LOCO: Objects for Memory-Semantic Networks (WiP)

George Hodgkins, Joseph Izraelevitz

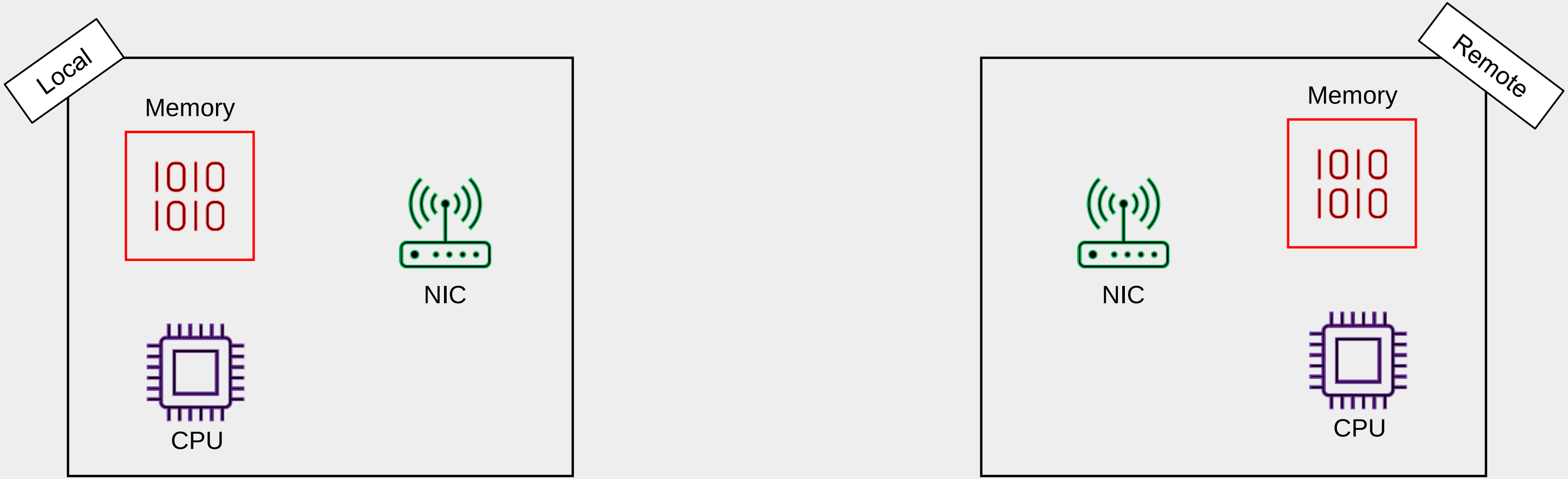
University of Colorado, Boulder

EMERALD 2024

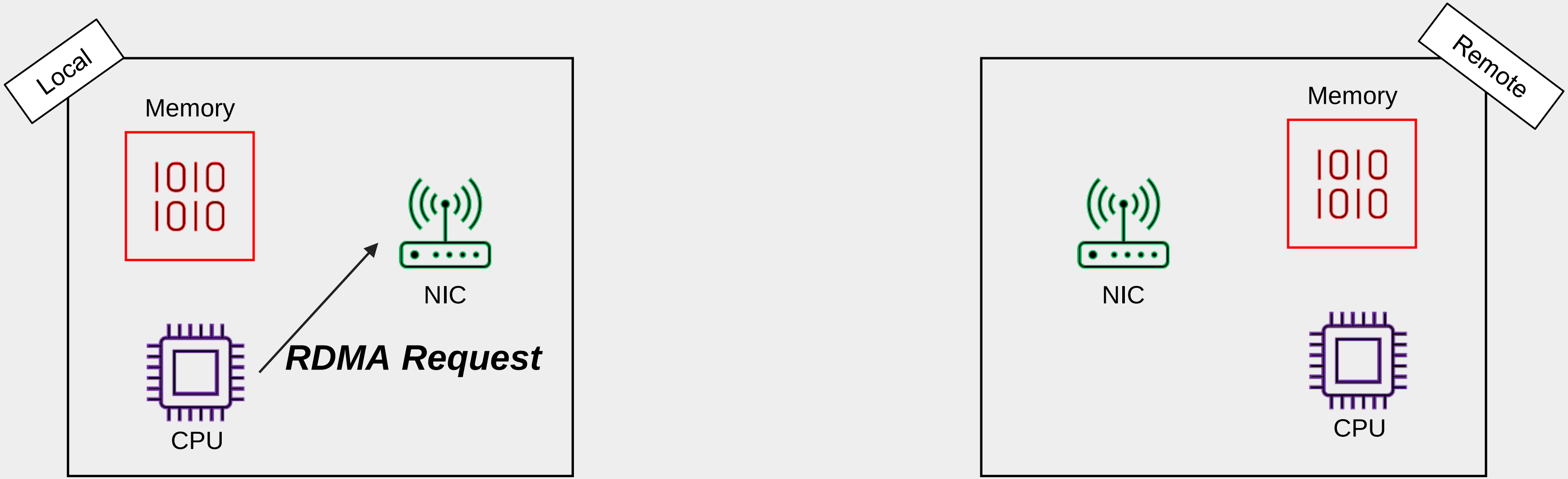
Agenda

1. **Motivation -** RDMA is fast, but hard to program
2. **Related work -** Existing abstractions are either too complex (hard to use) or overly simple (limited performance)
3. **Our contribution -** Objects are the solution

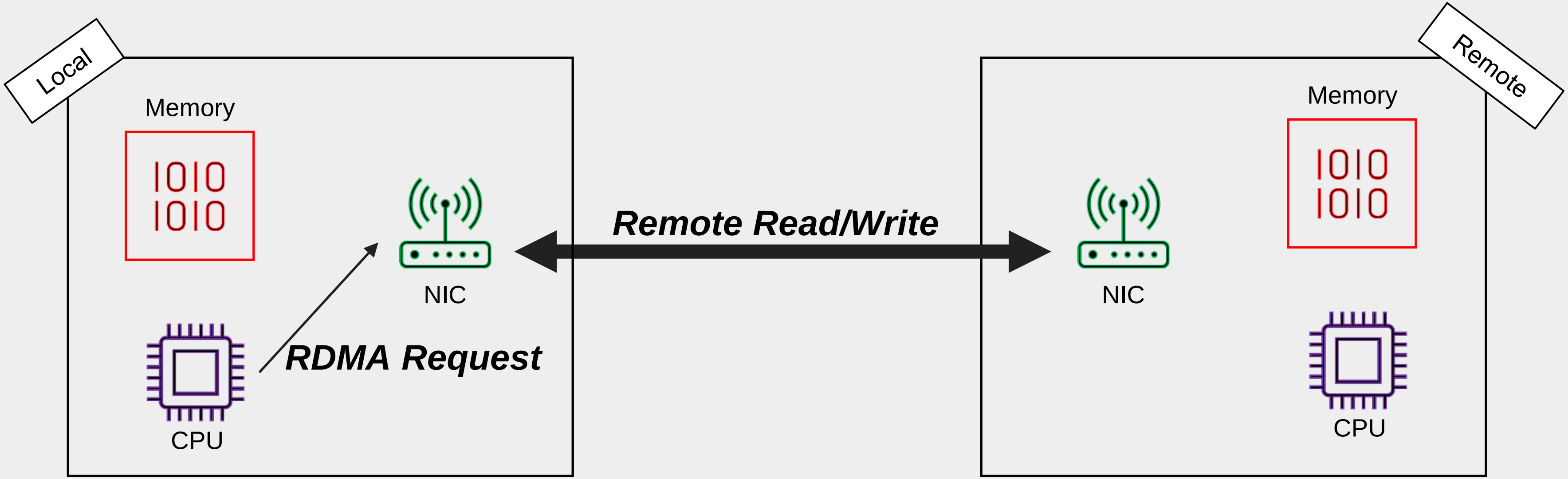
RDMA Overview



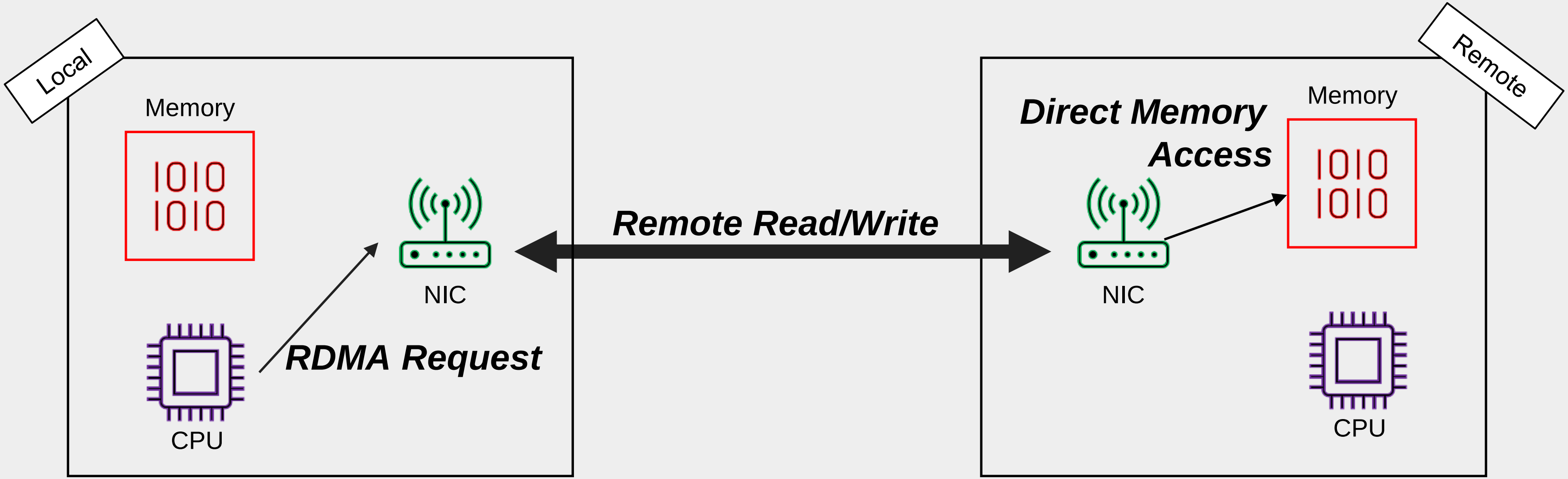
RDMA Overview



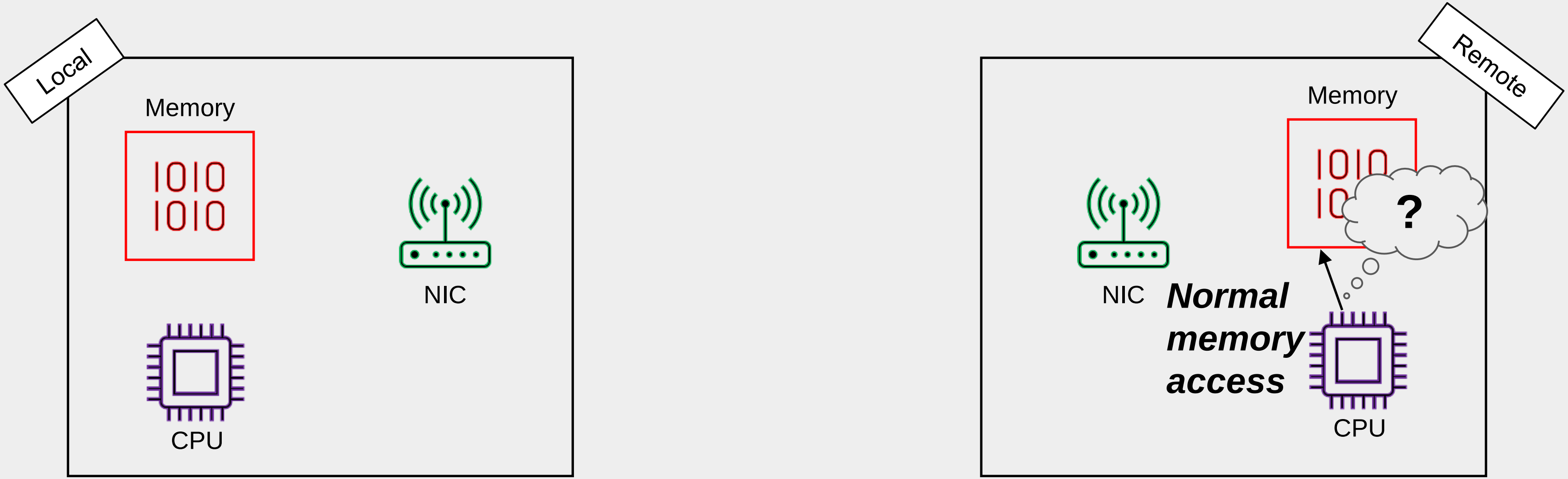
RDMA Overview



RDMA Overview



RDMA Overview



How is RDMA Used?

<u>Directly through native verbs API:</u>

- Complex interface
- Must reason carefully about consistency
- Often used to (re-)implement a specific existing application
- Ad-hoc usage limits setup flexibility, failure handling

How is RDMA Used?

Through MPI:

- Used on the backend to accelerate existing interface
- Can also directly allocate memory regions for read/write access
- ..but scalability is limited due to coarse-grained, non-customizable locking at the library level
- Other synchronization primitives (barriers) are library primitives as well

How is RDMA Used?

As a single coherent address space:

- Enforcing coherence & consistency limits performance
- Naively porting shared memory applications gives poor performance due to extremely non-uniform latency

Why Objects?

Objects are:

- **Encapsulated –** They hide complexity from the user in a controllable way
- **Composable and reusable –** Functionality can be reused and combined for new use cases
- **Intuitive –** An object model is a good fit for many applications

Existing object models

BCL: Berkeley Container Library (ICPP ’19)

- Containers built on a flat global address space
- Implemented on top of MPI or similar using a client-server model
- Containers are unique, neither reusable nor composable

HCL: Hermes Container Library (CLUSTER ‘20)

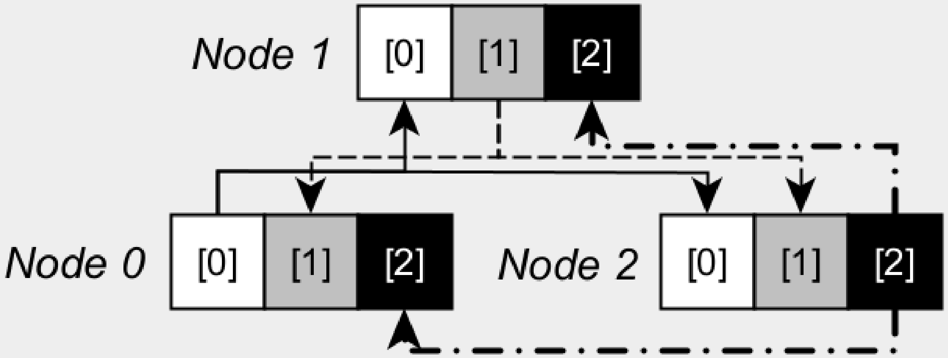
- Improved version of BCL
- Containers are now named and reusable, but still not composable
- Retains MPI backend & client-server model

LOCO Overview

- Object system built directly on verbs API
- Memory is accessed through typed primitives
- Objects are composable and reusable, referred to by name
- Symmetric peer model which supports dynamic join & drop (at both object and node level)
- Different object implementations can connect to each other to support asymmetric behavior

Barrier example: class members

```
class barrier : public loco::channel {
  unsigned count, num_nodes;
  loco::sst_var<unsigned> sst;
```



An SST with three participants. Arrows point from writers to readers.

Barrier example: owned_var

Each register in the SST is an `owned_var,` which provides single-writer atomicity using different strategies depending on the size of the underlying type:

- **≤ 8 bytes**: atomicity is guaranteed by the NIC-CPU interface
- **≤ 56 bytes**: sequence numbers written before and after each data write; reader retries if they do not match
- **> 56 bytes**: attach checksum to writes, reader retries if they do not match

Barrier example

Channel name

```
barrier(channel* parent,
  std::string name, manager& cm, int num) :
 channel(parent, name, cm,
  channel::expect_num(num-1)),
  sst(this, "sst", cm),
  count(0), num_nodes(num) {
    channel::join();
}
```

Number of participants

SST constructor

join() call

Barrier example

```
public:
void waiting() {
  // increment our counter
  count++;
  sst.store_mine(count);
  // acks used for memory consistency
  ack_key acks(mgr());
  // push local count to other nodes
  acks += sst.push_broadcast();
```

1. Increment counter

2. Broadcast new value

3. Wait for others

```
    // wait for other counters to match
    bool waiting = true;
    while(waiting){
      waiting = false;
      for (auto& row : sst) {
        if (row.load() < count) {
          waiting = true; break;
        }
      }
    }
    acks.wait();
    return;
  }
```

Barrier example

```
class barrier : public loco::channel {
  unsigned count, num_nodes;
  loco::sst_var<unsigned> sst;
```

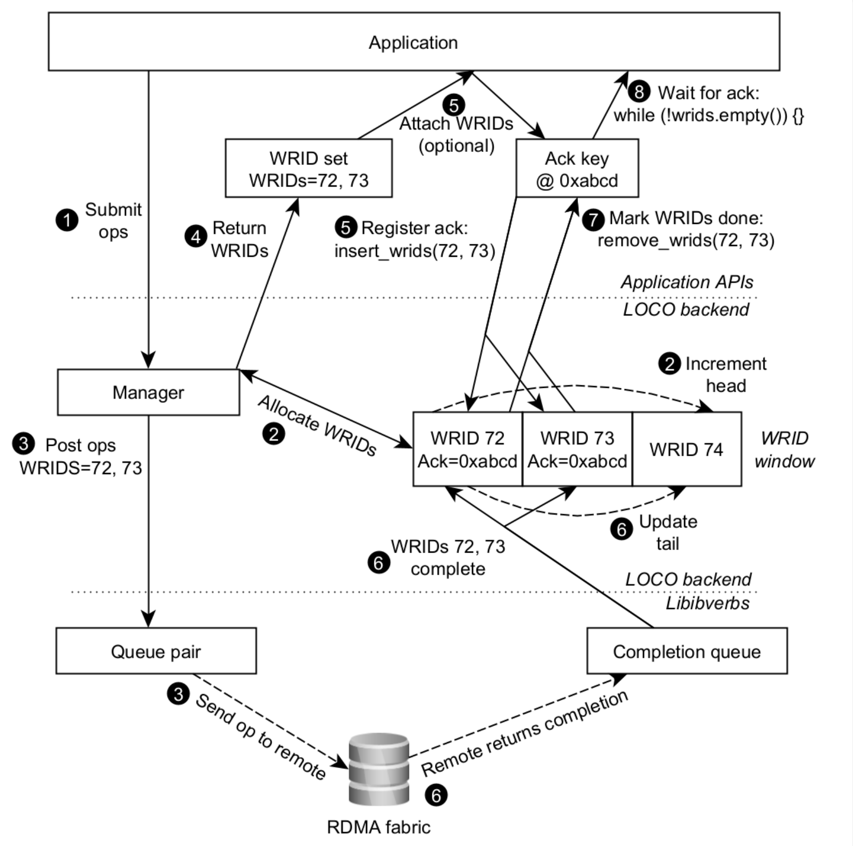
```
public:
void waiting() {
  // increment our counter
  count++;
  sst.store_mine(count);
  // acks used for memory consistency
  // see Section 5.1
  ack_key acks(mgr());
  // push local count to other nodes
  acks += sst.push_broadcast();

  // wait for other counters to match
  bool waiting = true;
  while(waiting){
    waiting = false;
    for (auto& row : sst) {
      if (row.load() < count) {
        waiting = true; break;
      }
    }
  }
  acks.wait();
  return;
}
```

```
  barrier(channel* parent,
    std::string name, manager& cm, int num) :
   channel(parent, name, cm,
    channel::expect_num(num-1)),
    sst(this, "sst", cm),
    count(0), num_nodes(num) {
      channel::join();
  }
```

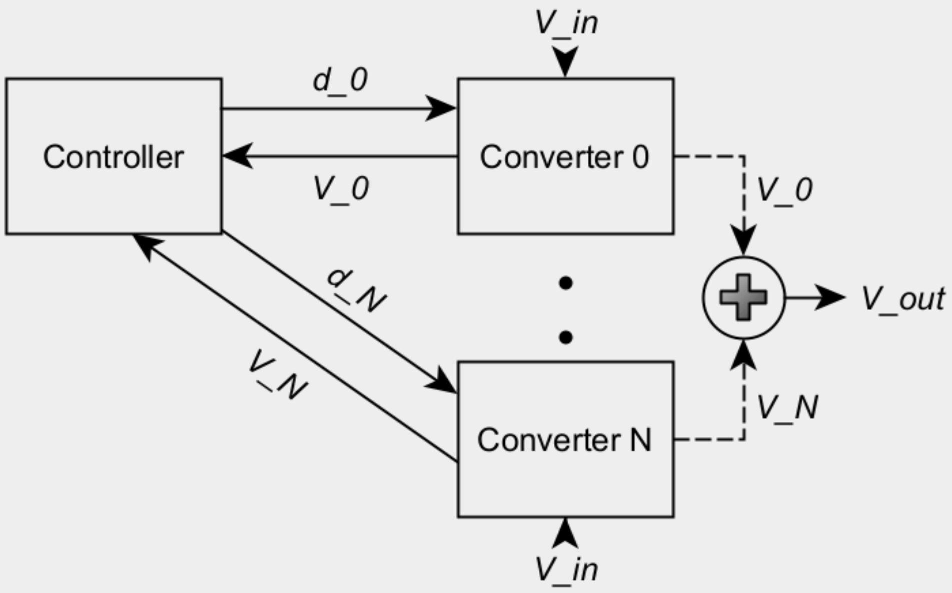
Ack keys

- An **ack key** is a pollable object representing used for monitoring the progress of one or more RDMA operations
- Each RDMA operation in LOCO corresponds to a 64-bit unsigned integer **work request ID (WRID)**
- A WRID can optionally be “attached” to an ack key to monitor progress of the corresponding operations
- The ack key is a bitset supporting lock-free insertion and removal
- The WRID is inserted in the bitset when attached, if not yet complete, and removed from the bitset when

Completion infrastructure



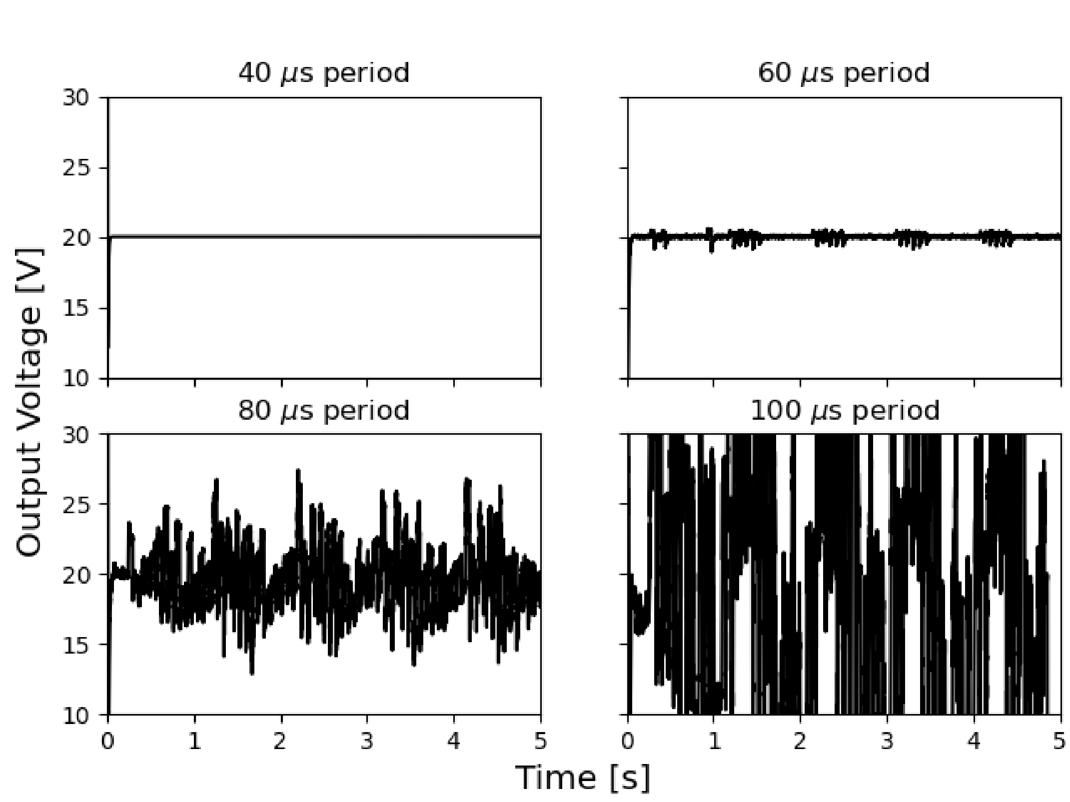
Distributed power converter



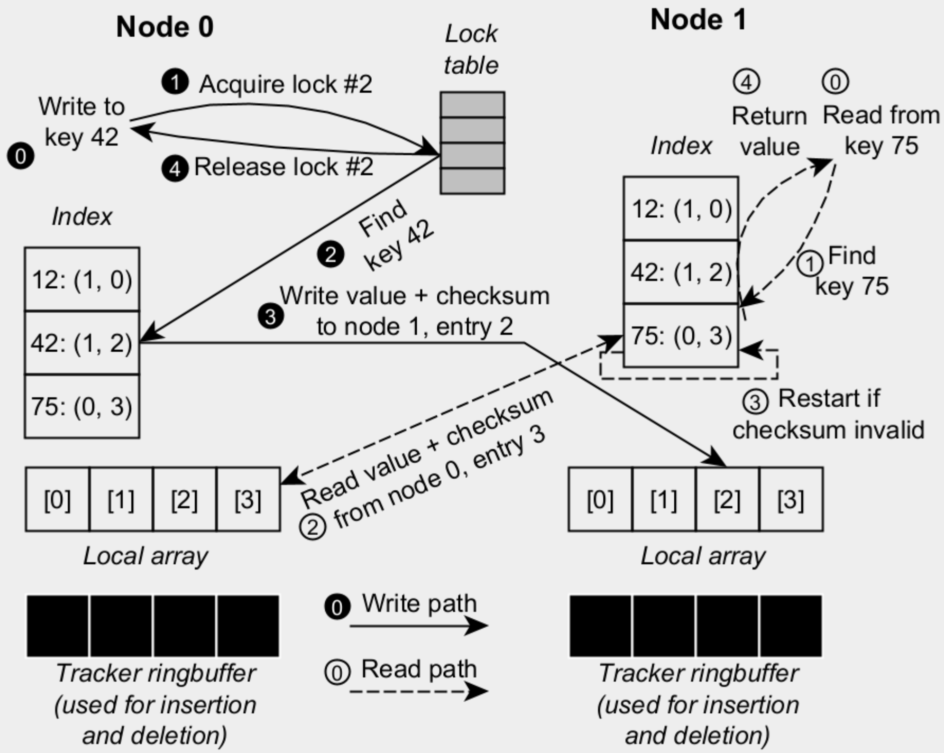
V_in: individual input voltage
V_out: aggregate output voltage

d_N: Duty cycle for converter N
V_N: Output voltage at converter N

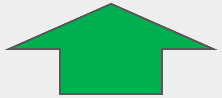
Power converter evaluation



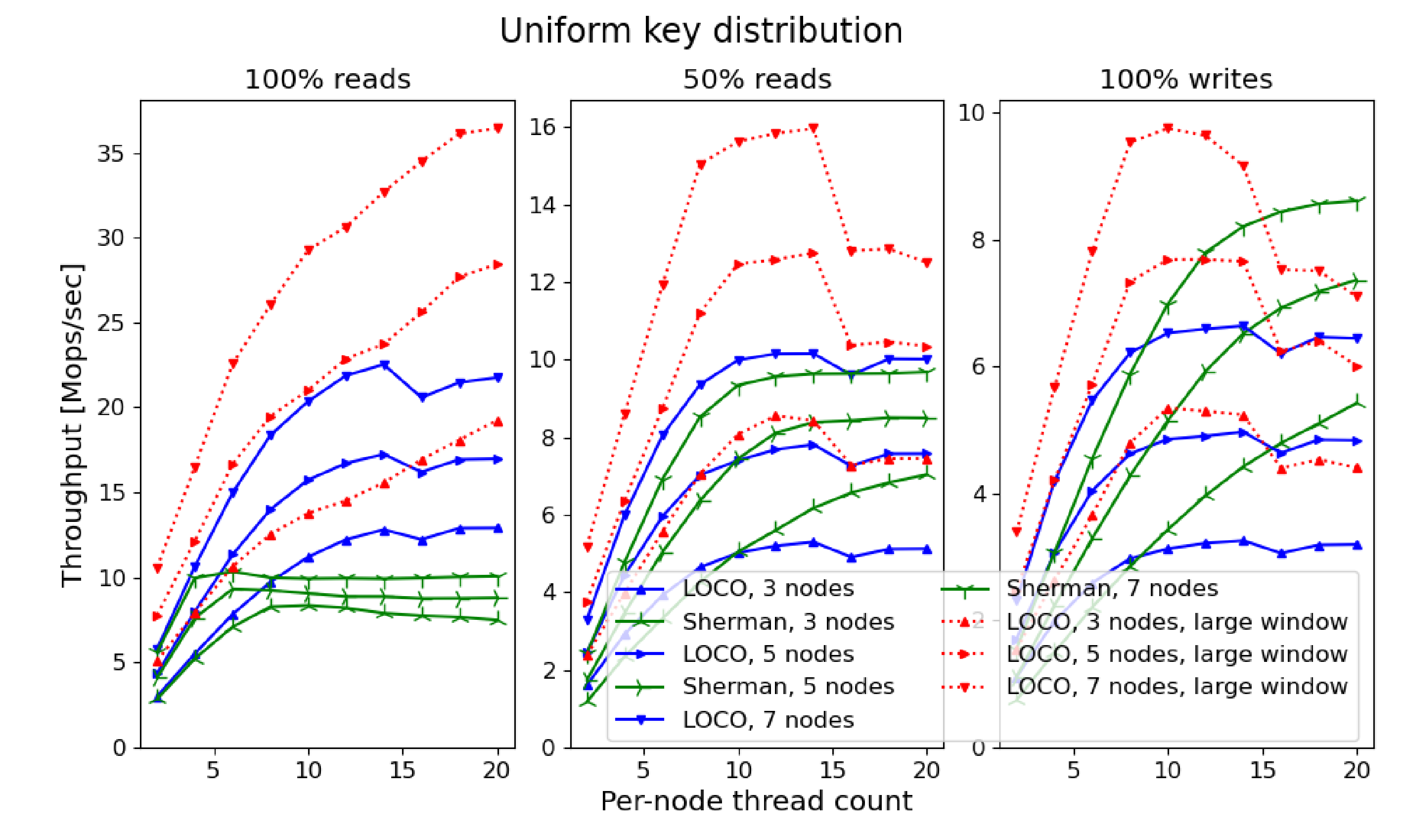
Distributed key-value store



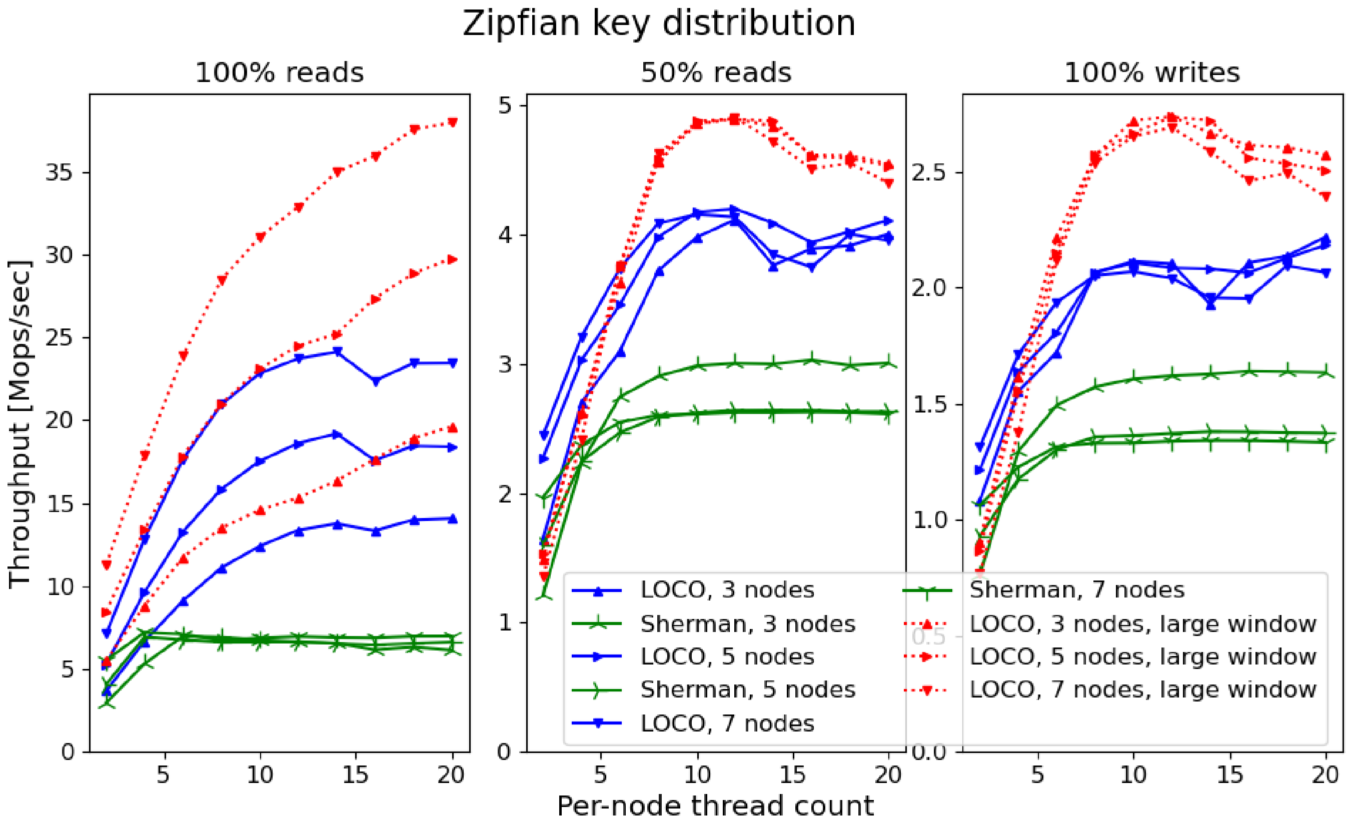
Key-value store evaluation

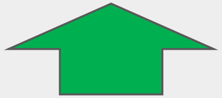


Up = faster



Key-value store evaluation





Up = faster

Summary

- RDMA is hard to program
- Existing abstractions limit performance or are difficult to use
- Objects present an attractive interface for hiding complexity while maintaining performance
- LOCO objects perform similarly to ad-hoc implementations

Extra