In Reference to RPC: It's Time to Add Distributed Memory

Stephanie Wang, Ben Hindman, Ion Stoica

Berkeley
UNIVERSITY OF CALIFORNIA

riselab
UC Berkeley

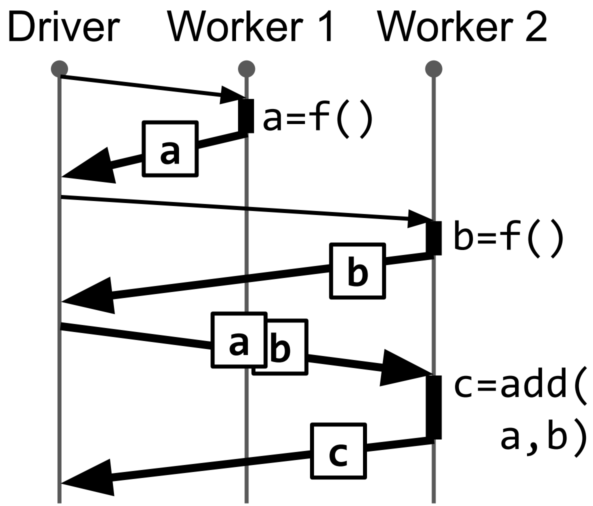
The need for distributed memory in RPC

The success of RPC

- RPC is used in virtually all distributed applications.
- Why is it so successful?
  - **Simple** semantics: all request/response values are *copied* → no shared state
  - Highly **efficient** implementations
  - **Interoperability** between RPC applications

Apache Thrift

The limitations of RPC: Pass-by-value

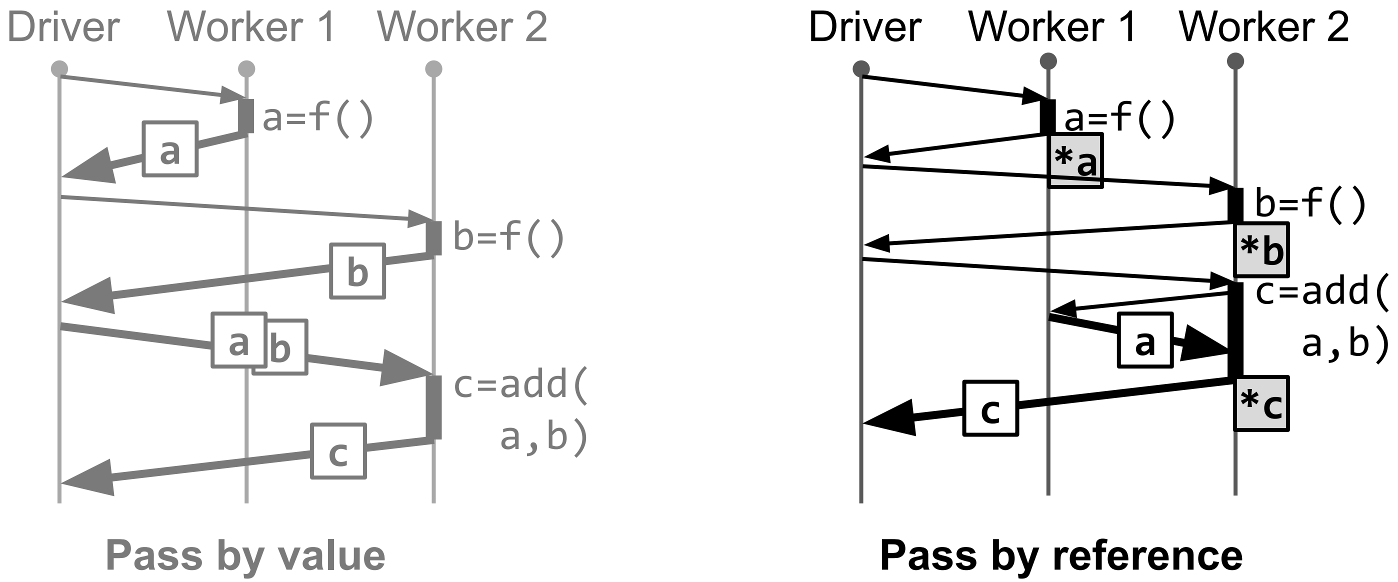


Program:

```
a = f()
b = f()
c = add(a, b)
```

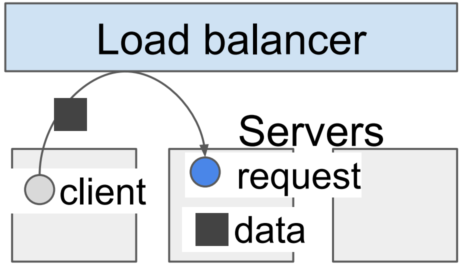
Expensive data movement

The limitations of RPC: Pass-by-value



Optimizing data movement at the application level

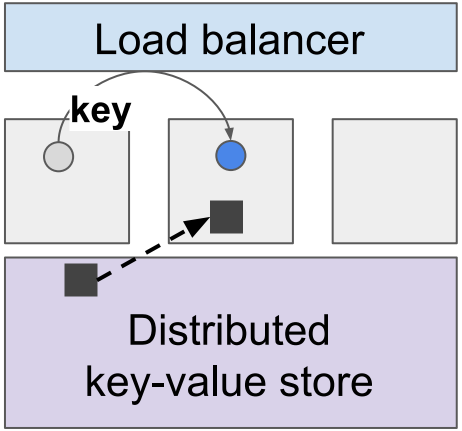
Pass-by-value RPC



\+ Simple memory management

\- Expensive data movement

Raw references



\+ Less data movement

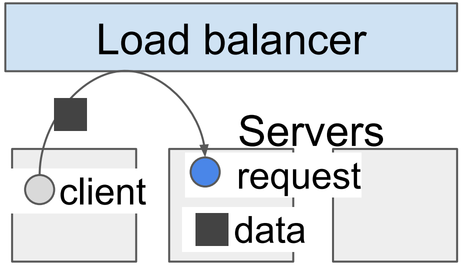
\- **Manual** memory management

Application-level references (“raw” references)

- Combine an existing RPC system with an existing key-value store
- **Application** functions call put/get on keys to store/retrieve values

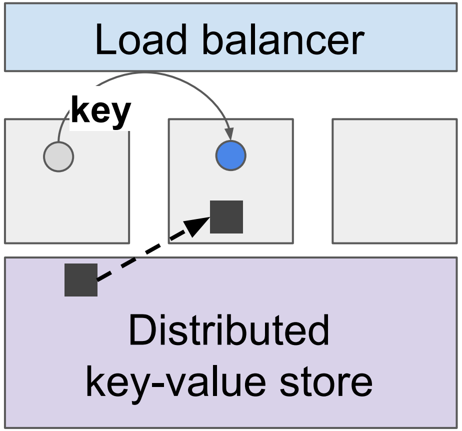
Optimizing data movement in specialized frameworks

Pass-by-value RPC



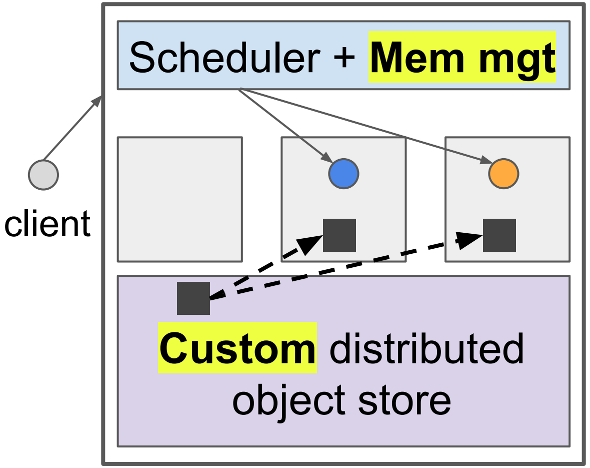
- \+ Simple memory management
- \- Expensive data movement
- \+ **Interoperability**

Raw references



- \+ Less data movement
- \- Manual memory management
- \+ **Interoperability**

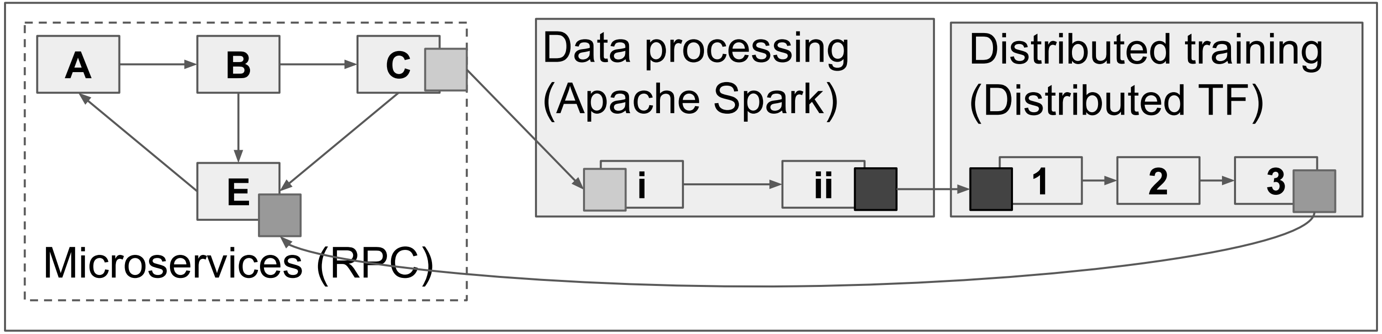
Specialized frameworks



- \+ Less data movement
- \+ Automatic memory management
- \- **Interoperability**

Why we need interoperability in data-intensive applications

- With no common foundation like RPC, data exchange must be addressed for every pair of applications or frameworks
- Stitching together applications often results in redundant copies of data and wrangling domain-specific data formats



A single "application"

Why we need interoperability in data-intensive applications

t be addressed for

a and

Problem: RPC for data-intensive applications

How would we re-design RPC for data-intensive applications? We want to:

1. Build applications like data processing directly on an RPC-like system, to enable **interoperability**.
2. Factor out **automatic memory management** to a common system, to reduce duplicated work and application burden.

Solution: Pass-by-reference RPC

1. Extend RPC with a **shared address space**.

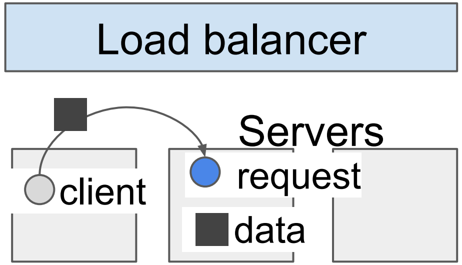
   → But doesn't a shared address space make RPC semantics more complex?

2. Make all values **immutable**, to preserve RPC's original semantics.
3. Introduce **references** as a first-class primitive in the RPC system.

   → RPC system is aware of references, including creation/destruction operations, request arguments passed by reference, etc.

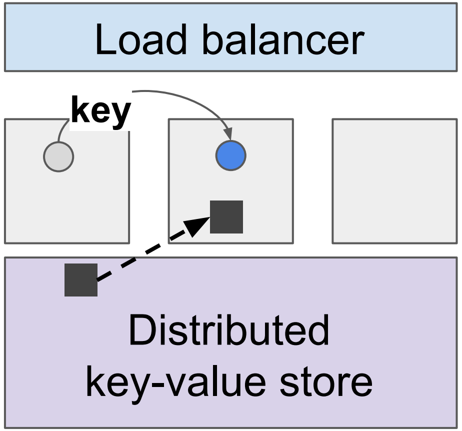
Solution: Pass-by-reference RPC

Pass-by-value RPC



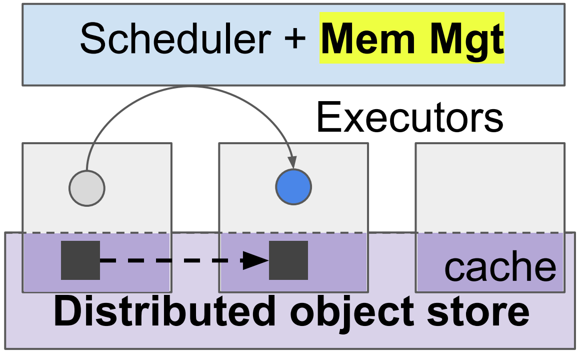
- \+ Simple memory management
- \- Expensive data movement
- \+ Interoperability

Raw references (app-level)



- \+ Less data movement
- \- **Manual** memory management
- \+ Interoperability

First-class references (system-level)



- \+ Less data movement
- \+ **Automatic** memory management
- **+ Interoperability**

First-class references for automatic memory management

A pass-by-reference RPC API

`f.remote(Ref r) → Ref`

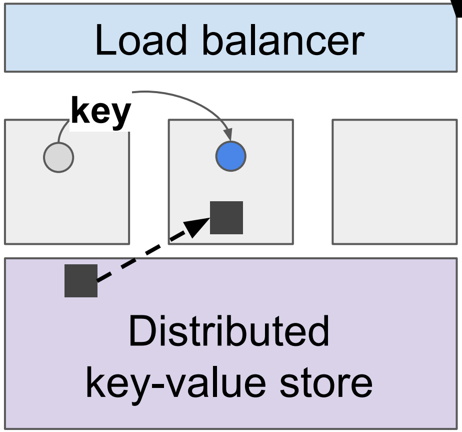
Invoke `f`.
Passes the argument by *reference*.
Returns a `Ref` that also acts as a *future* (a pointer to the eventual reply).

Why first-class references?

Pass-by-value RPC

- + Simple memory management
- + Interoperability
- \- Expensive data movement

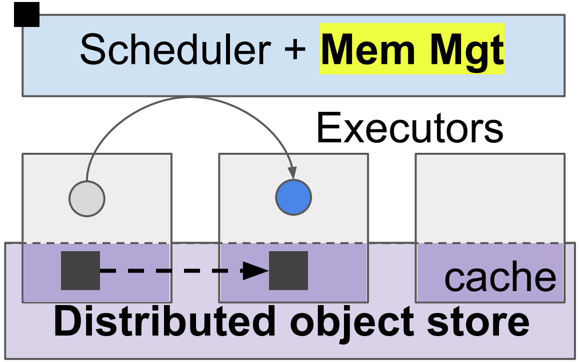
Raw references



- + Less data movement
- + Interoperability
- \- **Manual** memory management

VS.

First-class references



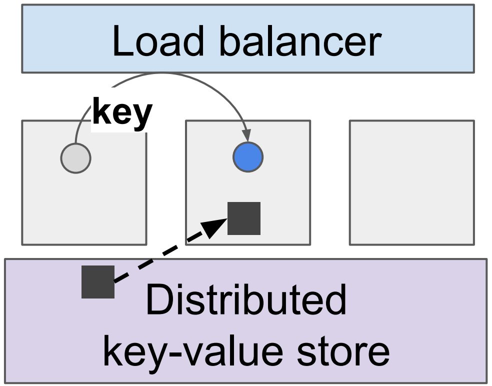
- + Less data movement
- + **Automatic** memory management
- **+ Interoperability**

Memory management operations

1. Allocation: Where to allocate a value?
2. Reclamation: When to deallocate a value?
3. Data movement: When/where to move a value?
4. Memory pressure: When memory is limited, ensure progress.

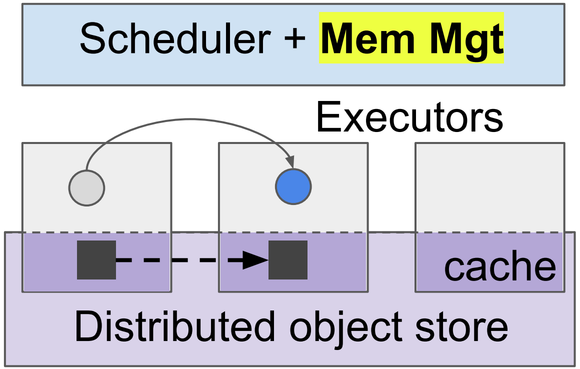
Allocation: Where to allocate a value?

Raw references



Application does not need to say where to put a key (key-value store decides where).

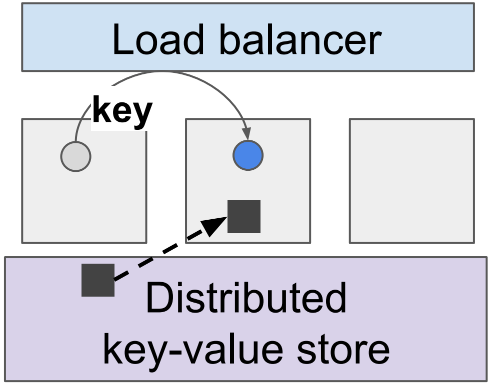
First-class references



Application does not need to say where to put a value (scheduler chooses function placement).

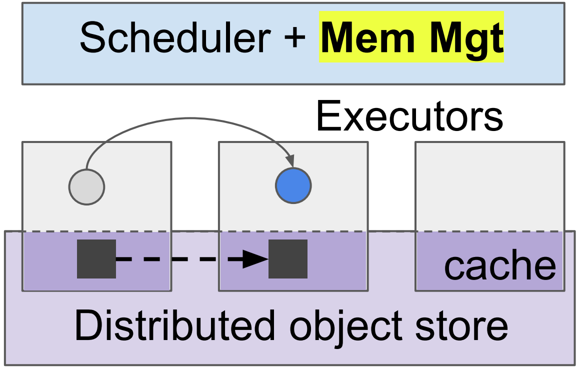
Reclamation: When to deallocate a value?

Raw references



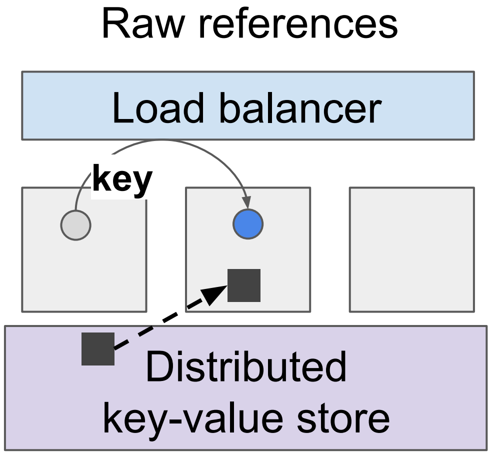
???

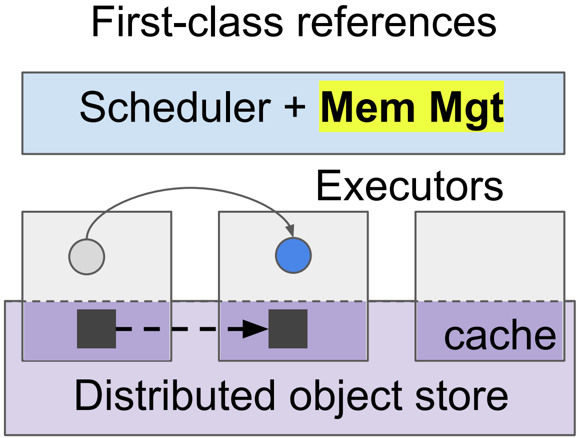
First-class references



System implements distributed garbage collection.

Data movement: When and where to move a value?





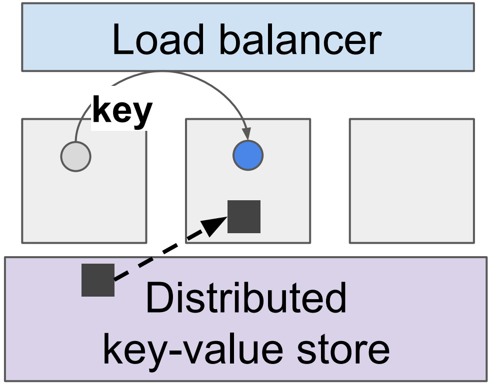
Key-value store can reduce some data movement, but the load balancer has no visibility into which keys will be requested.

Scheduler can optimize movement in the object store because each request's dependencies are visible. Examples:

- Data locality
- Pipelining I/O and compute

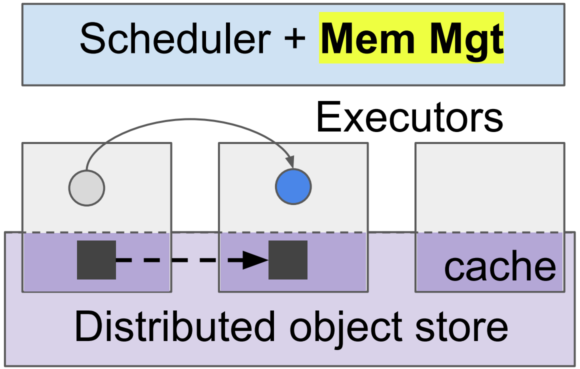
Memory pressure: Ensuring progress when memory is limited

Raw references



Too many requests fetching too-large values concurrently can trigger OS out-of-memory handling

First-class references



Scheduler can control memory usage by examining each request’s dependencies.

Why first-class references?

Enable automatic memory management:

- System is aware of all reference creation and destruction

  → memory safety and liveness

- System has visibility into each RPC's dependencies

  → optimizations in data movement and memory pressure

Conclusion

Data-intensive applications need a common system to enable **interoperability** and a common system for **automatic distributed memory management**.

We believe that pass-by-reference RPC is the right answer:

distributed memory + immutable data + first-class references

Check out the paper for information on:

- Pass-by-reference RPC in the wild (Ray and other systems)
- Application use cases
- Future challenges