Don't Let RPCs Constrain Your API

Daniel Bittman
UC Santa Cruz

Robert Soulé
Yale University

Ethan L. Miller
UC Santa Cruz
Pure Storage

Vishal Shrivastav
Purdue University

Pankaj Mehra
IEEE Member

Matthew Boisvert
UC Santa Cruz

Avi Silberschatz
Yale University

Peter Alvaro
UC Santa Cruz

ABSTRACT

As data becomes increasingly distributed, traditional RPC and data serialization limits performance, result in rigidity, and hamper expressivity. We believe that technology trends including high-density persistent memory, high-speed networks, and programmable switches make this the right time to revisit prior research on distributed shared memory, global addressing, and content-based networking. Our vision combines the code mobility of RPC with first-class data references in a global address space by co-designing the OS and the network around pervasive data identity. We have initial results showing the promise of the proposed co-design.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; **Distributed systems organizing principles**; *Abstraction, modeling and modularity*; • **Hardware** → Memory and dense storage; *Networking hardware*; • **Information systems** → Storage class memory.

ACM Reference Format:
Daniel Bittman, Robert Soulé, Ethan L. Miller, Vishal Shrivastav, Pankaj Mehra, Matthew Boisvert, Avi Silberschatz, and Peter Alvaro. 2021. Don't Let RPCs Constrain Your API. In *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21), November 10–12, 2021, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3484266.3487389

1 INTRODUCTION

Modular design is the bedrock of modern software development [24]. It improves programmer productivity by breaking design problems into smaller, re-usable parts that hide implementation details and are more easily debugged. In distributed systems, module composition is often realized via remote procedure calls (RPC). Decoupling components with RPCs allows them to scale independently—in principle, developers need only agree on a common interface and message format to leverage the benefits of software decoupling. Yet, in reality, RPCs enforce strict interface constraints and often trade adaptability (narrow interfaces are harder to evolve) for simplicity (narrow interfaces limit cross-component interactions), ultimately hampering the goal of scalability.

The chief problem with RPCs is that they are fundamentally location- and compute-centric: RPCs force a programmer to decouple an application by explicitly separating the computational endpoint or *location* where a function is invoked from the location where the function executes. As a consequence, they are well-suited to a relatively narrow set of use cases in which function arguments (which flow from invoker to executor) and returns (which flow back) must be serialized and sent in their entirety, and hence are small, and in which reference data must be located on the executor.

Many scenarios would benefit from decoupling but are simply not feasible using existing RPC mechanisms. For example, the invoking endpoint may have abundant data but limited compute, the invoker may wish to traverse a remote data structure, or the invoker may wish to refer to data that they lack privileges to read. In Section 2, we discuss how the increasingly important problem of distributed inference for edge devices can suffer from all of these problems. Rapidly growing model sizes, privacy concerns, and the proliferation of last-mile model customizations all exacerbate the issue.

To mitigate the problem of location-centric RPCs, data center operators often deploy discovery services, load balancers, or other forms of middleware [9, 12, 20, 28, 31]. These extra indirection layers make the execution endpoint abstract, but at the cost of increased latency and added system complexity. Moreover, we argue that such systems do not address the fundamental problem, which is *we need a more general mechanism for module composition in distributed systems.*

This mechanism must be more flexible than RPCs, but not at the cost of simplicity or performance. Satisfying these conflicting goals requires a shift in our programming models, from *location-centric* abstractions such as RPC to *data-centric* abstractions more akin to distributed shared memory (DSM). These data-centric abstractions can free programmers from

HotNets '21, November 10–12, 2021, Virtual Event, United Kingdom

ACM ISBN 978-1-4503-9087-3/21/11.
https://doi.org/10.1145/3484266.3487389

infrastructure-level concerns such as explicit data movement, caching, prefetching, *etc.*

There has been renewed interest in DSM, due to technology trends such as 100 Gig NICs, programmable switches with throughputs upwards of 10Tb/s, and high-density, byte-addressable, persistent memory. Reconsidering DSM designs at this time is not, in our view, controversial. After all, while referencing remote memory incurs 100× higher latency than accessing local DRAM [14, 15, 23], it is 100× *faster* than accessing local SSD[13, 23]. DSM allows a comparatively unconstrained model for data sharing, easily supporting patterns RPC does not. However, DSM is insufficient to subsume RPC, since the DSM model has no notion of code mobility.

We argue for a clean-slate design that combines the code mobility of RPC with the expressiveness of DSM-like solutions. Our design wraps all code and data in the system within a single global address space, and makes data references a first-class abstraction. This allows applications to invoke functions without the inefficient call-by-value semantics of RPC. At the network level, our design advocates for a convergence of memory and network protocols. We forward and route based on explicit identifiers that correspond to object addresses. To realize this, we co-design the network and OS to share a common language for data and code references.

We have begun prototyping by combining the Twizzler OS [4], which is designed for global references, with recent work on Packet Subscriptions [17] to facilitate network-understood data identity. We report early results in Section 4.

2 MOTIVATING EXAMPLE

To illustrate the poor fit of RPC as a decoupling mechanism for some classes of applications, consider an example from the distributed inference problem for edge devices. Here, sensors in mobile devices with modest processing and storage resources (*e.g.* mobile phones or autonomous vehicles) are the source of observations used both for training and inference. Recent work has focused on decoupling and distributing machine learning training across edge and cloud resources to minimize client-perceived latency, provide privacy guarantees, and maximize server-side throughput [19, 30].

In this example, we focus on the inference problem that arises in response to device input. Ideally, small models trained in the cloud (via a methodology such as federated learning) are periodically shipped in their entirety to edge devices, which perform local inference. Several trends are upsetting this model. The first is the aggressive growth—roughly 10×/year—of models, in particular language models. In 2018, the largest machine translation models at Google were 8.3 billion parameters [29]; a mere two years later, the largest models exceeded 800 billion parameters! Inference on sparse giant models which far exceed device resources must be performed server side, where model serving presents a substantial throughput bottleneck. This is further compounded by last-mile model customization for end users, in which inference tasks for different devices *must* be performed on slightly different models. As much as 70% of the processing time [8] for these model-serving applications is spent deserializing and loading the sparse personalized models into main memory at request time. Finally, users prefer local models remain local due to confidentiality concerns.

Consider a concrete example that is bedeviled by all of these complexities at once. A mobile device, Alice, in possession of a locally-trained model and an activation, wishes to perform a classification task that requires a partition of a sparse global model, located on cloud resource Bob. Further, imagine that Alice cannot perform the inference locally, either because the global model fragment is too large or because she has inadequate local compute. Finally, imagine that Bob is overloaded, while a separate cloud resource, Carol, is mostly idle. Figure 1, also discussed in Section 3, shows this.

An ideal solution will minimize the latency Alice perceives and maximize the throughput offered by both Bob and Carol, all while satisfying capacity constraints of each. It is easy to see that while this application requires decoupling, RPC is the wrong abstraction in terms of performance, expressivity, and flexibility. Data movement—whether from storage to DRAM on Bob or from Bob to Carol—requires costly serialization. If moving the data to Carol and performing inference there is the optimal solution, the application logic on Alice must orchestrate this infrastructure-level concern, either by pushing the data through Alice (Figure 1.1) or having the RPC executed on Carol address Bob directly and pull (Figure 1.2), adding complexity. Further, heterogeneity among end devices makes the "hard-coded" data movement strategy brittle: a subsequent classification request from client device Dave will be forced to run inference on the server side even if it is equipped with the resources to do the work locally.

A mechanism like RPC, in which movement of computation is explicit and movement of data implicit and limited, is a poor fit for any such use case, which we will generalize in Section 3. We need a flexible mechanism in which both code and data are mobile, but in which the application programmer need not make the movement of *either* explicit.

Patterns of RPC. While RPC is a poor fit here, there are applications for which RPC provides adequate decoupling. RPC shines in situations where decoupling in the application meshes well with having little data movement, where an RPC endpoint either fronts large data, large compute relative to the invoker, or some combination, with *small* arguments and return *values*. But call-by-small-value is a significant constraint, and there are many classes of applications that do not fit. We cannot paper over this problem, either. Because RPC is disconnected from a global notion of data identity,

we either *cannot* do call-by-reference (because references must cross machine boundaries) or we must shoehorn this functionality into the application logic and the RPC's APIs, resulting in brittle, repetitive, complex code to deal with the coordination, caching, and prefetching that comes from distributed data and global references when data moves.

3 OUR VISION

Our core vision is to combine the code mobility of RPC with the flexibility offered by DSM-like models in a global address space with global references as a first-class abstraction. By imbuing data with fundamental identity and pushing an understanding of data references into the OS and the network, we can leverage aspects of content-based networks to reduce the coordination typically required in a shared, distributed address space. The programmer is then free to express their computation through references to code to run on some *references* to data, instead of needing to serialize and copy *values* for arguments. Today, developers are often forced to implement functionality such as caching, prefetching, and manual data movement in preparation for some operation. With data references as a common language between the OS, the network, and applications, we can move this infrastructure-level functionality out of the application and back *into the infrastructure* where it belongs.

3.1 Computation and References

First-class support for call-by-reference instead of by-value allows an invoker to refer to data that *they do not have locally*. This allows greater freedom in the interface design between decoupled components. Figure 1 shows possible solutions to a problem like the example in Section 2, wherein an operation is scheduled to run on Carol using data currently located on Bob. Part (1) shows the naïve approach where the invoker copies the data locally (step i) before forwarding it to Carol (step ii) and eventually invoking the intended computation (step iii). In an attempt to alleviate an unnecessary copy, we could implement an additional RPC on Carol that allows it to copy data over from Bob itself (steps i and ii in part (2)) before we finally invoke our computation. Both (1) and (2) require additional logic on Alice to work—the programmer had to perform the infrastructure level task of data movement. Fundamentally, these issues arise because the system's core abstraction is location-based—the programmer is forced to manually orchestrate machines or resort to costly copying.

Figure 1 part (3) is closer to our vision. The first step is for the computation to move to Carol instead of first moving data in preparation. This may seem like a minor point, but it is not—by specifying *up-front* the computation we want to perform, we open the door for lower-level optimization to examine our requests before we go around manually moving

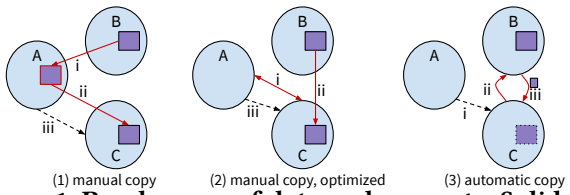

Figure 1: Rendezvous of data and compute. Solid red arrows are additional infrastructure-level tasks that are not fundamental to the requested computation.

data. In fact, in our model the programmer would not be directly asking Carol to perform the computation; instead the placement decision would be made by the system. Once the code starts executing, we can then move data on demand instead of having to move the entire object. Of course, the implementation is significant—without a global address space like the one we are proposing, implementing (3) would require brittle code that either tightly couples Carol and Bob or forces Alice to participate in the data movement by asking it to provide a location-based reference to the data on Bob[1].

As discussed previously, the "good" use case of RPC is one where code and data co-location has already been preordained by initial decoupling (after which it is rigid) and data transfer is minimal—often manifesting as something like a fronted key-value store service. This restricts code mobility (as it accounts for no change to decoupling later), and requires a myriad of RPC calls to implement all the ways a programmer might wish to view data (one need only look at the many S3 APIs available as an example). If we limit ourselves to traditional RPC, any situation (such as the one discussed in Section 2) that does not fit this pattern either results in expensive data movement and complex application logic, or it must be dismissed altogether and the application redesigned. By allowing applications to pass data references instead of just values, and by making data references a first-class abstraction in the OS and the network, applications become much simpler to express efficiently, even for what would be considered pathological cases for RPC.

Serialization. In traditional host- and process-centric systems, virtual memory spaces are private to a program instance and thus addresses are too. As a result, we serialize any data that passes between hosts (and sometimes even processes) because memory addresses on one host do not translate to any other host. However, in our model, pointers refer to data within the *global* address space, and thus a data structure containing pointers can be copied from one host to another with merely a byte-level copy, alleviating 100% of the loading overhead discussed in Section 2 and leaving only data transfer costs, which are fundamental. These transfer costs, however, can now be included in cost-models when

[1] Aside from the complexity, this also opens up TOCTTOU errors.

making placement decisions more easily, as they do not need to take the additional loading time into account.

The vision we have proposed here is, to be sure, non trivial. We will need to build it atop a number of new services, protocols, and low-level OS support. In the rest of this section, we will discuss how we plan to realize our goals through designs for reducing coordination and pushing identity and references into the OS and the network.

Global Addressing. Our primary goal with introducing a global address space is to reduce coordination. The idea of a single, large, and sparse address space is not new—single address space OSes have existed for some time [7, 10]. However, while the jump to 64 bit addresses was significant in the 90's, it was still insufficient to realize a distributed single address space [3]. Instead, we are extending Twizzler [4], which is designed around pointers that are invariant of local contexts like processors and hosts. Thus it has a system-level understanding of data references—precisely the abstraction we wish to push into the network for a global address space.

Instead of providing a small 64 bit address space or a type of `host:address` DSM-style addressing scheme, we will expose a 128 bit *object identifier* space, where objects are flat regions of memory that can be offset into. Objects act like pools of memory where smaller data structures can be placed, with references being encoded efficiently (see below) to either data in the same, or another, object. Thus data structures can be encoded in a machine- and process-independent format; in Twizzler, this facilitates orthogonal persistence, while we plan to use this feature for cheap data movement and call-by-reference across machine boundaries.

Pointers in Twizzler are encoded efficiently, such that the pointer itself takes up only 64 bits. This is done by having a separate table in each object, at a known location in the object, contain a list of external object IDs that the object has references to. A pointer encodes an index into this table along with an offset into the object, forming a 64 bit pointer that nonetheless references data in a 128 bit address space. This scheme has numerous benefits, and more details are available in the original paper [4]. One benefit for our model is that this table offers a translucent view into application semantics by way of a reachability graph for each object. This graph can be used by the system to perform prefetching based on data identity and actual reachability instead of some proxy for identity (*e.g.*, adjacency, as is used today).

The advantage of a large object ID space is reduced coordination. A space of 128 bits does not require a centralized arbiter to hand out new IDs when creating objects. Instead, programs can allocate new objects through numerous methods. Currently, Twizzler allocates object IDs in a flat namespace using secure random numbers, which meets our needs in that the chance of collision is vanishingly small.

3.2 Pushing Identity Into the Network

To realize this vision, the network needs to evolve towards a convergence between memory and network protocols. In particular, memory protocols will be packetized and routed. Note that this has already happened in shared-memory machines, such as the SGI UltraViolet [25, 26], where cache coherence protocols need to be adapted to the shared-memory architecture. More generally, we see beginnings of the convergence of coherent on-chip networks and bus standards.

Viewed from a distance, both existing networks and single-server memory buses share two fundamental abstractions: a small vocabulary of *operations* that can be performed, and a notion of *identity* that characterizes the target of the operations. In a single-server memory bus, the operations are loads and stores, and the unit of identity is physical memory words. In existing IP-based networks the operations are sending a payload (for datagram communication) or establishing a session and then reading and writing a bytestream (for stream communication), while the only identity is the coarse-grained notion of an address, which identifies a host.

In our view, the network and the memory bus should converge to a common set of operations and concept of identity. Since we provide a global address space for data, the network can expose a more bus-like interface by including loads and stores in its vocabulary. However, falling back to a `host:address` format that allows the network to route as it did before brings back the coordination spectre we wish to avoid, hence the network must participate in the understanding of references in a global address space. Thus, we plan to implement identity in the network with objects and IDs. Pointers, then, can be interpreted by the network layer as well as the OS and used for routing operations to endpoints that hold object data.

In the absence of cache coherence, memory messages are fairly straightforward. There are a handful of message types, consisting of requests and replies for read or write operations, followed by an address, and an optional payload with data, where payload size is usually a cache line [11]. Cache coherence requires additional message types, *e.g.*, to ensure exclusive access to data, upgrade access type, invalidate data, *etc.* An example of a fairly minimal, modern memory protocol with cache coherence is TileLink [1], which has publicly available implementations available for RISC-V cores.

A naïve approach to networking these messages would be to encapsulate them in an existing transport protocol (*e.g.*, TCP). A less bloated approach would be to put them into Ethernet frames and perform Layer 2 routing. However, if we need reliable delivery, then Ethernet alone is not sufficient, as it lacks a link-layer retry. Yet even Ethernet alone is likely too much overhead, as it requires a mapping from address identifier to MAC address. We argue that there will

need to be a new, light-weight form of reliable transmission, separated from the other features provided by TCP (*e.g.*, slow start). Additionally, existing transport protocols offer a fundamentally location-based abstraction. Layering address-based routing atop transport protocols requires intermediate name resolution steps and/or additional middleware, both of which increase operational cost while harming performance. Instead, our vision includes routing directly on references.

In many respects, this protocol is similar to prior work on content-based networking [2, 6]. Our approach differs from that prior work in that we are *not* proposing to redesign the Internet. Rather, we plan to leverage high-speed programmable network devices (*e.g.*, Intel's Tofino) to directly route on explicit identifiers, side-stepping the issues of custom hardware faced by other systems that routed on memory messages (*e.g.*, the Stanford FLASH Multiprocessor [21])

As an initial exploration into feasibility, we have prototyped this design using Packet Subscriptions [17], which allows for pub/sub-style communication based on user-defined packet formats. The forwarding rules generated by Packet Subscriptions are installed in a P4-defined forwarding pipeline on an Intel Tofino-based switch. With 64-bit ID fields, we could store ~1.8M exact entries and with 128-bit IDs, we could fit ~850K. To scale to larger deployments, we will explore hierarchical identifier overlay schemes.

4 INITIAL RESULTS

To investigate the feasibility of a network and OS co-design to implement a global object space, we implemented a NIC driver on Twizzler for use in emulation and used Mininet [22] to connect three Twizzler VMs to four interconnected switches that we programmed with P4 [5], where one VM drove accesses to objects and the other two responded. Our goal was to determine the overhead of the software-defined object ID routing in our design and to explore the number of round trips as objects move through the network. These experiments were also performed to explore tradeoffs in the design space of these protocols, both in the OS and in the network.

While our eventual goal is to target environments like the ones in Section 2, our initial exploration targets a more modest rack or row scale deployment of servers, to provide an environment in which we can emulate more complex ones if necessary. Thus, our initial experiments are targeted for this interconnected environment. In the future, we plan to continue our investigation more broadly, and will consider overlay networks to layer on WAN routing. We performed experiments in emulation with Mininet on Ubuntu 16.04 on an Intel Xeon Gold 5218 (emulation affected timings).

Our experiments model discovery: *i.e.*, how the network learns the location of objects. We considered two approaches: end-to-end (E2E) and controller based, which can be thought

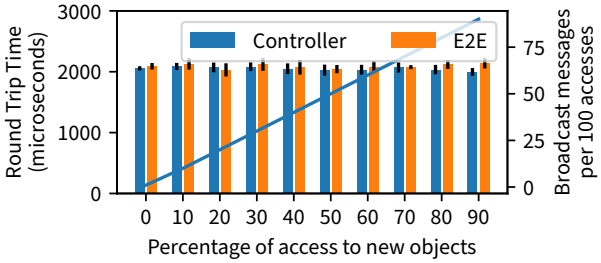

Figure 2: RTT of packets as the percent of new objects (the line) increases. Emulation impacting timings.

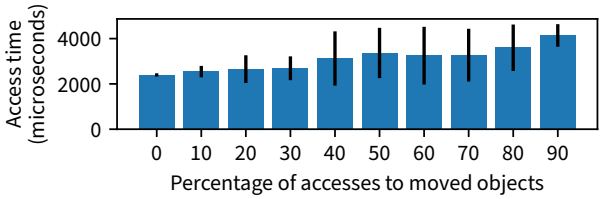

Figure 3: E2E RTT as cache gets stale due to movement.

of as a decentralized scheme analogous to ARP and a more centralized scheme using SDN controllers, respectively. In E2E, hosts store a destination cache, recording a map of object IDs and hosts that it must use broadcast to discover on first access, while in the controller scheme, hosts notify controllers about objects, which are then responsible for updating forwarding tables of switches. The E2E scheme is potentially more scalable, but has worst-case latency of 2 round-trip times (RTTs) if the cache grows stale (as this triggers a broadcast discovery packet followed by the unicast access packet), while the controller scheme has uniform latency of 1 RTT (and is unicast). As discussed above, however, memory constraints may impose limits at the switch. Thus, in our prototype, we are building both schemes so we can compare their efficacy at larger scales (and consider combinations of approaches in case of limited hardware capabilities).

Figure 2 shows RTT of both methods when accessing a mix of new and old objects. In the controller case, new objects are advertised to the switch, while in the E2E case, the host maintains tables. Our results show that switch processing overhead is minimal, even as new objects proliferate.

Figure 3 shows what happens as the destination cache in E2E grows stale. Rebroadcasts cause a significant amount of overhead, as the average number of RTTs goes up from 1 to 2. As staleness becomes overwhelming, the variability drops again since nearly all accesses require 2 round trips. Situations where the network can absorb some of the cost here, meaning that hosts do not have to deal with broadcast when staleness occurs, can reduce network traffic and latency.

5 DISCUSSION

Decoupling Through Coupling. There is an irony in our vision for decoupling applications in a more flexible manner

than RPC: this vision requires a *stronger* coupling between the OS and the network. Pushing data identity into the OS and the network as a first-class abstraction has a number of benefits, but it does imply more rigidity in the lower-level infrastructure. We argue that this is okay, and even desirable. Our goal is to provide *applications* with flexibility, scalability, and generality. By moving some application-level semantics like data references and identity into the lower-level system, we can implement in one place patterns like caching, prefetching, and query planning that often get reimplemented at many layers *because* these layers lack a common language to talk about data.

Uniformity Between Code and Data. Recently, Wang et al. [33] proposed an extension to RPC that passes first class immutable references as well as values in procedure calls and returns. The goal is to preserve the functional semantics of RPC while permitting the underlying system to avoid unnecessary copies and to perform memory management. Their design is a step in the right direction, addressing some of the weaknesses of RPC, by making it possible for the system to transparently move data. But, it only takes us halfway: RPC remains compute-centric and programmers must indicate *where* code should execute. For example, the optimization described in Section 2 in which Dave (the powerful edge device) performs inference locally could not be realized via *any* RPC mechanism. In our system, code (like data) is global and referenceable from anywhere—there would be no reason to provide a separate mechanism for specifying function invocations. Instead, we place all data *and* code in a single space, allowing code and data to reference each other. This dramatically improves expressivity, decoupling, and reuse, as we can now rely on the system to move not only data but also code to where it needs to be on demand without manual intervention and setup. In our model, the programmer primarily *orchestrates a rendezvous between code and data.*

Limitations and Challenges. Many challenges lie ahead. Perhaps foremost among them is the tension between partial failure (inevitable in any distributed system), fault tolerance, and mechanisms that attempt to hide the movement of computation and data [32]. Masking failures via replication gives rise to concerns about consistency; mechanisms that ensure consistency in the presence of possible conflicts are costly in general. We plan to address these challenges along two separate axes. At the level of the system co-design, we will experiment with offloading some synchronization and arbitration [16, 18] concerns to the programmable network (which now functions somewhat as a memory bus), letting us explore the consistency and coherence space together. At the level of programming model, we will explore how a whole-system view of object identity and references can interface with languages to support patterns for weakly consistent replication, such as auto-merging progressive objects like CRDTs [27] during data movement.

Although we free developers from explicitly moving data and code, some mechanism in the system must sill do this reasoning. We plan to explore placement issues through a co-design between query planning and optimization, and network-level scheduling. The structure of the global address space in Twizzler affords the system a view into the data layout, allowing lower levels of the stack to participate in making more intelligent placement decisions.

Future Plans. We are currently implementing this vision. We are adding networking support to Twizzler, which already supports global pointers in a global address space. We plan to use programmable network switches to develop our implementation of pointer routing. This will position us to explore protocol designs and tradeoffs in using combinations of traditional communication and RDMA in a global address space. Twizzler is well-suited to this kind of responsibility offloading because it exposes a view of objects to the NIC by abstracting physical memory, which may greatly reduce the costs of managing addressing in RDMA-like applications [3].

6 CONCLUSION

We are witnessing a convergence of trends: software's demands are growing; networks are becoming faster and capable of supporting richer semantics; and OSes are providing more abstractions for operating directly on data without the overheads of POSIX-like models. These trends strongly point towards more decoupling of compute resources, code, and data where the infrastructure orchestrates the rendezvous of distributed data and compute as requested by the programmer. Now is the time to turn our traditional host- or process-centric programming model on its head towards a more *data-centric* model supported by good pieces of DSM, content-based networks, and global address spaces. Recent advances in hardware (network speed and intelligence) and software (efficient global pointer abstractions) allow us to reconsider these previously discarded ideas in a new light. We plan to move our vision forward and realize it through the combination of new OS abstractions and network-supported bus-like protocols. Using Twizzler as a base, we will build a system to demonstrate the benefits our vision can provide.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (grants IIP-1266400, IIP-1841545), DARPA[I], gifts from Cisco, eBay, and Intel, and the members of the UCSC Center for Research in Storage Systems.

REFERENCES

[1] SiFive TileLink Specification 1.7.1. https://www.sifive.com/documentation, 2018.

[2] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A survey of information-centric networking. *IEEE Communications Magazine*, 50(7):26–36, 2012.

[3] D. Bittman, P. Alvaro, D. D. E. Long, and E. L. Miller. A tale of two abstractions: The case for object space. In *Proceedings of HotStorage '19*, July 2019.

[4] D. Bittman, P. Alvaro, P. Mehra, D. D. E. Long, and E. L. Miller. Twizzler: a data-centric OS for non-volatile memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 65–80. USENIX Association, July 2020.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[6] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, page 163–174, New York, NY, USA, 2003. Association for Computing Machinery.

[7] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, Nov. 1994.

[8] A. Dakkak, C. Li, S. G. De Gonzalo, J. Xiong, and W.-m. Hwu. Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service. IEEE CLOUD'19.

[9] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2016.

[10] G. Heiser, K. Elphinstone, S. Russell, and J. Vochteloo. Mungi: a distributed single address-space operating system. Technical Report 9314, School of Computer Science and Engineering, University of New South Wales, Nov. 1993.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012.

[12] IBM MQ. https://www-03.ibm.com/software/products/en/ibm-mq, 2019.

[13] Intel NVMe with 3D XPoint Technology chart. https://www.tomshardware.com/reviews/intel-micron-3d-xpoint-updates,4286.html#p1, 2015.

[14] Intel Skylake. https://www.7-cpu.com/cpu/Skylake.html, 2019.

[15] Intel Xeon Processor E7-8893 v3. https://ark.intel.com/content/www/us/en/ark/products/84688/intel-xeon-processor-e7-8893-v3-45m-cache-3-20-ghz.html, 2019.

[16] T. Jepsen, L. P. de Sousa, M. Moshref, F. Pedone, and R. Soulé. Infinite Resources for Optimistic Concurrency Control. In *NetCompute*, Aug. 2018.

[17] T. Jepsen, M. Moshref, A. Carzaniga, N. Foster, and R. Soulé. Forwarding and routing with packet subscriptions. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2020.

[18] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, Apr. 2018.

[19] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konečný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao. Advances and open problems in federated learning. *CoRR*, abs/1912.04977, 2019.

[20] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of The 6th International Workshop on Networking Meets Databases (NetDB'11)*, June 2011.

[21] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, and et al. The stanford flash multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 302–313, 1994.

[22] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, New York, NY, USA, 2010. Association for Computing Machinery.

[23] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[24] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[25] Sgiuv3000,uv30. https://www.risc.jku.at/projects/mach2/4555.pdf, 2016.

[26] Sgiuv3000 sets new throughput records. https://www.hpcwire.com/2016/03/25/sgi-posts-new-spec-cpu2006-results/, 2016.

[27] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[28] N. Shirokov and R. Dasineni. Open-sourcing Katran, a scalable network load balancer — Facebook Engineering. https://code.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/, May 2018.

[29] M. Shoeybi. Training Multi-billion parameter models in Megatron. https://hotchips.org/archives/hc32/, 2020.

[30] A. Singh, P. Vepakomma, O. Gupta, and R. Raskar. Detailed comparison of communication efficiency of split learning and federated learning. *arXiv preprint arXiv:1909.09145*, 2019.

[31] Tibco rendezvous. https://www.tibco.com/products/tibco-rendezvous, 2019.

[32] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. *Sun Microsystems Laboratories*, 1994.

[33] S. Wang, B. Hindman, and I. Stoica. In Reference to RPC: It's Time to Add Distributed Memory. HotOS '21.

[1]Research reported in this paper was performed in connection with Defense Advanced Research Projects Agency (DARPA) contract number W912CG-21-P-0012. The views and conclusions in this paper are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of DARPA or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.