

DAL: A Locality-Optimizing Distributed Shared Memory System

Gábor Németh Dániel Géhberger Péter Mátray
Ericsson Research

Abstract

Latency-sensitive applications like virtualized telecom and industrial IoT systems require a service for ultra-fast state externalization to become cloud-native. In this paper we propose a distributed shared memory system, called DAL, which achieves the lowest possible latency by transparently co-locating individual data items with applications working on them. Upon changes in data access patterns, the system automatically adapts data locations to keep the number of remote operations at a minimum. By avoiding the costs of network transport and using shared memory communication, the system can achieve 1 μ s data access latency. We envision DAL as a platform component which enables latency-sensitive applications to take advantage of the cloud.

1 Introduction

A key requirement for a wide range of cloud compute domains is to process a continuous influx of input data with ultra low latency. Virtualized telecom systems need to handle millions of signaling events every second while providing fluent user experience, even for delay-sensitive applications. The emerging field of Industrial IoT and cloud controlled collaborative systems pose even tougher challenges with respect to latency [3]. For instance, imagine a cloud controlled factory floor where the various fixed and mobile robots are without a precisely prescribed plan for activity and movement. Instead, they collaborate dynamically as dictated from their cloud-based control logic to fulfill a complex production task [4]. In order to appropriately drive these robots the cloud has to process the various sensor readings and video streams, and provide control feedback within strict delay bounds, e.g., within 10 ms [8].

Typical cloud-ready solutions involve a backing data store where application states are externalized. This stateless worker programming model improves the re-

siliency and scalability of virtual functions. However, in case of low latency & high reliability systems—such as the ones mentioned above—it is challenging to provide a data backend that is fast enough for state accesses to not severely degrade the end-to-end performance.

As distributed main-memory data stores have gained a lot of momentum in the recent years, there appeared specialized systems that are eagerly optimized towards latency performance, such as RAMCloud [10] and Pilaf [9]. These solutions concentrate on eliminating the various sources of delay in a data access transaction, such as buffering in protocols, context switches and blocking induced by thread and process synchronization. Their efforts were so successful that essentially LAN transport costs became the dominant factor in overall response times. To illustrate this, we tested RAMCloud in our lab equipped with commodity 10 GbE networking hardware, and found that requests spend 90% (19 μ s) of their time on transport, leaving only 10% (2.15 μ s) to end-host processing.¹

In this paper we propose DAL, a distributed shared memory system which goes beyond the known latency optimization techniques, and tries to eliminate transport costs by taking data sharding to the extreme. Instead of applying traditional hash- or range-based schemes to map keys to data servers, we handle the location of each data element separately. Using this single-key sharding approach it becomes possible to migrate data elements between server instances one-by-one. We facilitate local data access by co-locating data with client applications whenever possible.

By exploiting local data storage and shared memory communication, we achieve the lowest possible data access latency for applications where incoming events can be routed consistently to stateless worker instances, e.g., on a per subscriber basis in telecom core networks, or per-device in collaborative robot control systems. For

¹Even on high-end InfiniBand hardware, 66% of a RAMCloud request's time is spent on network transport [10].

such stateless applications DAL can move externalized states to the worker process’ physical location. Upon any change in event routing, due to e.g., a session mobility event or the failure or scaling of worker instances, relevant states are seamlessly moved to the node hosting the new worker, effectively minimizing data access costs.

In the rest of the paper we first describe the concepts behind DAL, then evaluate our prototype implementation through a simplified use-case, and finally discuss fault tolerance, applicability and security aspects.

2 Concept and design

DAL is a purely RAM-based solution consisting of two main components: a server process and a client library. To enable local data access, each physical server that runs client processes should also host a server instance. The ensemble of these server processes forms a DAL cluster. Every server instance maintains an indexed memory area, the *pool* for storing data items read and written by clients. The union of all pools comprises the distributed memory of the service.

Separation of keys and values. One of the key design goals of DAL is to allow individual data elements to be moved physically close to the applications accessing them—preferably to the same machine. To achieve this, we apply a layered approach for addressing data. In the lower (internal) layer data elements are addressed *directly* by the target server’s IP address and a pool index. The upper layer employs *key-based* addressing, essentially providing a key-value interface to the clients.

To establish the binding between the two layers, we differentiate two server roles: the *key-server* is responsible for managing key operations, such as creating or deleting keys, and resolving key to direct location queries. *Data servers*, on the other hand, process data manipulation commands with direct addresses. A single DAL server instance can and typically does combine both roles to simplify the deployment.

To access a certain key, we apply a two-phase lookup mechanism. To route key operations, DAL uses a traditional hash-based sharding scheme: the client takes the hash of the key modulo the number of key servers, and then queries the responsible server for the data location. Hereafter, the client can issue direct data manipulation commands to the designated data server. To avoid the overhead of unnecessary key operations, the client can store the direct address for the key in a handle, e.g., in a variable or a cache.

Data move. For each data item, the data server keeps track of the number and origin of accesses. When the server detects that an item is mostly accessed from a specific remote location, it proposes a *data move*. Then, as

it is depicted in Fig. 1, the client initiates a data move transaction at the responsible key server. Next, the data is copied over to the new data server, the key-to-location mapping is updated, and finally the original value is removed. If the key server is co-located with the new data server, the complete transaction between 1–8 takes two remote operations, and four otherwise.

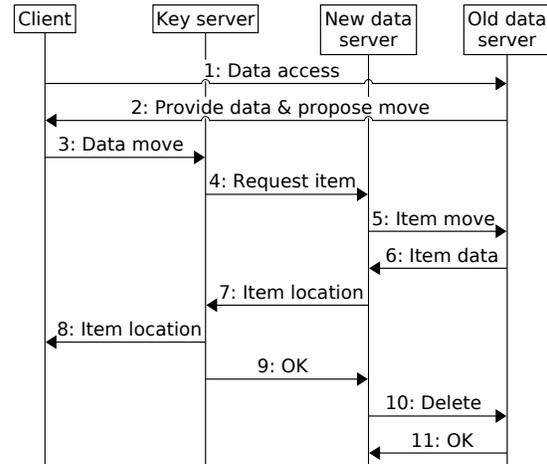


Figure 1: The steps of the data move transaction.

As a result, data items that are dominantly used by a single client will be accessed locally after a short transient time. When a client attempts to access an item that has been moved, the operation fails, and the key server is re-queried for the new location.

Access patterns. Besides locality optimization, another design goal of DAL is to support various access patterns according to which cloud applications can share data. Beyond the store-retrieve functionality, DAL servers can act as message brokers: clients can subscribe to keys and whenever a write happens to its data, the server pushes the new value to the subscribers. In essence, applications can turn traditional keys into publish-subscribe channels this way. A convenient aspect of this solution is that producers can use the same API to write data to a key, regardless of how consumers are reading it, i.e., whether they wish to receive every update or just read the value occasionally.

To facilitate the dynamic scaling of applications, consumers of a channel may be *grouped*. Within each group, messages are load-balanced among all receivers in a round-robin fashion or consistently, based on the highest random weight algorithm [13].

Moreover, DAL supports request-response communication between clients. This API is also built on top of the key-value abstraction, and as a result, the same load balancing mechanisms can be used for request routing as for messaging.

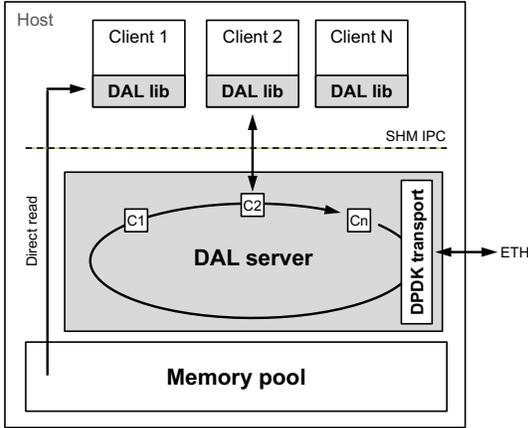


Figure 2: DAL architecture.

3 Prototype

To prove the viability of our concept and measure its performance we implemented the DAL server and client library in around 6000 lines of C++ code running on GNU/Linux. This section highlights critical decisions of the implementation we took to ensure lowest possible data access latency.

Fig. 2 shows the key elements of the architecture. The server-allocated memory pool stores both the key-to-location mapping and the value items, assuming that the key and data server roles are combined. Data items can have arbitrary sizes up to free pool capacity, and carry metadata fields, such as the auth field and version number. The auth field is a unique ID for the key-value-location triplet that guards against stale accesses via already invalidated data handles. Version numbers are atomically incremented on write operations to facilitate change detection by local clients.

Local operations. A client process interacts with its local DAL server via shared memory IPC, as shown in Fig. 2. We make local data reads as fast as possible by memory-mapping the storage pool of the server read-only in all client processes. Data items are then read from the pool directly without server involvement. The latency of this operation is only subject to the performance characteristics of underlying paging and scheduling mechanisms. To ensure that the returned data indeed corresponds to the intended key and that it was kept intact during the read operation, we employ a form of optimistic concurrency control. The client library compares the version number and auth fields right before and after the read, and re-reads the data if either of these fields have changed during the operation.

For local non-reads and all remote operations the server allocates a separate *control block* for each local

client, in the form of another shared memory area denoted C_k in Fig. 2. This area is writable both by the client and the server, and contains data structures for controlling concurrent access to its fields. Clients request specific data operations through their control blocks, which the server checks periodically in a busy loop and services requests immediately.

To eliminate extra latency costs from context switches and inter-thread synchronization, all client requests are handled on a single thread of execution. As a consequence, this design enforces serialization of data operations, ensuring consistent access to data elements. As the latency overhead of inter-process signaling is high ($\approx 7\mu\text{s}$), for local operations the client library polls the control structure until completion by default.

Remote operations. For remote data manipulation the local DAL server serializes and proxies the requests, thus hiding the details of network transport from its clients. One of those details is the use of DPDK to bypass the kernel network stack, providing low latency network access [5].

Clients can configure an *operation timeout*, the time they are willing to wait for the response. To balance between latency and CPU usage, clients can also set the wait strategy, telling the library whether it should use polling or the signaling functionality of the OS to get notified when the reply arrives.

To avoid uncontrollable buffer growth, the transport layer uses connectionless UDP transmissions. Reliability is provided by automatic retransmissions based on a *transport timeout*. Contrary to the operation timeout, the transport timeout is not set by the application but inferred from network delay statistics collected by the DAL server, and adjusted on a per request basis.

In the absence of a local server, the client library can still transmit via the standard kernel socket API—with considerably higher latencies.

The data move algorithm. In order to optimize the locality of individual data items, servers continuously analyze recent access histories on a per data item basis, as described in Sec. 2. To decide *when* an item should be moved, the algorithm checks if the majority of the last N accesses came from a single remote host (N is configurable at item creation). At negligible extra compute cost, this simple algorithm achieves significant gain for applications where data elements have a single dominant process accessing them, such as the state externalization model outlined in Sec. 1. A more sophisticated method would require resiliency against ping-pong effects and locality optimization for applications with heavy state sharing among processes, which are topics we plan to investigate in the future. The latter would require the combination of our data movement capabilities with an advanced orchestration system, so that both processes and

| ELEMENTARY DATA OPERATIONS | 50% | 99% |
|-----------------------------------|------|------|
| Local read | 0.7 | 1.1 |
| Local write | 1.1 | 1.7 |
| Remote read or write | 21.9 | 22.8 |
| COMPOUND REMOTE READ/WRITE | | |
| With local key lookup (A) | 22.6 | 26.6 |
| With remote key lookup (B) | 42.4 | 46.6 |
| With move & local key server (C) | 47.2 | 52.8 |
| With move & remote key server (D) | 87.4 | 94.6 |

Table 1: Access times for selected operations in μs . Samples for compound operations A–D are shown in Fig. 3.

data can be moved to their optimal location based on observed access patterns.

4 Evaluation

In this section, we evaluate the performance of our prototype via a stateless worker application. For the tests we use three physical machines equipped with Intel Xeon E5-2670 v3 CPUs, 64 GB RAM and Intel X540-AT2 10 GbE NICs connected through a commodity 10 Gb switch. We deploy both the DAL servers and the application components as Docker containers. Each machine hosts a single DAL server container sharing its IPC namespace with the application containers to enable shared memory communication.

The upper part of Table 1 lists the performance figures for the elementary data operations. To obtain these results, we measured 1 million randomized reads and writes with 100 byte values. As shown in the table, we can achieve sub-microsecond access times with local reads, while a remote data operation costs $22\mu\text{s}$, of which $19\mu\text{s}$ is spent on network transport.

A single server instance was measured to serve 1.6 M local writes, or 0.9 M remote read or write operations per second. As noted earlier, local reads are executed by the client processes without server involvement, so the total data access rate can exceed these numbers.

A stateless application. In order to evaluate our system in a dynamic setting, we simplified an existing DAL use case to emulate a stateless application with geographic partitioning. The latter is an inherent property of spatially distributed applications, such as session controllers in mobile networks, for which DAL can provide seamless handover of session states during mobility events. In an edge cloud setting this means that relevant states are spatially following the mobile entity and context transfer is provided as a reusable service.

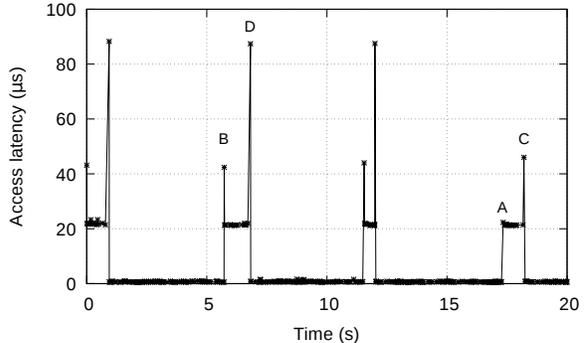


Figure 3: Data access times for a randomly selected state, with handovers between workers. Transients indicate remote access regimes, repeatedly optimized by DAL.

The test application receives input events via DAL messaging, with each event carrying a session identifier. Sessions have their corresponding states stored in DAL, and whenever the application receives a new event it reads the matching state from DAL, updates it, and writes it back.

The application is scaled out to run in multiple instances, and we use a two-tier mechanism to route events to instances. First, to imitate the spatial partitioning present in many mobile systems, we organize instances into groups, each group being responsible for a certain area. We route events to groups based on an emulated location for the session. Second, we use the session ID to route messages to instances within a group in a sticky fashion, cf. the paragraph on access patterns in Sec. 2.

DAL seamlessly adapts the location of each externalized session state, so that data gets co-located with the corresponding worker instance. Changes in input event routing may happen due to various reasons, of which we exemplify two cases in our experiments. The first is the *scaling of applications* upon changing load. During a scaling event, a large set of sessions is redirected instantaneously to a different worker instance. The second case is *session mobility*, i.e., when the source of input events moves from one pre-defined area to another. This induces individual state transfers between application groups.

Tracing a single session. In this experiment, we run the test application and emulate frequent mobility events for each session. To examine the impact on performance, in Fig. 3 we plot the data access times for a single randomly chosen session state. The time-series indicates that initially the data item is remote for the worker process. After a few accesses DAL migrates the item at $t = 1\text{ s}$,

and subsequent accesses are executed much faster.² As routing changes kick in every few seconds, data access times jump to the remote regime again, and then get re-optimized by DAL.

The transients can be further broken down into three stages. Right after a route change, the first data access by the new worker requires an extra key-to-location lookup. Depending on the location of the responsible key server, this step may (*B*) or may not (*A*) trigger an extra network round-trip. Then, we have a remote access regime, which is settled by a data move transaction (detailed in Fig. 1). This last step involves either two (*C*) or four (*D*) network round-trips—again, depending on the location of the key server—and appears as a spike reaching up to 47 μ s or 87 μ s, respectively. The lower part of Table 1 lists statistics for the *A–D* cases. It can be seen that all categories exhibit low variability.

Mass evaluation. In order to demonstrate the behavior of a DAL cluster under heavy load, we repeat the previous experiment, and execute scaling events to trigger mass state migrations.

Fig. 4 shows how the share of various data operations from Table 1 evolves during the experiment. Initially, we experience a high ratio of key lookups, and as the states are distributed randomly between the three DAL servers, most operations are remote. In the range $2\text{ s} \leq t \leq 10\text{ s}$, states are automatically moved to their optimal location, and the system reaches a steady state. We can observe that the share of local accesses does not reach 1. This is because we continuously generate session mobility events, causing a small fraction of session states to always be dislocated.

The right side of the figure shows the effect of adding an extra worker to one of the application groups at $t = 39\text{ s}$. As events begin to be routed to the new instance, it starts to access states kept in remote DAL servers. As a result, there appears a sudden increase in the number of key lookups, and a drop in local data operations. This is relaxed during the period $40\text{ s} \leq t \leq 50\text{ s}$, after which the system returns to the optimized steady state.

5 Related work

Data partitioning and placement methods in key-value stores are typically strict, disallowing fine-grained data relocation. Furthermore, in most cases, the storage server cluster is physically separated from clients which makes local data access impossible. In recent years various competing solutions have been developed to address ultra-low latency data access [10, 2, 6, 9, 14, 7]. All

²The test system would work even better if we auto-moved session states on the very first remote access. For the sake of presentation we configured the window size here to $N = 8$.

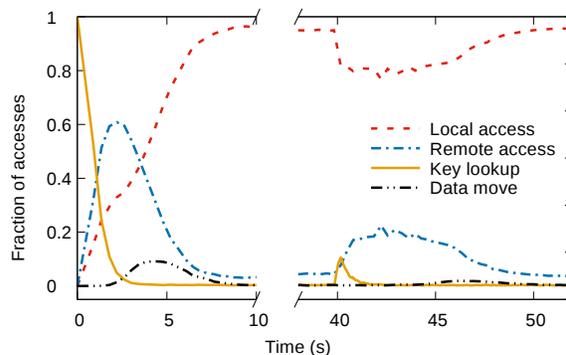


Figure 4: The mixture of elementary and compound operations during the mass experiment. Key lookups and data moves also include a remote data access.

of these systems use hash-based partitioning and separate servers and clients. FaRM [2] accepts location hints when creating objects making co-location and single machine transactions possible. While RAMCloud [10] achieves impressive results with 5 μ s remote read operations over InfiniBand, without the possibility of local data access and flexible data movement, access times will still be dominated by transport delays.

The cluster mode of Redis [11] supports local data access and it is also possible to enforce the static co-location of objects. However, even local communication goes through the kernel network stack, thus its latency is in the range of tens of microseconds.

Besides the above mentioned solutions, there are other non latency-oriented data systems which can apply finer control over the placement of data elements. Agarwal et al. proposed a spring force model to optimize web session data placement between datacenters. Their solution is based on the offline analysis of access logs [1]. Swift [12] studies data placement methods for optimizing execution times of a transaction manager on top of HBase. The method is based on clustering related items of the data store to limit the number of nodes involved in each transaction.

6 Conclusions

In this paper we proposed DAL, a distributed shared memory system that achieves 1 μ s data access by optimizing the physical location of each individual data element. Besides ultra-low latency, DAL provides seamless elasticity for scaling and session mobility events, making state externalization of latency sensitive applications possible.

Discussion

Although the paper does not address it, replication is important for many applications to ensure resilience against node failures. Asynchronous replication can be applied to our design without major impact on read/write latencies. Due to the weaker consistency guarantees of asynchronous replication, in corner cases node failures can lead to lost updates. However, some application types can simply recover from such events. For example, many telecom VNFs can either rebuild session states from other (significantly slower) data sources, or can simply drop the affected sessions, and wait for a reconnection. Similarly, certain robot control applications are designed to continuously adapt their states through tight feedback loops, thus a single lost state update does not break the control logic. Consequently, our design can provide low latency with fault tolerance, at the price of possibly dropping a small fraction of state updates in the event of a failure. We believe this is a viable trade-off for many soft real-time cloud applications.

The two-phase lookup mechanism produces single-key shards, making the relocation of individual data elements possible. This comes with the cost of an extra location lookup (in most cases via the network), making the first data access slower than subsequent ones. As a result, our concept best suits applications where it is possible to reuse the results of location lookups. In cases when data handles cannot be effectively cached (e.g., due to the non-partitionable nature and excessive size of the key-space), the performance will lag behind that of existing systems.

Due to the key-location separation, data servers can be added to the system at any time—local clients, if any, will trigger data item relocations through the optimization normally. On the other hand, the scaling of key servers needs further work, most notably the handling of large batches of item location relocations during resharding of key ranges.

Finally, the prototype does not consider security and privacy aspects. If an application knows a key, it can perform operations on the corresponding value. This issue can be addressed with a more fine-grained access control, for instance by providing a separate memory pool for each tenant.

References

- [1] AGARWAL, S., DUNAGAN, J., JAIN, N., SAROIU, S., WOLMAN, A., AND BHOGAN, H. Volley: Automated Data Placement for Geo-distributed Cloud Services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 2–2.
- [2] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 401–414.
- [3] HOLFELD, B., WIERUCH, D., WIRTH, T., THIELE, L., ASHRAF, S. A., HUSCHKE, J., AKTAS, I., AND ANSARI, J. Wireless Communication for Factory Automation: an opportunity for LTE and 5G systems. *IEEE Communications Magazine* 54, 6 (2016), 36–43.
- [4] HU, G., TAY, W. P., AND WEN, Y. Cloud robotics: architecture, challenges and applications. *IEEE network* 26, 3 (2012).
- [5] INTEL. Data Plane Development Kit. <http://dpdk.org>.
- [6] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 295–306.
- [7] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 429–444.
- [8] MAHMUD, N., SANDSTRÖM, K., AND VULGARAKIS, A. Evaluating industrial applicability of virtualization on a distributed multicore platform. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE* (2014), IEEE, pp. 1–8.
- [9] MITCHELL, C., GENG, Y., AND LI, J. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 103–114.
- [10] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3 (Aug. 2015), 7:1–7:55.
- [11] SANFILIPPO, S., AND NOORDHUIS, P. Redis. <https://redis.io/topics/cluster-tutorial>, 2009.
- [12] SWIFT, B. P. Data Placement in a Scalable Transactional Data Store. http://www.globule.org/publi/DPSTDS_master2012.pdf, 2012. Master's thesis.
- [13] THALER, D. G., AND RAVISHANKAR, C. V. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking (TON)* 6, 1 (1998), 1–14.
- [14] WANG, Y., ZHANG, L., TAN, J., LI, M., GAO, Y., GUERIN, X., MENG, X., AND MENG, S. HydraDB: A Resilient RDMA-driven Key-value Middleware for In-memory Cluster Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 22:1–22:11.