Efficient Communication and Collection with Compact Normal Forms

ICFP 2015

Motivation 1

- Let’s consider sending a data structure to a peer. The peer may run in:
  - Another thread in the same process.
  - Another thread in the network, trusted to be running the same binary.
  - A trusted endpoint in the network, which may not run the same binary.
  - An untrusted endpoint across the network.

Principle 1

- To minimize serialization time, in-memory representation and network representation should be the same.

Problem 1: Continuous In-Memory Representation

- Prepare a memory region first (a block of memory, i.e. a page).

- Store data structures continuously in the region.

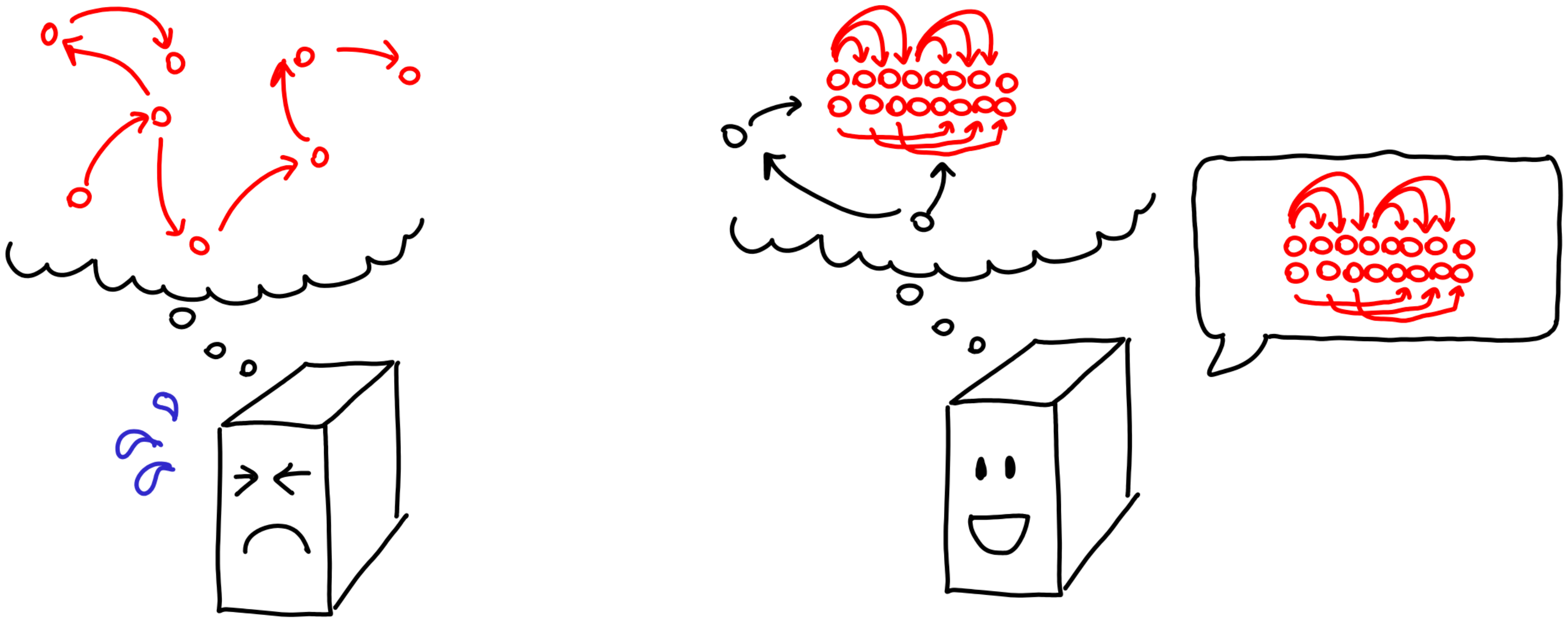
- In order to use the region, probably use this API:
  - sendBytes sock (buildTreeToRegion x)

Principle 2

- Copying is acceptable, as long as the copy is amortized across all sends of the same data.

Problem 2: Safety

- Data structures stored in a continuous region should not have out-bound pointers.



Principle 3

- Immutable data with no-outgoing pointers is highly desirable, from both a network transmission and a garbage collection standpoint.

Compact Normal Form

- getCompact :: Compact a -> a
- mkCompact :: IO(Compact ())
- appendCompact :: Compactable a => a -> Compact b -> IO(Compact a)
- sendCompact :: Socket -> Compact a -> IO ()

To send a binary tree using CNF

- do c <- newCompact (buildTree x)
  sendCompact sock c

You can even perform a test

- isCompact :: a -> IO (Maybe (Compact a))

Use case

```
do c <- mkCompact
   r1 <- appendCompact [3,2,1] c
   r2 <- appendCompact (4 : (getCompact r1)) c
```

Implementation

Old tricks for a new dog

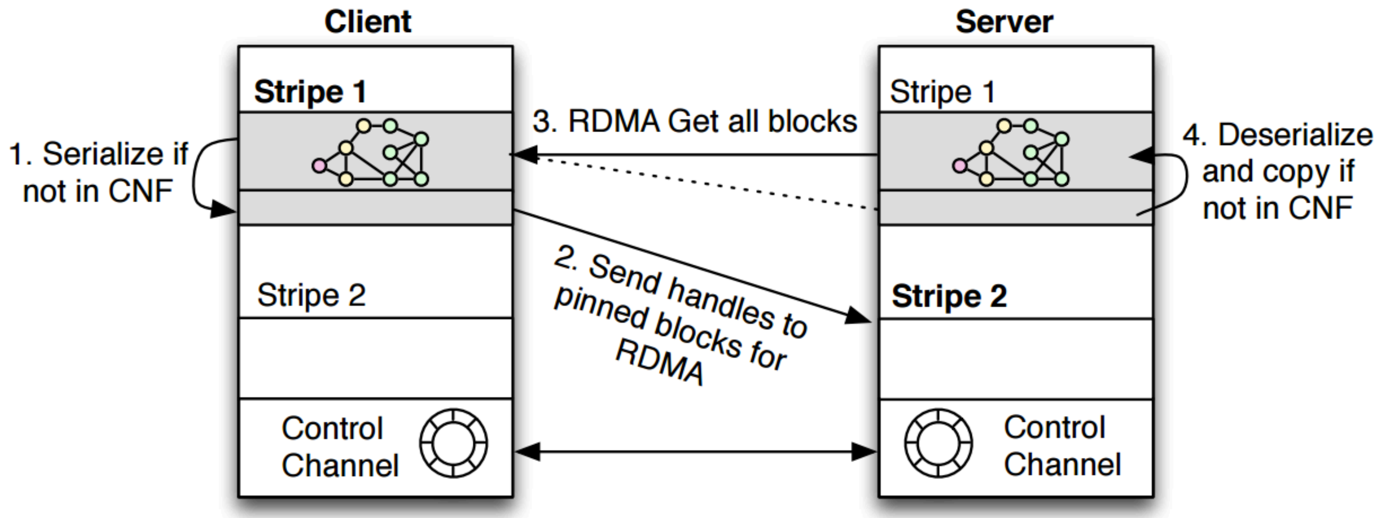
Partition the heap

Network Communication

- Serialization a compact region is fast:
  - All the blocks of the region
  - Length of each block
  - Pointer to the root of the data structure stored in the region

- Deserialization may be fast:
  - If the region is placed at the same memory address.
  - Otherwise need to adjust all the internal pointers.

Good for RDMA



(b) Rendezvous (pull-based) RDMA protocol. This is the zero-copy case where the client sends metadata of the tree to the server. The server pulls data using remote read into a stripe that it has reserved for the client so that no pointer fixups are required.

Serialization Performance

Table 1: Median latency for serialization with CNFs versus serialization with Haskell binary and Java, for the bintree data structure.

Size	Compact	Binary	Java
2^{23} leaves	0.322 s	6.929 s	12.72 s
2^{20} leaves	38.18 ms	0.837 s	1.222 s
2^{17} leaves	4.460 ms	104.1 ms	109 ms
2^{14} leaves	570 ns	8.38 ms	9.28 ms
2^{11} leaves	72.4 ns	255 ns	1.13 ms

Serialization Performance

Table 2: Serialized sizes of the selected datatypes using different methods.

Method	Type	Value Size	MBytes	Ratio
Compact	bintree	2^{23} leaves	320	1.00
Binary			80	0.25
Cereal			80	0.25
Java			160	0.50
Compact	pointtree	2^{23} leaves	512.01	1.00
Binary			272	0.53
Cereal			272	0.53
Java			400	0.78
Compact	twitter	1024MB	3527.97	1.00
Binary			897.25	0.25
Cereal			897.25	0.25
Java			978.15	0.28

Socket Communication Latency

Table 3: Median end-to-end latency for socket communication with CNFs versus serialization by Haskell binary and Java, for the different data structures bintree and pointtree.

Type	Size	Compact	Binary	Java
bintree	2^{23} leaves	3.180 s	6.98 s	9.595 s
	2^{20} leaves	382.4 ms	982 ms	837 ms
	2^{17} leaves	59.93 ms	100 ms	90 ms
	2^{14} leaves	8.380 ms	10.54 ms	11 ms
	2^{11} leaves	1.833 ms	1.238 ms	2 ms
pointtree	2^{23} leaves	4.978 s	23.58 s	15.71 s
	2^{20} leaves	624.0 ms	2.64 s	1.461 s
	2^{17} leaves	81.31 ms	321 ms	141 ms
	2^{14} leaves	13.3 ms	37.1 ms	35 ms
	2^{11} leaves	2.6 ms	4.33 ms	3 ms

RDMA Performance

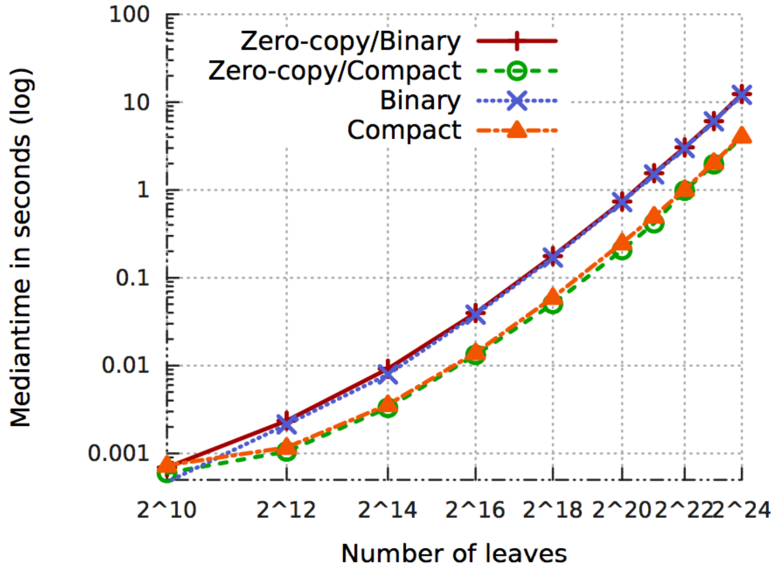


Figure 10: Median time it takes to send a `bintree` of varying tree depths from a client to the server using RDMA. At depth=26, it takes 48s to serialize, send and deserialize a 640MB `Binary` tree (for a throughput of 13MB/s), whereas it takes 16s for a 2.5GB `Compact` tree (for a throughput of 160MB/s).

In-memory KV Store

Table 4: Requests handled by server for varying database sizes. The size corresponds to the space used by values in the Haskell heap.

Keys	DB size	Binary	Compact
100	6.56 MB	17,081	69,570
1,000	65.6 MB	15,771	63,285
10,000	656 MB	15,295	57,008