MAKING LOW LATENCY STORES PRACTICAL AT CLOUD SCALE

by

Chinmay Satish Kulkarni

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Chinmay Satish Kulkarni**

has been approved by the following supervisory committee members:

Ryan Stutsman, Chair(s) __
Date Approved

Robert Ricci, Member __
Date Approved

John Regehr, Member __
Date Approved

Kobus Van Der Merwe, Member __
Date Approved

Badrish Chandramouli, Member __
Date Approved

by **Mary Hall**, Chair/Dean of

the Department/College/School of **Computing**

and by **David B Kieda**, Dean of The Graduate School.

ABSTRACT

CONTENTS

CHAPTERS

LIST OF FIGURES

ACKNOWLEDGEMENTS

CHAPTER 1

EXTENSIBILITY AND MULTI-TENANCY

Since the end of Dennard scaling, disaggregation has become the norm in the datacenter. Applications are typically broken into a compute and storage tier separated by a high speed network, allowing each tier to be provisioned, managed, and scaled independently. However, this approach is beginning to reach its limits. Applications have evolved to become more data intensive than ever. In addition to good performance, they often require rich and complex data models such as social graphs, decision trees, vectors [14, 16] etc. Storage systems, on the other hand, have become faster with the help of kernel-bypass [7, 17], but at the cost of their interface – typically simple point lookups and updates. As a result of using these simple interfaces to implement their data model, applications end up stalling on network round-trips to the storage tier. Since the actual lookup or update takes only a few microseconds at the storage server, these round-trips create a major bottleneck, hurting performance and utilization. Therefore, to fully leverage these fast storage systems, applications will have to reduce round-trips by pushing compute to them.

Pushing compute to these fast storage systems is not straightforward. To maximize utilization, these systems need to be shared by multiple tenants, but the cost for isolating tenants using conventional techniques is too high. Hardware isolation requires a context switch that takes approximately 1.5 microseconds on a modern processor [12]. This is roughly equal to the amount of time it takes to fully process an RPC at the storage server, meaning that conventional isolation can hurt throughput by a factor of 2 (Figure 1.1). Splinter relies on a type- and memory-safe language for isolation. Tenants push extensions – a tree traversal for example – written in the Rust programming language [19] to the system at runtime. Splinter installs these extensions. Once installed, an extension can be

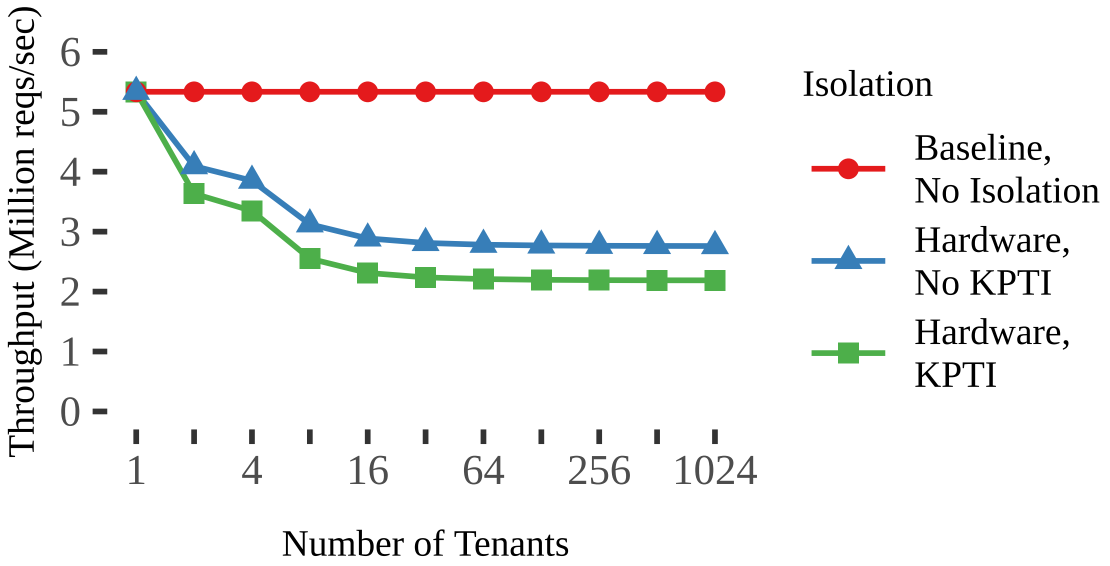

Figure 1.1: Simulated throughput of a system that isolates pushed code using hardware. The baseline represents the upper bound when there is no isolation. At high tenant density, isolation hurts throughput by nearly 2x.

remotely invoked (executed) by the tenant in a single round-trip. For applications such as tree traversals which would ordinarily require round-trips logarithmic in the size of the tree, splinter can significantly improve both throughput and latency.

In addition to lightweight isolation, splinter consists of multiple mechanisms to make pushing compute feasible. Cross-core synchronization is minimized by maintaining *tenant locality*; tenant requests are routed to preferred cores at the NIC [8] itself, and cores steal work from their neighbour to combat any resulting load imbalances. Pushed code (an extension) is scheduled *cooperatively*; extensions are expected to yield down to the storage layer frequently ensuring that long running extensions do not starve out short running ones. This approach is preferred over conventional multitasking using kthreads because preempting a kthread requires a context switch, making it too expensive for microsecond timescales. Uncooperative extensions are identified and dealt with by a dedicated watchdog core. Data copies are minimized by passing immutable references to extensions; the rust compiler statically verifies the lifetime and safety of these references. With the help of these mechanisms, Splinter can isolate 100's of granular tenant extensions per core while serving millions of operations per second with microsecond latencies.

Overall, Splinter adds extensibility to fast kernel-bypass storage systems, making it easier for applications to use them. An 800 line Splinter extension implementing

Facebook's TAO graph model [2] can serve 2.8 million ops/s on 8 threads with an average latency of 30 microseconds. A significant fraction of TAO operations involve only a single round-trip. Implementing these on the client using normal lookups and implementing the remaining operations using the extension helps improve performance to 3.2 million ops/s at the same latency. This means that an approach that combines normal lookups/updates with Splinter's extensions is the best for performance; the normal lookups do not incur isolation overhead (no matter how low), and the extensions reduce the number of round-trips. In comparison, FaRM's [6] implementation of TAO performs 6.3 million ops/s on 32 threads with an average latency of 41 microseconds. This makes Splinter's approach, which performs 0.4 million ops/s per thread, competitive with FaRM's RDMA based approach, which performs 0.2 million ops/s per thread.

CHAPTER 2

FAST DATA MIGRATION

In order to meet their latency and throughput goals, kernel-bypass storage systems often start out with simple, stripped down designs with the focus on fast, efficient request processing [15]. When it comes to distribution, hash partitioning is often the norm since it is a simple, efficient, and scalable way of distributing load across a cluster of machines. Most systems tend to pre-partition records and tables into coarse hash buckets, and then move these buckets around the cluster in response to load imbalances [5]. However, coarse pre-partitioning can lead to high request fan-out when applications exhibit temporal locality in the records they access, hurting performance (Figure 2.1) and cluster utilization [11]. Therefore, in order to be able to support a diverse set of applications with different access patterns, these systems need to be more flexible and lazy about how they partition and distribute data.

Flexible and lazy partitioning creates a unique challenge for kernel-bypass storage systems. Once a decision to partition is made, the partition must be quickly moved to it's new home with minimum impact to performance. Doing so is hard; these systems offer latencies as low as 5 microseconds, so even a few cache misses will significantly hurt performance. Rocksteady [11] is a fast, low-impact data migration protocol that tackles this problem. Built on top of RAMCloud [17], Rocksteady's key insight is to leverage application skew to speed up data migration while minimizing the impact to performance. When migrating a partition from a source to a target, it first migrates ownership of the partition. Doing so moves load on the partition from the source to the target, creating headroom on the source that can be used to migrate data. To keep the partition online, the target pulls records from the source on-demand; since applications are skewed – most requests are for a small set of hot records – this on-demand process converges quickly.

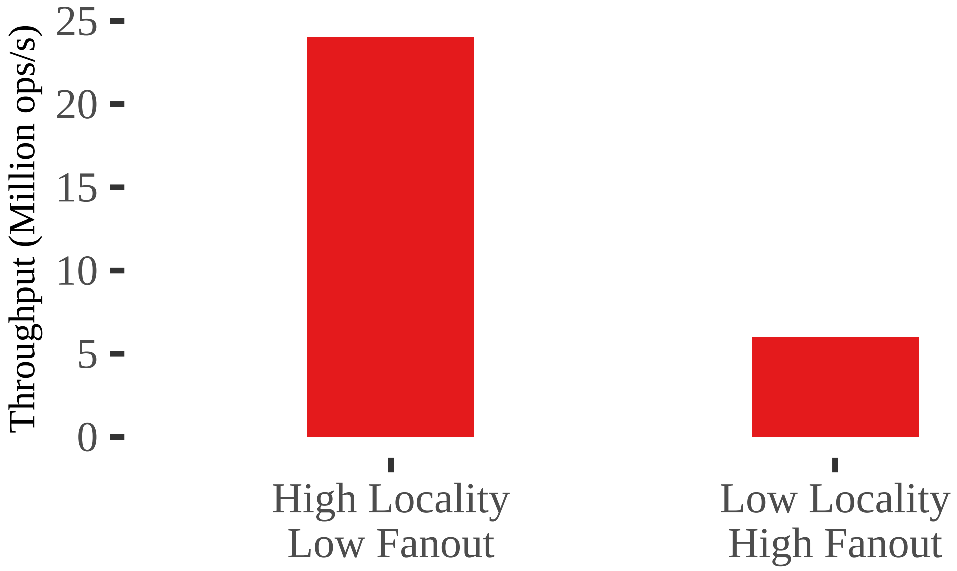

Figure 2.1: The impact of locality on cluster throughput. When locality is low due to high fanout, a cluster of RAMCloud servers performs 6 Million ops/s. On improving locality, by potentially migrating data to reduce fanout, throughput improves to 24 Million ops/s

To fully utilize created headroom, Rocksteady carefully schedules and pipelines data migration on both the source and target. Migration is broken up into tasks that work in parallel over RAMCloud's hash table; doing so keeps the pre-fetcher happy, improving cache locality. A shared-memory model allows these tasks to be scheduled on any core, allowing any idle compute on the source and target to be used for migration. To further speed up migration, Rocksteady delays re-replication of migrated data at the target to until after migration has completed. Fault tolerance is guaranteed by maintaining a dependency between the source and target at RAMCloud's coordinator (called lineage) during the migration, and recovering all data at the source if either machine crashes. Recovery must also include the target because of early ownership transfer; the target could have served and replicated writes on the partition since the migration began. Putting all these parts together results in a protocol that migrates data 100x faster than the state-of-the-art while maintaining tail latencies 1000x lower.

Overall, Rocksteady's careful attention to ownership, scheduling, and fault tolerance allow it to quickly and safely migrate data with low impact to performance. Experiments show that it can migrate at 758 MBps while maintaining tail latency below 250 microseconds; this is equivalent to migrating 256 GB of data in 6 minutes, allowing for quick scale-up and scale-down of a cluster. Experiments also show the benefits of leveraging skew: at low skew, Rocksteady keeps median latency below 75 microseconds, at high skew, it keeps median latency below 28 microseconds, and at even higher skew, Rocksteady keeps median latency below 17 microseconds. Additionally, early ownership transfer and lineage help improve migration speed by 25%. These results have important implications on system design; fast storage systems can use Rocksteady as a mechanism to enable flexible, lazy partitioning of data.

CHAPTER 3

LOW COST COORDINATION

Millions of sensors, mobile applications, users, and machines now continuously generate billions of events. These events are processed by streaming engines [?, ?] and ingested and aggregated by state management systems (Figure 3.1). Real-time queries are issued against this ingested data to train and update models for prediction, to analyze user behavior, or to generate device crash reports, etc. Hence, these state management systems are a focal point for massive numbers of events and queries over aggregated information about them.

Recently, this has led to specialized key-value stores (KVSs) that can ingest and index these events at high rates – 100 million operations (Mops) per second (s) per machine – by exploiting many-core hardware [3, 21]. These systems are efficient if events are generated on the same machine as the KVS, but, in practice, events need to be aggregated from a wide and distributed set of data sources. Hence, fast indexing schemes alone only solve part of the problem. To be practical and cost-effective, a complete system for aggregating these events must ingest events over the network, must scale across machines as well as cores, and must be elastic (by provisioning and reconfiguring over inexpensive cloud resources as workloads change).

The only existing KVSs that provide similar performance [10, 13, 15, 18] rely on application-specific hardware acceleration, making them impossible to deploy on today's cloud platforms. Furthermore, these systems only store data in DRAM, and they do not scale across machines; adding support to do so without cutting into normal-case performance is not straightforward. For example, many of them statically partition records across cores to eliminate cross-core synchronization. This optimizes normal-case performance, but it makes concurrent operations like migration and scale out impossible;

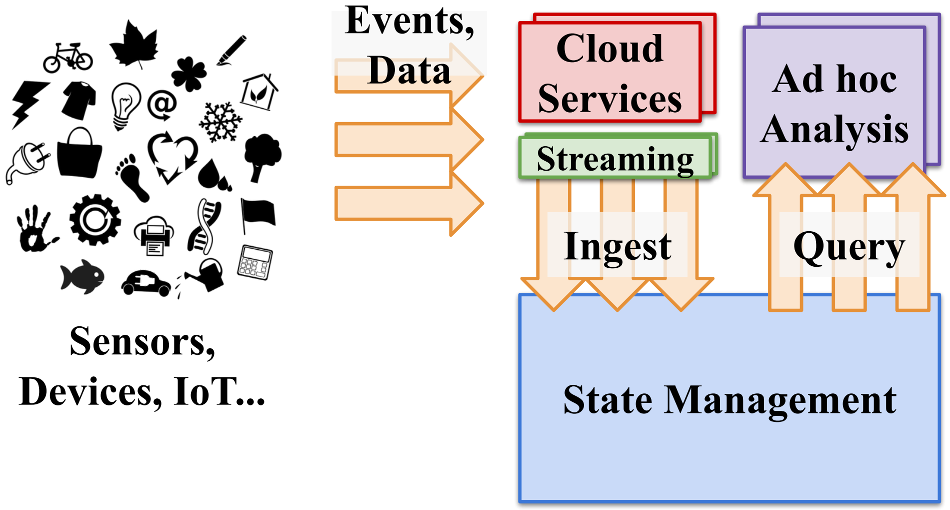

Figure 3.1: A typical data processing pipeline. Services receive and process raw events and data. A state management system ingests processed events and data, and serves offline queries against them.

transferring record data and ownership between machines and cores requires a stop-the-world approach due to these systems' lack of fine-grained synchronization.

Achieving this level of performance while fulfilling all of these requirements on commodity cloud platforms requires solving two key challenges simultaneously. First, workloads change over time and cloud VMs fail, so systems must tolerate failure and reconfiguration. Doing this without hurting normal-case performance at 100 Mops/s is hard, since even a single extra server-side cache miss to check key ownership or reconfiguration status would cut throughput by tens-of-millions of operations per second. Second, the high CPU cost of processing incoming network packets easily dominates in these workloads, especially since, historically, cloud networking stacks have not been designed for high data rates and high efficiency. However, this is changing; by careful design of each server's data path, cloud applications can exploit transparent hardware acceleration and offloading offered by cloud providers to process more than 100 Mops/s per cloud virtual machine (VM).

Shadowfax is a new distributed KVS that transparently spans DRAM, SSDs, and cloud blob storage while serving 130 Mops/s/VM over commodity Azure VMs [4] using conventional Linux TCP. Beyond high single-VM performance, its unique approach to distributed reconfiguration avoids any server-side key ownership checks and any cross-core coordination during normal operation and data migration both in its indexing and network interactions. Hence, it can shift load in 17 s to improve cluster throughput by 10 Mops/s with little disruption. Compared to the state-of-the-art, it has 8× better throughput (than Seastar+memcached [1]) and scales out 6× faster (than Rocksteady [11]).

Three key pieces of Shadowfax help eliminate coordination throughout the client- and server-side by eliminating cross-request and cross-core coordination:

Low-cost Coordination via Global Cuts: In contrast to totally-ordered or stop-the-world approaches used by most systems, cores in Shadowfax avoid stalling to synchronize with one another, even when triggering complex operations like scale-out, which require defining clear before/after points in time among concurrent operations. Instead, each core participating in these operations – both at clients and servers – independently decides a point in an *asynchronous global cut* that defines a boundary between operation sequences in these complex operations. Shadowfax extends asynchronous cuts from cores within one process [3] to servers and clients in a cluster. This helps coordinate server and client threads (through partitioned sessions) in Shadowfax's low-coordination data migration and reconfiguration protocol.

End-to-end Asynchronous Clients: All requests from a client on one machine to Shadowfax are asynchronous with respect to one another all the way throughout Shadowfax's client- and server-side network submission/completion paths and servers' indexing and (SSD and cloud storage) I/O paths. This avoids all client- and server-side stalls due to head-of-line blocking, ensuring that clients can always continue to generate requests and servers can always continue to process them. In turn, clients naturally batch requests, improving server-side high throughput especially under high load. This batching also suits hardware accelerated network offloads available in cloud platforms today further lowering CPU load and improving throughput. Hence, despite batching, requests complete in less than 40 µs to 1.3 ms at more than 120 Mops/s/VM, depending on which transport and hardware acceleration is chosen.

Partitioned Sessions, Shared Data: Asynchronous requests eliminate blocking *between*

> *requests* within a client, but maintaining high throughput also requires minimizing coordination costs *between cores* at clients and servers. Instead of partitioning data among cores to avoid synchronization on record accesses [1, 9, 15, 20], Shadowfax partitions network sessions across cores; its lock-free hash index and log-structured record heap are shared among all cores. This risks contention when some records are hot and frequently mutated, but this is more than offset by the fact that no software-level inter-core request forwarding or routing is needed within server VMs.

Evaluating Shadowfax in detail against other state-of-the-art shared-nothing approaches shows that by eliminating record ownership checks and cross-core communication for routing requests, it improves per-machine throughput by 8.5× on commodity cloud VMs. It also retains high throughput during migrations and scales to a cluster that ingests and indexes 400 Mops/s in total, which is the highest reported throughput for a distributed KVS till date.

REFERENCES

[1] *Seastar Framework*. http://seastar.io. Accessed: 4/22/2020.

[2] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, *TAO: Facebook's Distributed Data Store for the Social Graph*, in Proceedings of the 2013 USENIX Annual Technical Conference, USENIX ATC '13, San Jose, CA, 2013, USENIX Association, pp. 49–60.

[3] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, *Faster: A concurrent key-value store with in-place updates*, in Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, New York, NY, USA, 2018, ACM, pp. 275–290.

[4] M. Copeland, J. Soh, A. Puca, M. Manning, and D. Gollob, *Microsoft Azure: Planning, Deploying, and Managing Your Data Center in the Cloud*, Apress, USA, 1st ed., 2015.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, *Dynamo: Amazon's Highly Available Key-value Store*, in Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, New York, NY, 2007, ACM, pp. 205–220.

[6] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, *FaRM: Fast Remote Memory*, in Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI '14, Berkeley, CA, 2014, USENIX Association, pp. 401–414.

[7] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, *No compromises: Distributed transactions with consistency, availability, and performance*, in Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, New York, NY, USA, 2015, Association for Computing Machinery, p. 54–70.

[8] Intel Corporation., *Flow Director*. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf. Accessed: 2018-09-27.

[9] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, *H-store: A High-performance, Distributed Main Memory Transaction Processing System*, Proc. VLDB Endow., 1 (2008), pp. 1496–1499.

[10] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, *High performance packet processing with flexnic*, in Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, New York, NY, USA, 2016, Association for Computing Machinery, p. 67–81.

[11] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman, *Rocksteady: Fast Migration for Low-latency In-memory Storage*, in Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, New York, NY, 2017, ACM, pp. 390–405.

[12] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman, *Splinter: Bare-metal extensions for multi-tenant low-latency storage*, in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, Oct. 2018, USENIX Association, pp. 627–643.

[13] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, *Kv-direct: High-performance in-memory key-value store with programmable nic*, in Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, New York, NY, USA, 2017, ACM, pp. 137–152.

[14] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, *Scaling distributed machine learning with the parameter server*, in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, CO, Oct. 2014, USENIX Association, pp. 583–598.

[15] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, *MICA: A Holistic Approach to Fast In-memory Key-value Storage*, in Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI '14, Berkeley, CA, 2014, USENIX Association, pp. 429–444.

[16] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, *Scaling memcache at facebook*, in 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), Lombard, IL, Apr. 2013, USENIX Association, pp. 385–398.

[17] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, and et al., *The ramcloud storage system*, ACM Trans. Comput. Syst., 33 (2015).

[18] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, *Floem: A programming system for nic-accelerated network applications*, in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, USA, 2018, USENIX Association, p. 663–679.

[19] *The Rust Programming Language*. `http://www.rust-lang.org/en-US/`. Accessed: 2018-09-27.

[20] M. Stonebraker and A. Weisberg, *The voltdb main memory DBMS*, IEEE Data Eng. Bull., 36 (2013), pp. 21–27.

[21] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, *Anna: A kvs for any scale*, in 2018 IEEE 34th International Conference on Data Engineering (ICDE), 2018, pp. 401–412.