUBE, EU-XFEL, Petra3, DUNE, and many more



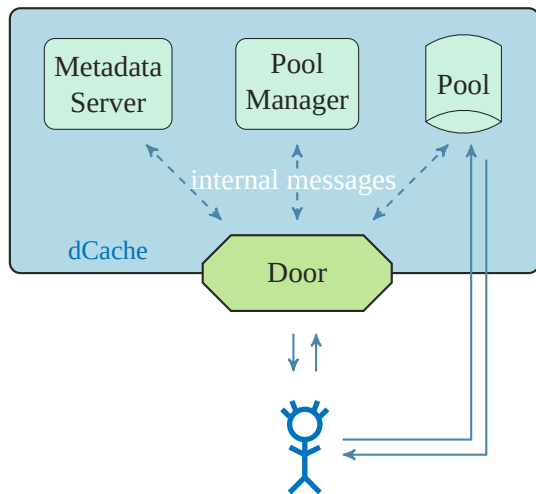
# Data Management & Workflow Control





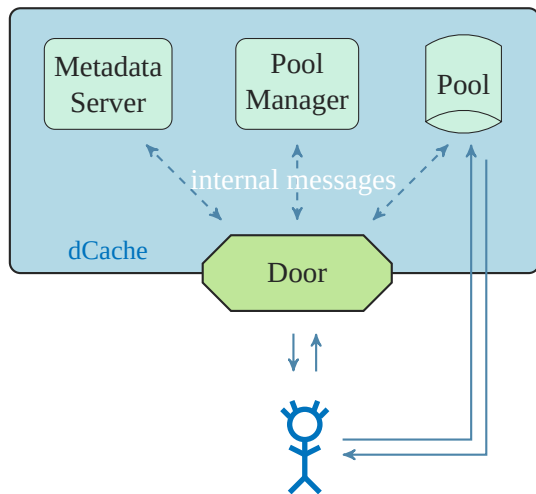
## Example: Accessing a File in dCache

- Example: single domain dCache
- User wants to read a file
  - Client communicates with his favorite access protocol door (e.g. WebDAV/NFS/...)
  - Door asks Metadata Server for information
  - Door asks Poolmanager for pool storing the file
  - Pool reference is returned to client for direct access (pNFS)



## => More Interactive Usage of dCache!

- Batch analysis → interactive usage of dCache (WebDAV, NFS)  
→ **Latency significant!**
- User request triggers multiple internal messages being sent  
→ Encoded and decoded!
- **GOAL:** Faster responses to user requests
- **APPROACH:** Make internal messaging faster by improving encoding/decoding speed



- dCache uses **Java Object Serialization (JOS)**: the native serialization protocol in Java
- **PROs:**
  - Trivial to first introduce and extend to new classes
  - To make a class serializable, just implement the `Serializable` interface
  - Serializes invisibly: `stream.writeObject(obj); stream.readObject(obj);`
- **CONs:**
  - Slow
  - Large encoded format (includes methods, not just state)
  - Difficult to make changes to existing serializable classes
  - JVM-specific! Cannot be interpreted outside of JVM languages

- Initial motivation for replacing current encoding method:  
→ **Speed** + possible **language independence**
- Survey among dCache developers in order to rate criteria for a new encoding protocol
  - Regarding **system functionality**
  - Regarding **development ease**

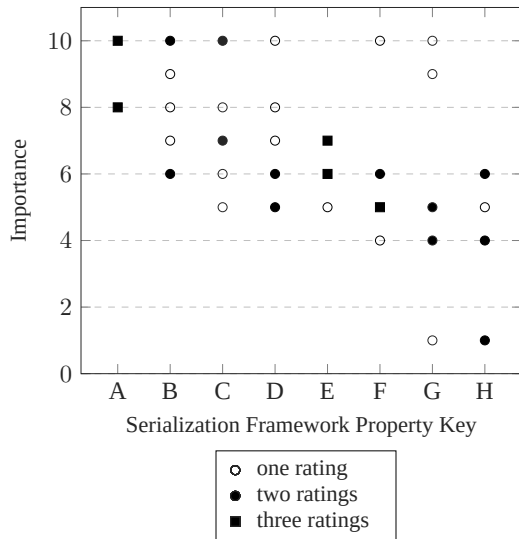
---

## Property

- Run in parallel with JOS
  - Speed improvements compared to JOS
  - Support for schema evolution
  - Introduction effort and maintainability
  - Documentation and gentle learning curve
  - Framework independence of a schema/an encoding format
  - Platform and language independence
  - Smaller serialized format than with JOS
-

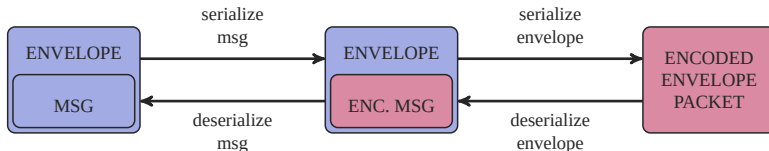


# Criteria for a New Serialization in dCache



| Label | Property  |
|-------|---|
| A     | Run in parallel with JOS                              |
| B     | Speed improvements compared to JOS                    |
| C     | Support for schema evolution                          |
| D     | Introduction effort and maintainability               |
| E     | Documentation and gentle learning curve               |
| F     | Framework independence of a schema/an encoding format |
| G     | Platform and language independence                    |
| H     | Smaller serialized format than with JOS               |

- Messages: **159** different, non-abstract message classes written in *Java*
  - Are used to exchange information regarding states and operations
  - Contain methods + data fields
- CellMessage envelope is always sent
  - Contains the payload message
  - May be (de)serialized independently for routing



- Representing data structures: abstract **SCHEMA + INSTANTIATION**
- Two types of serializers:
  - A.** Automatic schema inference: **full object graph serializer (FOGS)**
    - + Easy and intuitive to use and extend
    - Slower + larger encoding size + less control + may be vulnerable to deserialization attacks
  - B.** Explicit declaration of schema required: **schema-based serializer (SBS)**
    - + Faster + smaller encoding size + more control + safer
    - More complicated to introduce, use and create new serializable classes, may need extra compilation step, needs the used schema for decoding

- Eventually, one may need to change a serializable data structure (adding/removing/renaming fields, changing types, ...)  
→ Different versions of the same message may exist!
- **Backward compatibility:** deserializer can decode current and previous versions of messages  
e.g. Decoding stored serialized data
- **Forward Compatibility:** deserializer can decode current and future versions of messages  
e.g. Old microservice receives a message by a new one

- **Apache Avro (Avro)** – SBS, binary + JSON format, platform agnostic
- **Fast-serialization (FST)** – FOGS, binary, primarily Java bound
- **Hessian** – FOGS, binary, platform agnostic
- **Java Object Serialization (JOS)** – FOGS, binary, JVM bound
- **Kryo** – FOGS, binary, Java bound
- **Protocol Buffers (Protobuf)** – SBS, binary, platform agnostic
- **Protostuff Runtime (Protostuff)** – FOGS, binary, in theory platform agnostic

1. **Performance** (& encoding size) relative to data structure complexity
  - Goal: Create metric for classifying data structure complexity to evaluate speed/size in general for a protocol
  - Evaluate performance (& size) for each protocol + example messages
  - Generalize results
2. Support for **schema evolution**
3. **Qualitative** framework **features** (usability)
  - Created criteria according to the *Likert scale* (ratable [1, 5])
  - Rated each framework/protocol, evaluated (summarized) results

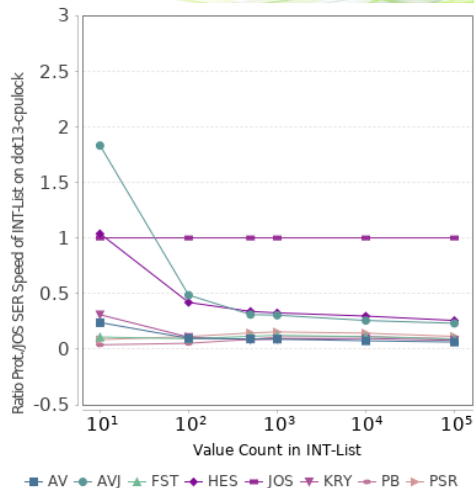
- Ultimate goal: A precise performance value/function for each protocol
- **Problems:**
  1. One can only ever benchmark one serializer on one input object  
→ How to **GENERALIZE PERFORMANCE** from results on independent inputs?
    - Several sets of structures with different analyzed parameters
  2. The computing environment will affect the measured performance  
→ How to **MINIMIZE influence of the ENVIRONMENT**?
    - Dedicated test hardware equivalent to production
    - Used quasi-standard JMH microbenchmarking tool
- Overall time for benchmarking took > 3400h → parallelization!

- Types of data structures with different fixed and variable parameters:
1. **TypeList** set: `IntList`, `DoubleList`, `StringList`
    - Six different sizes each: 10, 100, 500, 1000, 10000, 100000
    - Values randomly generated and stored to avoid fluctuations
  2. **Composites** set: `C0`, `C1` ... `C5`
    - Six different objects, filled the same every time
    - Pairwise comparable: contain nothing, basic types, equivalent class types, list/map types, ...
  3. **dCache-like** set: `PoolManagerPoolUpMessage`
    - One of the most frequent, regular messages with dCache: representative

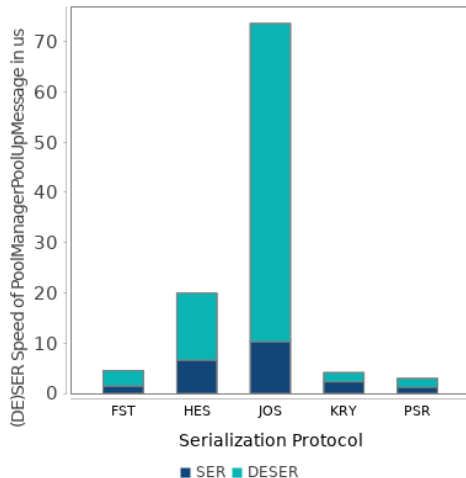
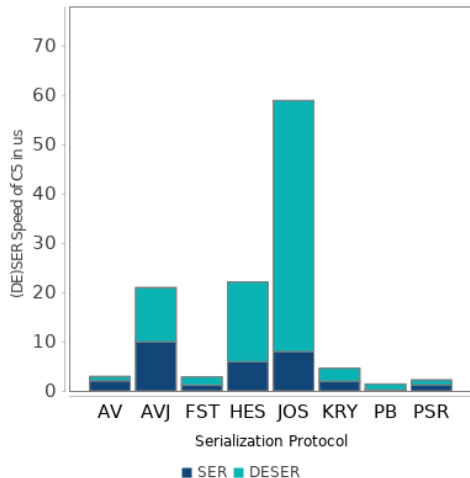


# Comparison of Protocol Performance

- Comparison of TypeList performance normalized by JOS
- All protocols are generally faster than JOS!
- Schema-based protocols are fastest, FOGSs or language independent formats are slowest



# Comparison of Protocol Performance: Composites



- **GOAL:** Faster serialization at a reasonable cost
- **RESULTS** of evaluation:
  - dCache message structures currently too complicated to use schema-based serializers (fastest ones!)  
→ Only consider FOGS for now
  - **FASTEST** FOGS: Protostuff-runtime, FST, Kryo
  - Best support for **SCHEMA EVOLUTION**: Protostuff, Kryo
  - Best **QUALITATIVE** features: Kryo

- Serializing **CellMessage** envelope using **protobuf**
- Current **payload messages** very complex
  - Serialize them using the FOGS **FST**
  - (De)serializing  $\sim 10\%$  faster!
- Gradually reducing complexity and number of messages
  - Eventually use **protobuf** for **message payloads** as well!?



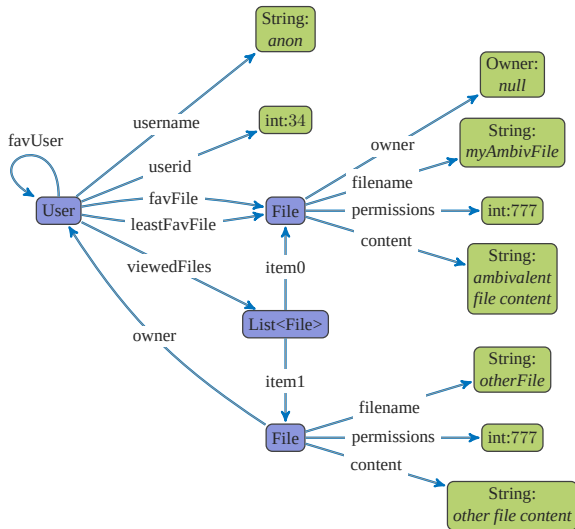
dreamstime.com

# Thank You!

# B A C K U P

- Example – Heartbeat PoolManagerPoolUpMessage sent by pools to the PoolManager:

```
1 public class PoolManagerPoolUpMessage extends PoolManagerMessage {
2     private final String      _poolName ;
3     private final long        _serialId ;
4     private final PoolCostInfo _poolCostInfo ;
5     private final PoolV2Mode   _mode;
6     // ... more fields and methods...
7 }
8
9 public class PoolCostInfo implements Serializable {
10     private PoolQueueInfo _store;
11     private PoolQueueInfo _restore;
12     // ... more fields and methods...
13 }
14
15 // ....
```



- Data types:
  - **BASIC**: single-valued
  - **COMPOSITE**: struct-like, lists, ..
- Object at runtime: directed **TREE**
  - May contain loops, repeated references → store references?
  - May contain null or a subclass in a class container

→ Language to encode object trees

- How to store type and field information?
- How are they encoded (space efficient, flexible, ...)



- **Encoding is optimized for different desired features:**  
size, speed, self-describing/schema necessary, blocks readable, ...
- Data types are differently represented in different systems  
→ How do we encode types? Example numerical types:
  - Optimize space usage by offering different sizes (e.g. `int32` & `int64` etc.) or using variable length encoding
  - Zigzag-Encoding: small size of all small absolute values (two's complement!)
- Often references converted to copies to be more efficient



- **General**
  - Multiple measurements to reduce errors in statistics
- **Hardware**
  - Compare measurements on same machine
  - Evaluate if relations are preserved on another machine
  - Disable hardware multithreading
- **Software**
  - Use containerized deployment (singularity) of jar-file
  - Lock benchmarker process to certain CPU: no hopping
  - Correct software benchmarking is difficult: use special tool and care!

- Java benchmarking is especially difficult due to **smart code execution** by the JVM
  - JIT compilation: JVM generates **optimized byte code only after a certain time** of execution, usually interprets it: deoptimization and recompilation effects  
→ Warmup period!
  - Many **optimizations**: loop unrolling, lock elision/fusing, constant folding, dead code elimination, method in-lining, on-stack replacements, ...  
→ Knowledge of how Java handles code to avoid certain pitfalls
- **Java Microbenchmark Harness (JMH)**
  - Makes it easier to avoid pitfalls, generates report

- Tests were conducted in three different environments:
  - **dot1**: Machine *dcache-dot1*, user context
  - **dot13**: Machine *dcache-dot13*, user context
  - **dot13cpulock**: Machine *dcache-dot13*, superuser, container CPU-locked

| Feature         | dcache-dot1                | dcache-dot13               |
|-----------------|----------------------------|----------------------------|
| Linux Kernel    | 3.10.0-862.14.4.el7.x86_64 | 3.10.0-957.21.3.el7.x86_64 |
| HEP-SPEC06 v1.2 | 125.58                     | 363.03                     |
| Processors      | 12 Intel(R) Xeon(R)        | 20 Intel(R) Xeon(R)        |
|                 | CPU E5-2440 0 @ 2.40GHz    | CPU E5-2660 v2 @ 2.20GHz   |
| Memory in kB    | 49243160                   | 131812036                  |

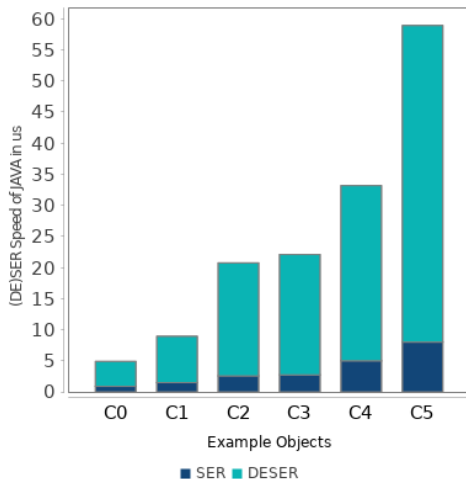
## Evaluating Performance

# Benchmarking Process

- Benchmarking serializing + deserializing all objects with each protocol
- Time for benchmarking one method (1 protocol + 1 object + ser/deser):  
 $2 * [ 4 * 10s \text{ (Warmup period)} + 10 * 100s \text{ (Measurement period)} ]$   
 $= T_{\text{bench}} = 2080s \text{ or } 34.6min$
- Time for benchmarking all Composites for 1 protocol:  
 $T_{\text{bench}} * 6 * 2 \text{ (ser+deser)} = 24960s \text{ or } 6.93h$
- Time for benchmarking all TypeList for 1 protocol:  
 $T_{\text{bench}} * 3 * 6 * 2 \text{ (ser+deser)} = 74880s \text{ or } 20.8h$
- Everything was done 5 times in each of the 3 environments for each of the 8 protocols! (**138.65d** without PMPUM)  
→ Running several benchmarkers in parallel, locked to individual CPUs

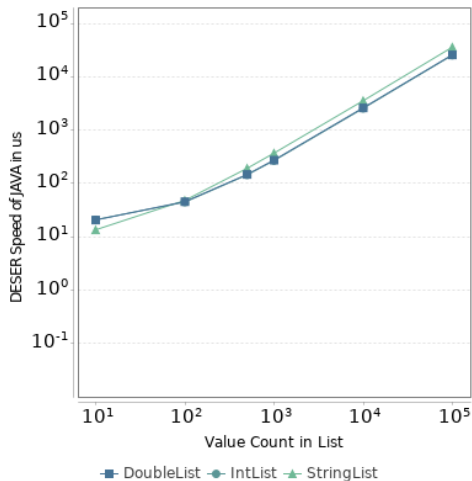
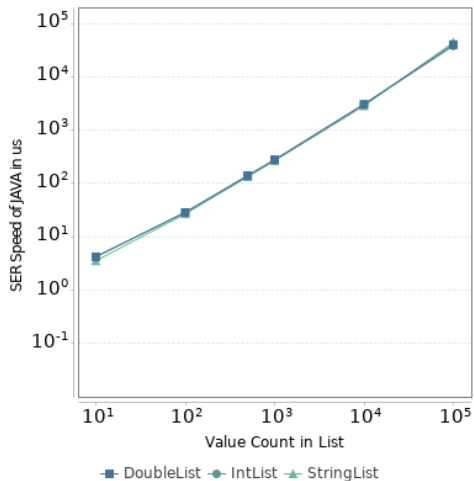
# Evaluating Individual Protocols

- Individual plots per protocol for each set
- Focus on protocol response to different inputs & ser/deser mode
  - Statistical uncertainties were found to be negligible
  - Deserialization in most cases much slower than serialization
  - Relative durations within sets not always similar for different protocols



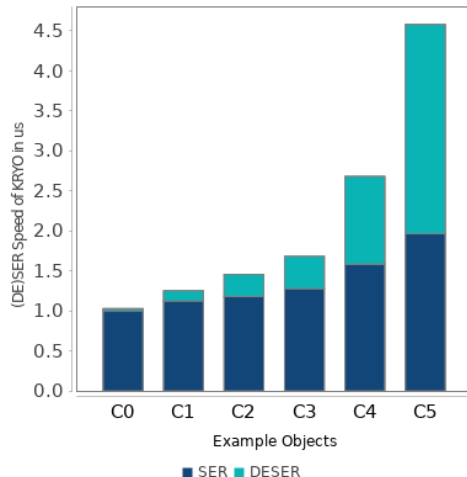
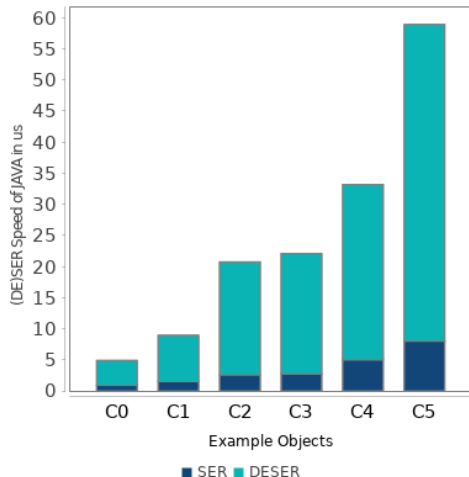
## Evaluating Performance

### Evaluating Individual Protocols



## Evaluating Performance

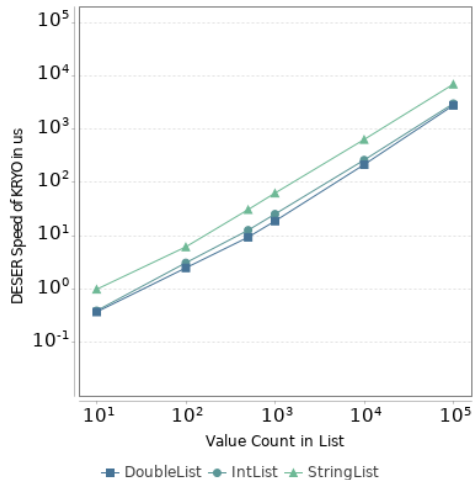
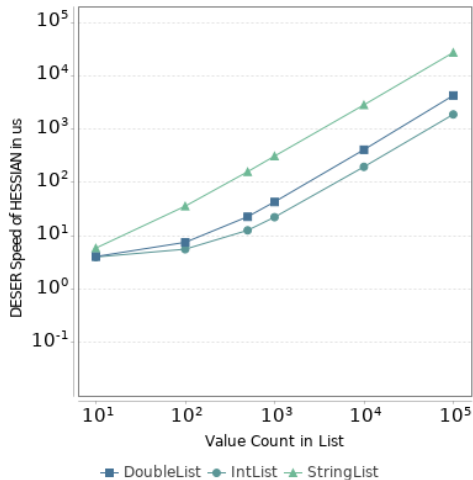
## Evaluating Individual Protocols: JOS vs Kryo





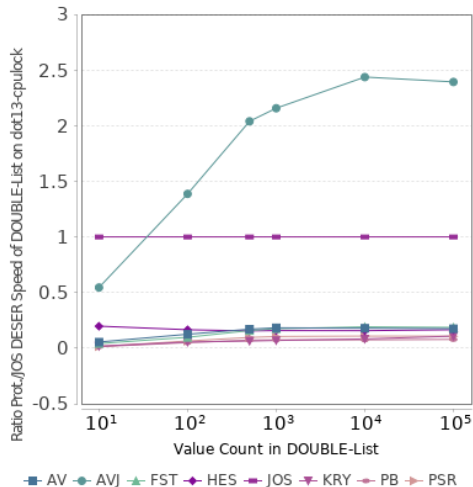
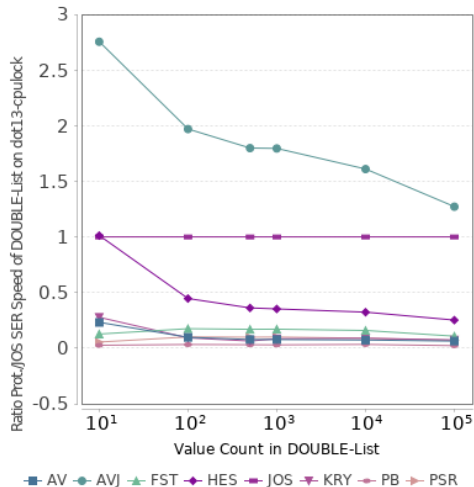
## Evaluating Performance

## Evaluating Individual Protocols: Hessian vs Kryo



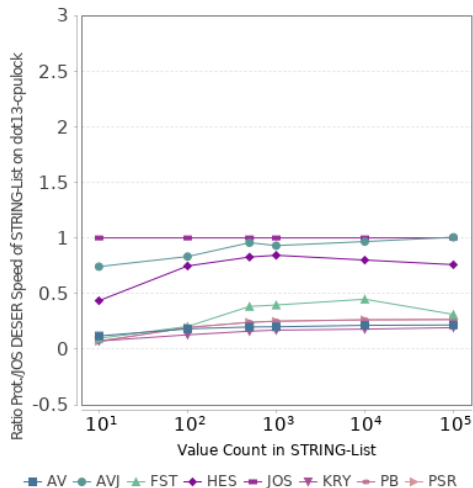
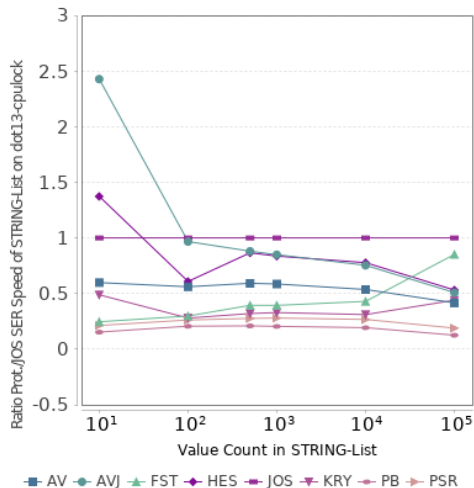
## Evaluating Performance

## Comparison of Protocol Performance: DoubleList



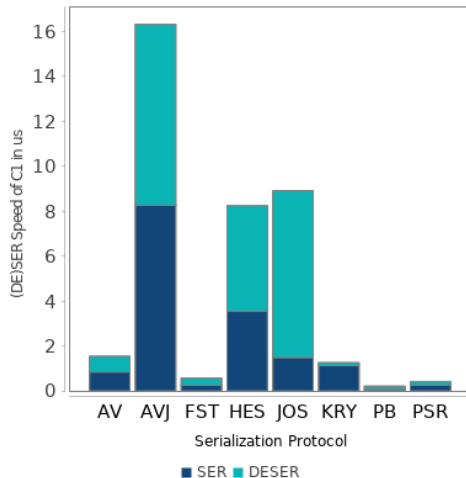
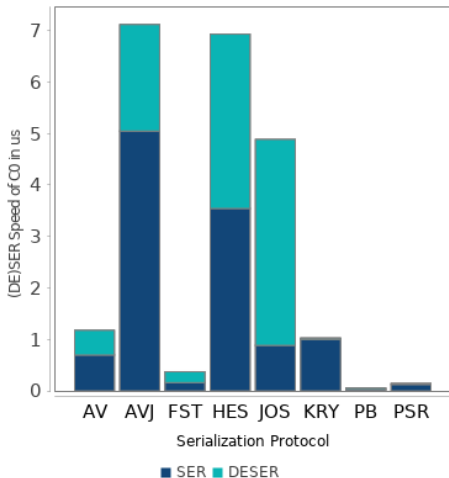
## Evaluating Performance

## Comparison of Protocol Performance: StringList



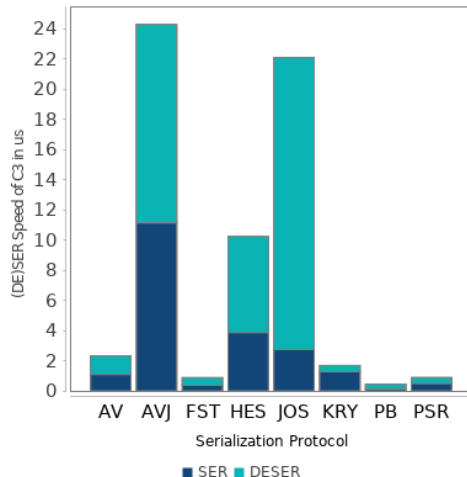
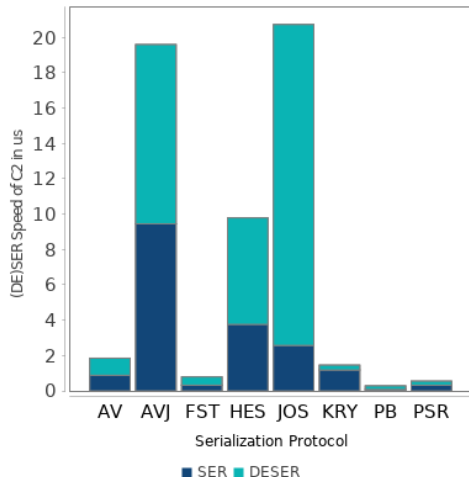
## Evaluating Performance

## Comparison of Protocol Performance: Composites



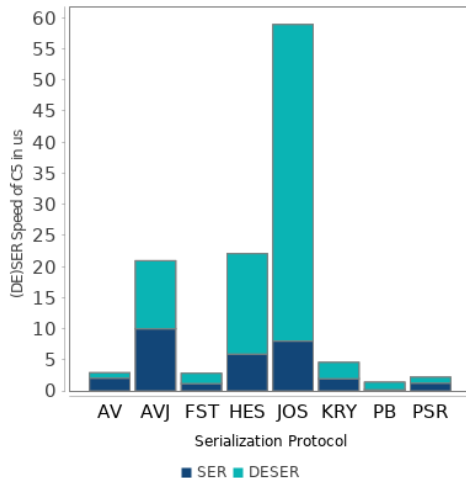
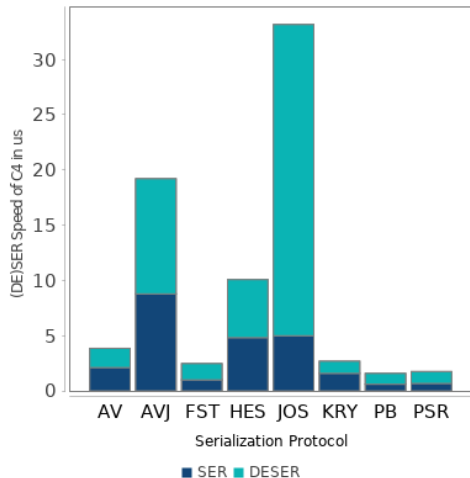
## Evaluating Performance

## Comparison of Protocol Performance: Composites



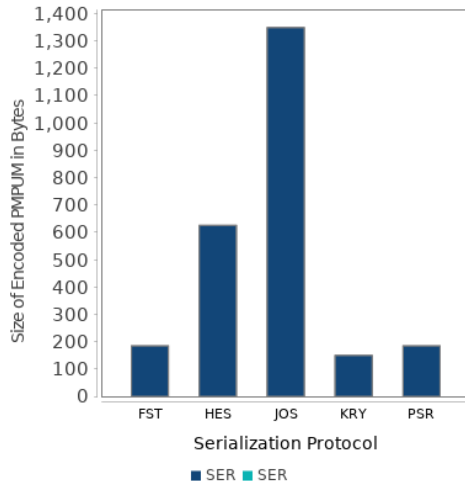
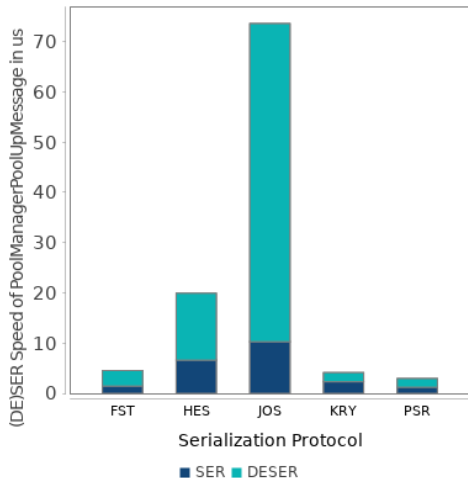
## Evaluating Performance

## Comparison of Protocol Performance: Composites



## Evaluating Performance

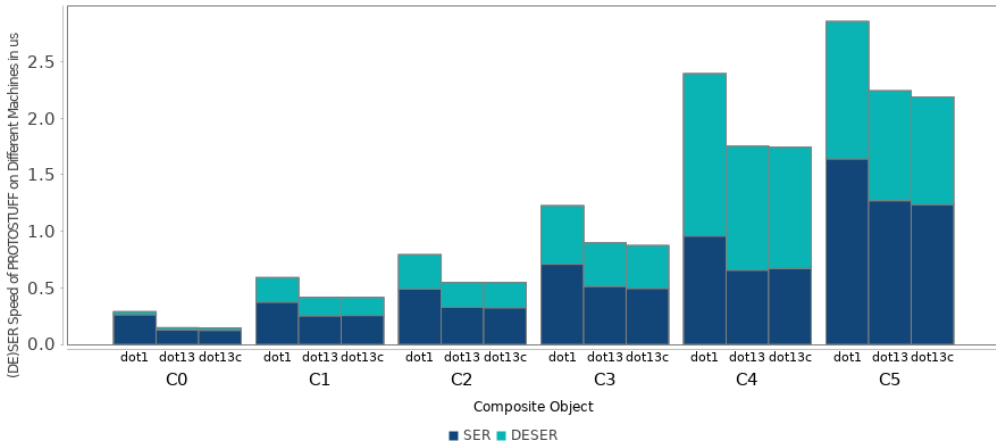
# Comparison of Protocols: PMPUM



## Evaluating Performance

# Influence of the Computing Environment

- Relative Performance was found to be preserved between machines/environments





# Real World: Using a Serialization Framework in dCache

- **PROBLEMS:**

- Protostuff could not handle **complexity of dcache-messages**
- Kryo needs to know **which class to deserialize**
- Ensuring backward compatibility: two serializers in parallel

- **SOLUTIONS:**

- Use **FST** for serializing
- Backward Compatibility:
  - Tag serialized bytestream, know which serializer was used/deserializer to use
  - Choose serialization method based on dCache version of communication endpoint
    - FST where possible!
    - FST has limited schema evolution support: Repack message for any endpoint with different dCache version! ⚡

- Headnode contains NFS door, database, PoolManager

