April 2016 Issue # 1

THE MORNING PAPER
QUARTERLY REVIEW

ARIES
A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks

HYPERLOGLOG IN PRACTICE
Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm

NO COMPROMISES
Distributed Transactions with Consistency, Availability, and Performance

A selection of papers from the world of computer science, as featured by Adrian Colyer on The Morning Paper blog

InfoQ

CONTENTS

Issue # 1, April 2016

WELCOME

Adrian Colyer

The idea of The Morning Paper blog is simple: every weekday I take a computer science research paper and write it up as a post. If you prefer to have the day's paper delivered straight to your inbox, there's an option to subscribe to a mailing list as well. On the blog, you'll find a mix of past papers and current research results. I cover a fairly wide range of topics, but with a bias towards distributed systems and data. The blog grew from my habit of reading research papers during my commute time - I figured they'd give me much more lasting value than the newspapers many of my fellow commuters were reading! Reading papers has been a wonderfully beneficial activity for me (see my QCon London keynote if you're interested in learning more about this), and I very much hope that you'll get enjoyment and benefit from learning about some of the wonderful developments being made in computer science too.

Over the course of a year, subscribers to the blog will be exposed to just over 200 papers and ideas on average. Of course, not everyone will have the time to read all 200 posts! So when Charles Humble from InfoQ approached me with the idea of putting together a quarterly eMag highlighting some of my favourite papers/posts from the quarter, I jumped at the chance. The result is the eMag you're reading now. I hope the papers I've chosen inspire you to dig deeper into the wonderful world of computer science.

ABOUT ADRIAN COLYER

I'm a Venture Partner with Accel in London, and also act as advisor to a number of companies including Atomist, ClusterHQ, Skipjaq, and Weaveworks. Prior to joining Accel in mid 2014, I've always worked in the technology industry, including CTO roles with SpringSource, VMware, and Pivotal. I love meeting with founders and entrepreneurs building technology-based businesses and hearing their stories, and I've found that technical entrepreneurs often enjoy meeting with someone from the venture capital industry who can (sometimes!) understand what they're doing on a deeper level. If you're building a technology-based business (especially out of Europe) and have reached the point where you're thinking about venture capital investment, get in touch!

INTRODUCING THE PAPERS

It's a hard task whittling down 68 posts from the first quarter of this year to just 5 papers! In the end, with apologies to all the researchers whose great work I had to omit, I chose the following:

- Mohan, C. et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks. *ACM Transactions on Database Systems*, 1992

- Dragojević, A. et al. No compromises: distributed transactions with consistency, availability, and performance. *ACM Symposium on Operating Systems Principles*, 2015

- Pillai, T.S. et al. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

- Kaloper-Meršinjak, D., Mehnert, H. et al. 2015 Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation. *24th USENIX Security Symposium*, 2015

- Heule, S. et al. 2013 HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. *EDBT/ICDT*2013

A TRANSACTION RECOVERY METHOD

The 5th edition of the 'Red Book' - *Readings in Database Systems* - was recently released with Peter Bailis, Joseph Hellerstein, and Michael Stonebraker as editors. I kicked off the year on The Morning Paper by looking at the papers from chapter 3 of the Red Book, "Techniques Everyone Should Know." In his introduction to the chapter, Peter Bailis writes:

> *In this chapter, we present primary and near-primary sources for several of the most important core concept in database system design: query planning, concurrency control, database recovery, and distribution. The ideas in this chapter are so fundamental to modern database systems that nearly every mature database system implementation contains them.*

'ARIES: A Transaction Recovery Method...' is one of the those primary sources. IBM's ARIES was an incredibly influential system, and the ARIES recovery manager algorithm based on Write-Ahead Logging with 'No Force' and 'Steal' policies became the canonical approach to recovery. The paper is a great reminder of just how much engineering effort goes into ensuring a system can recover from failure with no data loss. I think on this when I meet with a company that tells me "we don't support failure recovery yet, but it's in our roadmap for the next version." Designing for failure is a fundamental part of system design, and it's not at all obvious you can always bolt it on after-the-fact.

NO COMPROMISES

I'm fascinated by the hardware changes making their way into modern data centers. There are big advances coming in storage, networking, memory, and processing. But I confess, I've never been over excited by hardware in its own right. It's the *implications* of hardware changes and what they mean for the systems we build on top that gets me excited. System design is about making trade-offs within a design space. If some of the underlying assumptions that we built systems on the basis of change, then the optimal system design to achieve some end may also change. I wrote a summary piece called "All Change Please" that looks at some of these issues if you're interested in digging deeper.

'No Compromises: Distributed Transactions with Consistency, Availability, and Performance' is one of the papers that examines these trade-offs. It looks at the implications of fast networks and persistent memory. 'No Compromises' describes a system called FaRM

that provides *distributed* transactions with strict serializability (the gold standard), high availability and throughput, and low-latency. It just might challenge some of your own assumptions about what is possible too.

ALL FILE SYSTEMS ARE NOT CREATED EQUAL

Confusing syntax for semantics is a classic trap. Just because you've specified the *interface* for something, that doesn't mean that you've necessarily specified all of the important semantics that need to be understood to use it correctly and safely. In "All file systems are not created equal," Pillai et al. demonstrate this principle in practice with respect to the POSIX file system API. The POSIX standard leaves open to interpretation how disk state is mutated in the event of a crash. Does that matter? Yes! If you're building something that relies on a file system, then this paper is essential reading. The authors show that persistence properties vary widely among file systems, and even among different configurations of the same file system. Testing 34 different configurations across 11 applications (including LevelDB, SQLite, PostgreSQL, Git, and HDFS) the tools developed by the authors found many vulnerabilities with severe consequences such as silent errors or data loss. Building a crash consistent application on top of a file system it turns out, is really rather difficult.

NOT QUITE SO BROKEN TLS

The evidence strongly suggests we have a problem with the way we build systems software when it comes to security. In 'Not-quite-so-broken TLS' we get to see not only a more robust implementation of TLS, but also the *processes and practices* the team used to create it. And that, for me, is the bigger and more valuable story. If you've ever thought to yourself 'surely there must be a better way,' then I'm sure this paper will appeal to you. There is no one silver bullet, but through a combination of techniques (and a lovely analysis that shows how those techniques protect against different classes of vulnerabilities) we get a glimpse of what it might take for us to move the state of the practice forward, and the hope that it might just be possible!

HYPERLOGLOG IN PRACTICE

In March, I covered Alfred Spector's 2012 paper on 'Google's Hybrid Approach to Research' which provides a fascinating look at the way Google balances fundamental and applied research, and how they integrate research into the product teams themselves. 'HyperLogLog in Practice' is a wonderful illustration of this approach in action. The base HyperLogLog algorithm itself - used for estimating the counts of distinct data items in large sets of data - is pretty cool to start with, and in this paper we get to see how Google engineers and researchers incrementally improve it step by step. The improvements decrease the amount of memory required, and increase the accuracy for a range of important cardinalities. This matters to Google because they end up performing millions of such computations a day through PowerDrill and other systems.

For more great advice from Google, check out 'Machine Learning: The High-Interest Credit Card of Technical Debt,' 'Ad Click Prediction: A View from the Trenches,' 'Inferring Causal Impact Using Bayesian Structural Time-Series Models,' and of course 'Maglev: A Fast and Reliable Sofware Network Load Balancer' - all of which have featured on The Morning Paper this past quarter.

ARIES

A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks

Mohan et al. 1992

This is part 5 of a 7 part series on (database) 'Techniques Everyone Should Know.'

From Peter Bailis' introduction to this paper in chapter 3 of the Redbook:

Another major problem in transaction processing is maintaining durability: the effects of transaction-processing should survive system failures. A near-ubiquitous technique for maintaining durability is to perform logging: during transaction execution, transaction executions are stored on fault-tolerant media (e.g., hard drives on SSDs) in a log. Everyone working in data systems should understand how write-ahead logging works, preferably in some detail. The canonical algorithm for implementing an "No Force, Steal" WAL-based recovery manager is IBM's ARIES algorithm... In ARIES, the database need not write dirty pages to disk at commit time ("No Force"), and the database can flush dirty pages to disk at any time ("Steal"); these policies allow high performance and are present in almost every commercial RDBMS offering but in turn add complexity to the database.

ARIES stands for "Algorithm for Recovery and Isolation Exploiting Semantics." It was designed to support the needs of *industrial strength* transaction processing systems. ARIES uses logs to record the progress of transactions and their actions which cause changes to recoverable data objects. The log is the source of truth and is used to ensure that committed actions are reflected in the database, and that uncommitned actions are undone. Conceptually the log is a single ever-growing sequential file (append-only). Every log record has a unique log sequence number (LSN), and LSNs are assigned in ascending order.

Log records are first written to volatile storage (e.g. in-memory), and at certain times – such as transaction commit – the log records up to a certain point (LSN) are written to stable stor-

"Everyone working in data systems should understand how write-ahead logging works, preferably in some detail. The canonical algorithm for implementing an "No Force, Steal" WAL-based recovery manager is IBM's ARIES algorithm"

age. This is known as *forcing* the log up to that LSN. A system may periodically force the log buffers as they fill up.

The WAL protocol asserts that the log records representing changes to some data must already be on stable storage before the changed data is allowed to replace the previous version of that data on non-volatile storage... To enable the enforcement of this protocol, systems using the WAL method of recovery store in every page the LSN of the log record that describes the most recent update performed on that page.

Log records may contain *redo* and *undo* information. A log record containing both is called an undo-redo log record, there may also be undo-only log records and redo-only log records. A redo record contains the information to redo a change made by a transaction (if they have been lost). An undo record contains the information needed to reverse a change made by a transaction (in the event of rollback). This information can simply be

a copy of the before/after image of the data object, or it can be a description of the operation that needs to be perform to undo/ redo the change.

Operation logging permits the use of high concurrency lock modes, which exploit the semantics of the operations performed on the data. For example, with certain operations, the same field of a record could have uncommitted updates of many transactions.

(An example given later in the paper is operational logging of increment and decrement operations for a balance, vs. storing the absolute before/after values of the balance).

A *compensation log record* (CLR) is used to log the action of rolling back a change (for example, to a prior *savepoint* in a transaction). In ARIES, CLRs are redo-only log records.

Transaction status is also stored in the log, and no transaction can be considered complete until its committed status and all its log data are safely recorded on stable storage by forcing the log up to the transaction's commit log record's LSN. This allows a restart recovery procedure to recover any transactions that completed successfully but whose updated pages were not physically written to nonvolatile storage before the failure of the system. A transaction is not permitted to complete its commit processing until the redo portions of all log records of that transaction have been written to stable storage.

All of this logging is in aid of recovery from failure. There are three basic types of failure we need to concern ourselves with:

1. Failure of a transaction (such that its updates need to be undone).

2. Failure of the database management system itself – in this scenario we assume that volatile storage contents are lost and recovery must be performed using the nonvolatile versions of the database and log.

3. Failure of media/device – in this scenario the contents of just that media are lost, and the lost data must be recovered using an image copy (archive dump) version of the lost data plus the log. Recovery independence is the notion that it should be possible to perform media recovery or restart recovery of objects at different granularities rather than only at the entire database level.

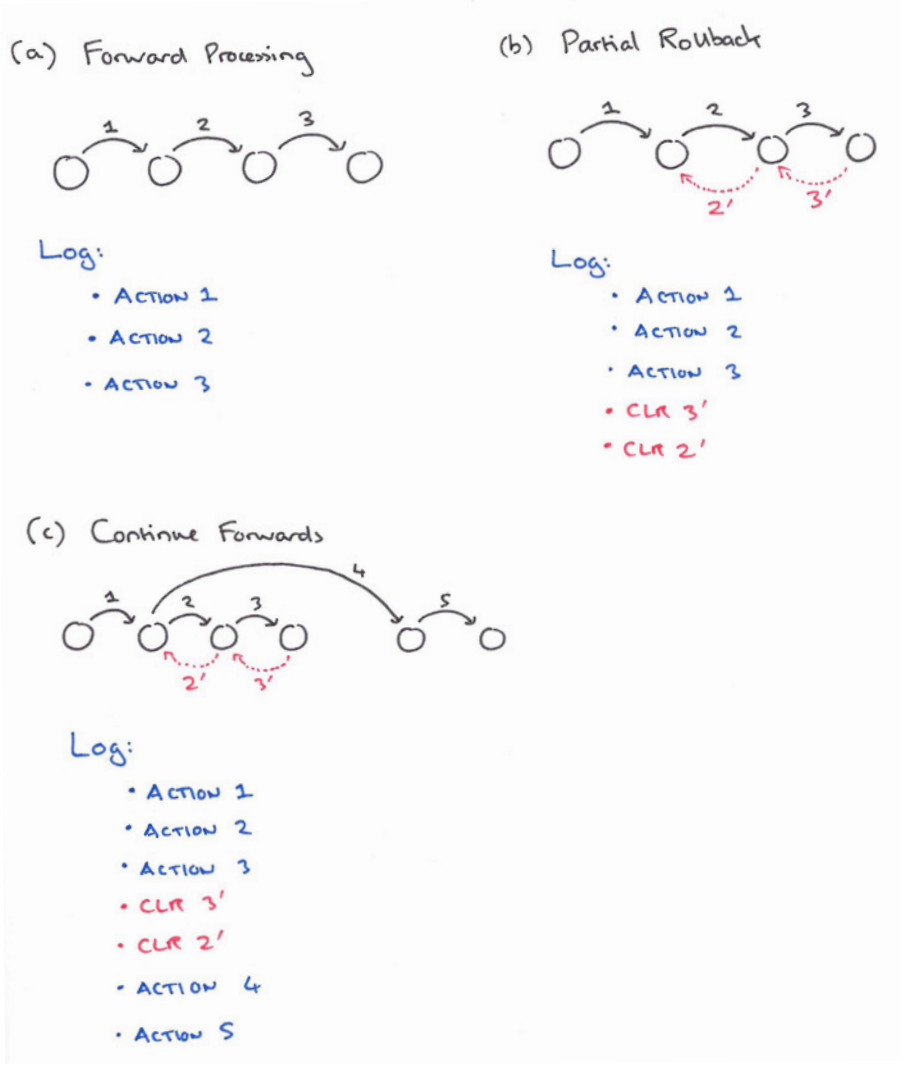
Forcing and stealing policies determine what we can assume about the relative consistency of the log and the database with respect to transaction status.

If a page modified by a transaction is allowed to be written to the permanent database on nonvolatile storage before that transaction commits, then the steal policy is said to be followed by the buffer manager... Steal implies that during normal or restart rollback, some undo work might have to be performed on the non-volatile storage version of the database. If a transaction is not allowed to commit until all pages modified by it are written to the permanent version of the database, then a force policy is said to be in effect. Otherwise, a no-force policy is said to be in effect. With a force policy, during restart recovery, no redo work will be necessary for committed transactions.

The combination of no force and steal policies offers the highest performance. This means that on recovery we many have to undo changes that have been made in the database but are not yet commited, and we may have to redo changes that have been committed but not yet made in the database. We want to support *logical* undo – that is, we should be able to undo change even if the underlying physical organisation of a database has altered since the change was originally made.

Besides logging, on a per-affected page basis, update activities performed during forward processing of transactions, ARIES also logs, typically using compensation log records (CLRs), updates performed during partial or total rollbacks of transactions during both normal and restart processing... In ARIES, CLRs have the property that they are redo-only log records. By appropriate chaning of the CLRs to log records written during forward processing, a bounded amount of logging is ensured during rollbacks, even in the face of repeated failures during restart of nested rollbacks.

In the following example, a transaction performs three actions before performing a partial rollback by undoing actions 3 and 2. It then starts going forward again and performs actions 4 and 5.



ARIES uses a single LSN on each page to track the page's state. Whenever a page is updated and a log record is written, the LSN of the log record is placed in the page_LSN field of the updated page. This tagging of the page within the LSN allows ARIES to precisely track, for restart- and media-recovery purposes, the state of the page with respect to logged updates for that page.

ARIES also takes periodic checkpoints that identify the active transactions, their states, and the LSNs of their most recently written log records as well as the dirty data in the buffer pools.

Transactions synchronously write their prepare record to the log, including the list of update-type locks (IX, X, SIX etc.) held by the transactions. Locks are logged so that they can be reacquired during recovery. Once the prepare record is written, read locks can be released. Once a transaction enters the in-doubt state, it is committed by writing an end record and releasing its locks. It is rolled back by writing a rollback record, rolling back the transaction to its beginning, discarding the pending actions list, releasing its locks, and then writing the end record.

Whether or not the rollback and end records are synchronously written to stable storage will depend on the type of two-phase commit protocol used. Also, the writing of the prepare record may be avoided if the transaction is not a distributed one or is read-only.

We now have enough background to look at how ARIES does system restart recovery. Recovery takes place in three phases or passes over the log: an *analysis* pass, a *redo* pass, and an *undo* pass.

The Analysis Pass

During the analysis pass, ARIES scans the log from the first record of the last checkpoint to the end. The purpose of the scan is to obtain information about the dirty pages and in-flight transactions as of the end of the log. The analysis pass determines:

- The starting point (redo LSN) for the following redo pass
- The list of transactions to be rolled back in the undo pass
- The LSN of the most recently written log record for each in-progress transaction

The only log records written during this phase are end records for transactions that had totally rolled back before system failure, but for whom end records are missing.

The Redo Pass

The purpose of the redo pass is to restore the database to its state as of the time of the system failure.

...during the redo pass, ARIES repeats history, with respect to those updates logged on stable storage (i.e. in the log), but whose effects on the database pages did not get reflected on nonvolatile storage before the failure of the system. This is done for the updates of all transactions, including the updates of those transactions that had neither committed nor reached the in-doubt state of two-phase commit by the time of the system failure (i.e., even the missing updates of the so-called loser transactions are redone).

A log record's update will be redone if the affected page's LSN is less than the log record's LSN. The redo pass also obtains the locks needed to protect the uncommitted updates of those distributed transactions that will remain in the in-doubt (prepared) state at the end of restart recovery.

The Undo Pass

In the third and final pass, the undo pass, all loser transactions' updates are rolled back in reverse chronological order, in a single sweep of the log.

This is done by continually taking the maximum of the LSNs of the next log record to be processed for each of the yet-to-be-completely-undone loser transactions, until no transaction remains to be un-

done. When a non-CLR is encountered for a transaction during the undo pass, if it is an undo-redo or undo-only log record then its update is undone. In any case, the next record to process for that transaction is determined by looking at the 'Previous LSN' field of that non-CLR. Since CLRs are never undone, when a CLR is encountered during undo, it is used just to determine the next log record to process.

Checkpoints

Periodic checkpoints are taken to reduce the amount of work that needs to be performed during restart recovery. Checkpoints can be taken asynchronously while transaction processing is still going on. These are called *fuzzy checkpoints*. To take a fuzzy checkpoint a begin checkpoint record is written to the log, and then when all information has been gathered an end checkpoint record will be written including within it the contents of the transaction table etc..

Once the end checkpoint record is constructed, it is written to the log. Once that information reaches stable storage, the LSN of the begin checkpoint record is stored in the master record which is kept in a well-known place on stable storage. If a failure were to occur after the end checkpoint record migrates to stable storage, but after the begin checkpoint record migrates to stable storage then that checkpoint is considered an incomplete checkpoint.

ARIES does not require that any dirty pages be forced to nonvolatile storage during a checkpoint. The assumption is that the buffer manager is continuously writing out dirty pages in the background.

It is also possible to take checkpoints during the different stages of restart recovery processing. For example, by taking a checkpoint at the end of the analysis pass, we can save some work if a failure were to occur during recovery.

Media Recovery

We assume that media recovery will be required at the level of a file or some such entity. A fuzzy image copy (also called a fuzzy archive dump) operation involving such an entity can be performed concurrency with modifications to the entity by other transactions. With such a high concurrency image copy method, the image copy might contain some uncommitted updates...

The begin checkpoint record LSN for the most recent complete checkpoint is noted when the fuzzy image copy starts – this is known as the image copy checkpoint. All updates logged in log records with LNSs less than this are assumed to have been externalized to nonvolatile storage by the time the copy operation began. Hence, the image-copied version of the entity should be at least as up to date as of that point in the log. This point is called the media recovery redo point.

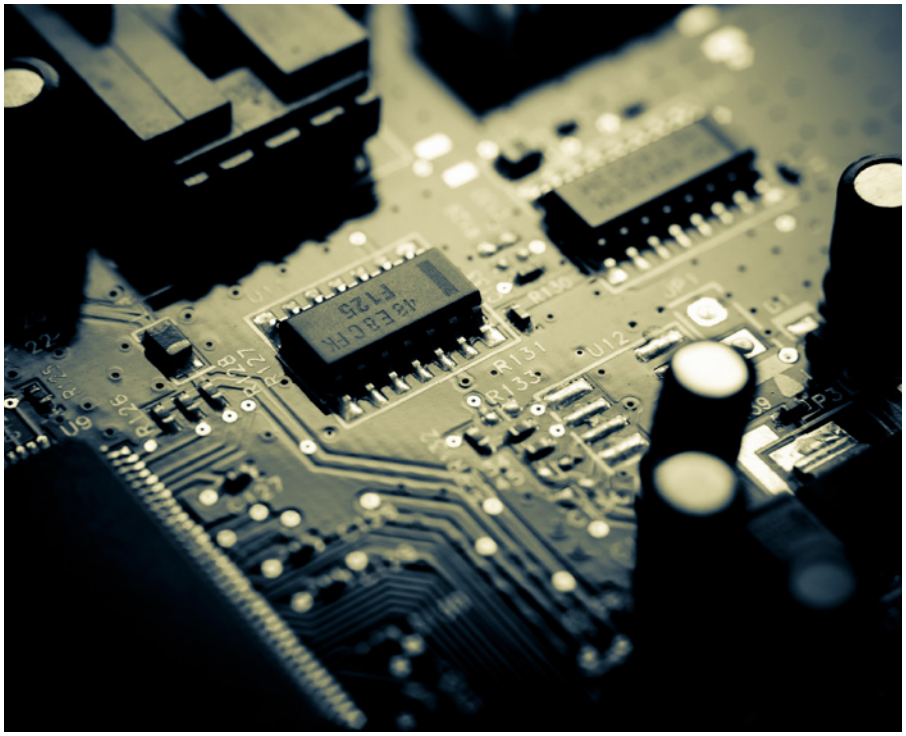
When media recovery is required, the image-copied version of the entity is reloaded and then a redo scan is initiated starting from the media recovery redo point. During the redo scan, all the log records relating to the entity being recovered are processed and the corresponding updates are applied, unless the information in the image copy checkpoint record's dirty pages list, or the LSN on the page makes it unnecessary.

It is the page-oriented logging that provides recovery independence amongst objects.

NO COMPROMISES

Distributed Transactions with Consistency, Availability, and Performance

Dragojević et al. 2015



"...a distributed UPS integrates Lithium-ion batteries into the power supply units of each chassis in a rack. The estimated cost is less than $0.005 per Joule, and it's more reliable than a traditional UPS. All these batteries mean that we can treat memory as if it was stable storage"

Let's do a thought experiment. In the last couple of days we've been looking at transaction commit protocols and assessing their cost in terms of the number of message delays and forced-writes. But suppose for a moment that network I/O and storage I/O were not the bottleneck, that those operations where really cheap. Instead, imagine if CPUs were the bottleneck. In such a crazy world would the optimum design for a transaction processing system (and datastore) look the same? What trade-offs would you make differently?

This crazy world is real, it's powered by lots of little batteries, and it changes everything! We could call it FaRMville. In FaRMville you can have distributed transactions with *strict serializability* and still get incredibly high performance, low latency, durabilty, and high availability. In FaRMville transactions are processed by FaRM – the Fast Remote Memory system that we first looked at last year. A 90 machine FaRM cluster achieved 4.5 million TPC-C 'new order' transactions per second with a 99th percentile latency of 1.9ms. If you're prepared to run at 'only' 4M tps, you can cut that latency in half. Oh, and it can recover from failure in about 60ms. Of course, since FaRMville is such a topsy-turvy world that challenges all of our assumptions, the authors had to design new trans-

action, replication, and recovery protocols from first principles...

This paper demonstrates that new software in modern data centers can eliminate the need to compromise. It describes the transaction, replication, and recovery protocols in FaRM, a main memory distributed computing platform. FaRM provides distributed ACID transactions with strict serializability, high availability, high throughput and low latency. These protocols were designed from first principles to leverage two hardware trends appearing in data centers: fast commodity networks with RDMA and an inexpensive approach to providing non-volatile DRAM.

RDMA we've encountered before on The Morning Paper – it stands for Remote Direct Memory Access. One-sided RDMA is the fastest communication method supported by RDMA and provides one-way direct access to the memory of another machine, *bypassing the CPU*.

DRAM in the data center is becoming plentiful and cheap:

A typical data center configuration has 128-512GB of DRAM per 2-socket machine, and DRAM costs less than $12/GB. This means that a petabyte of DRAM requires only 2000 machines, and this is sufficient to hold the data sets of many interesting applications.

It's those 'lots of little batteries' that I mentioned earlier that make things really interesting though... Instead of a centralized, expensive UPS (uninterruptible power supply), a distributed UPS integrates Lithium-ion batteries into the power supply units of each chassis in a rack. The estimated cost is less than $0.005 per Joule, and it's more reliable than a traditional UPS. All these batteries mean that we can treat memory as if it was stable storage...

A distributed UPS effectively makes DRAM durable. When a power failure occurs, the distributed UPS saves the contents of memory to a commodity SSD using the energy from the battery. This not only improves common-case performance by avoiding synchronous writes to SSD, it also preserves the lifetime of the SSD by writing to it only when failures occur. An alternative approach is to use non-volatile DIMMs (NVDIMMs), which contain their own private flash, controller and supercapacitor (e.g., [2]). Unfortunately, these devices are specialized, expensive, and bulky. In contrast, a distributed UPS uses commodity DIMMs and leverages commodity SSDs. The only additional cost is the reserved capacity on the SSD and the UPS batteries themselves.

In the *worst case* configuration, it costs about $0.55/GB for non-volatility using this approach, and about $0.90/GB for the storage cost of the reserved SSD capacity. The combined additional cost is therefore about 15% of the base DRAM cost.

The combination of high-performance networking with RDMA and the ability to treat memory as stable storage eliminates network and storage bottlenecks, and as a consequence exposes CPU bottlenecks.

FaRM's protocols follow three principles to address these CPU bottlenecks: reducing message counts, using one-sided RDMA reads and writes instead of messages, and exploiting parallelism effectively.

The reason to reduce message counts is all of the CPU involved in marshaling, unmarshmaling, and dispatching (we've seen serialization and deserialization become the bottleneck before, even in less extreme hardware scenarios). FaRM uses primary-backup replication and an optimistic concurrency four-phase commit protocol.

One-sided RDMA enables direct access to remote data without involving the remote CPU at all, and places minimal requirements on the local CPU. But if there's no remote CPU involvement, there's not going to be any additional processing in the remote region either, and that requires thinking about the design of the system a little differently. (The processing of one-sided RDMA requests is handled in the NIC).

FaRM transactions use one-sided RDMA reads during transaction execution and validation. Therefore, they use no CPU at remote read-only participants. Additionally, coordinators use one-sided RDMA when logging records to

non-volatile write-ahead logs at the replicas of objects modified in a transaction. For example, the coordinator uses a single one-sided RDMA to write a commit record to a remote backup. Hence, transactions use no foreground CPU at backups. CPU is used later in the background when lazily truncating logs to update objects in-place.

Because it's so unusual, it's worth stressing again that these write-ahead logs are simply held in memory, because memory is stable storage.

Failure detection and recovery also has to look a little different...

For example, FaRM cannot rely on servers to reject incoming requests when their leases expire because requests are served by the NICs, which do not support leases. We solve this problem by using precise membership to ensure that machines agree on the current configuration membership and send one-sided operations only to machines that are members. FaRM also cannot rely on traditional mechanisms that ensure participants have the resources necessary to commit a transaction during the prepare phase because transaction records are written to participant logs without involving the remote CPU. Instead, FaRM uses reservations to ensure there is space in the logs for all the records needed to commit and truncate a transaction before starting the commit.

Transaction Model

FaRM uses optimistic concurrency and provides strict serializability. During the execution phase of a transaction, objects reads happen from memory (local access or via RDMA), and all writes are buffered locally. The address and version of every object accessed is recorded. At the end

The combination of high-performance networking with RDMA and the ability to treat memory as stable storage eliminates network and storage bottlenecks, and as a consequence exposes CPU bottlenecks.

of the transaction, the commit protocol proceeds as follows:

1. **Lock phase:** the coordinator writes a LOCK record to the log on each machine that is a primary for any written object. The record contains the versions and new values for every written object. Primaries process these messages by attempting to lock the objects at the specified version using compare-and-swap. They send back a message indicating whether or not all locks were successfully taken. If any object version has changed since it was read by the transaction, or another transaction has the object locked, then the coordinator aborts the transaction and writes an abort record to the log on all primaries.

2. Validation phase: the coordinator now checks that the versions of all objects read by the transaction have not changed. This can be done via one-sided RDMA with no remote CPU involvement. If a primary holds more than some threshold tr objects then an RPC is used instead (the current cut-off point where RPC becomes cheaper is 4). The transaction is aborted if the versions have changed.

3. **Commit backups phase:** the coordinator writes a commit-backup record to the logs at each backup, and waits for an ack from the NIC (remote CPU is not involved).

4. **Commit primaries phase:** once acks have been received from the NICs at every backup (this is necessary to guarantee strict serialization in the presence of failures), the coordinator writes a commit-primary record to the logs at each primary. Completion can be reported to the application once at least one hardware ack is received. Primaries process these records by updating objects in place, incrementing their versions, and unlocking them.

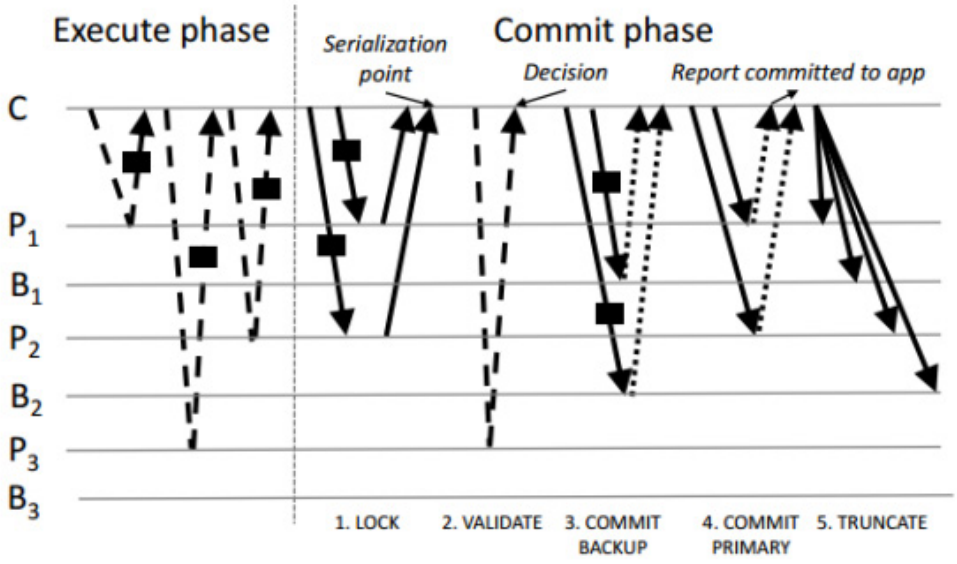
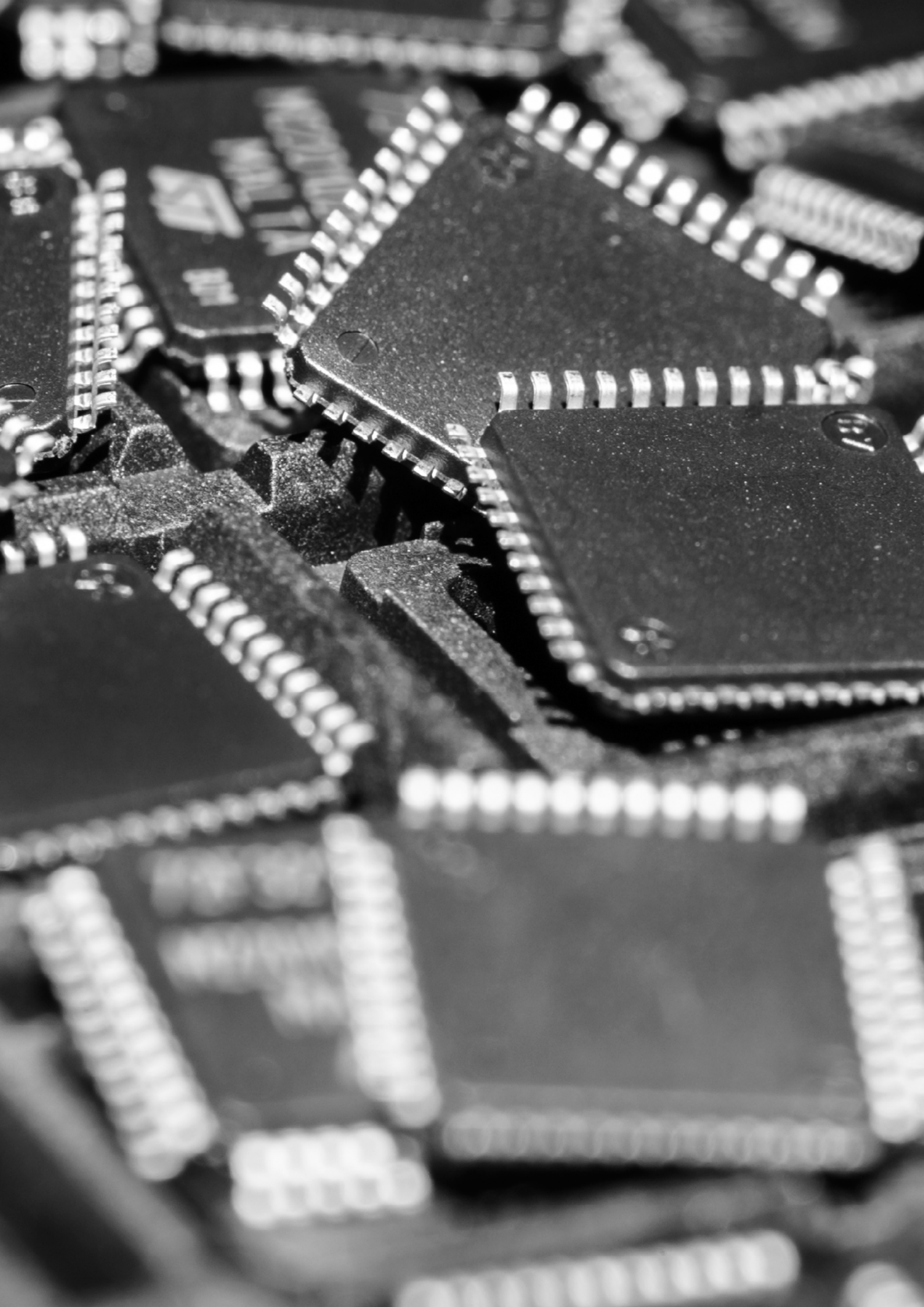


Figure 4. FaRM commit protocol with a coordinator C, primaries on P_1, P_2, P_3, and backups on B_1, B_2, B_3. P_1 and P_2 are read and written. P_3 is only read. We use dashed lines for RDMA reads, solid ones for RDMA writes, dotted ones for hardware acks, and rectangles for object data.



5. **Truncation:** "Backups and primaries keep the records in their logs until they are truncated. The coordinator truncates logs at primaries and backups lazily after receiving acks from all primaries. It does this by piggybacking identifiers of truncated transactions in other log records. Backups apply the updates to their copies of the objects at truncation time."

The FaRM commit phase uses Pw(f + 3) one-sided RDMA writes where Pw is the number of machines that are primaries for objects written by the transaction, and Pr one-sided RDMA reads where Pr is the number of objects read from remote primaries but not written. Read validation adds two one-sided RDMA latencies to the critical path but this is a good trade-off: the added latency is only a few microseconds without load and the reduction in CPU overhead results in higher throughput and lower latency under load.

Epochs

FaRM relies on precise knowledge of the primary and backups for every data region. After a configuration change (for example, because a machine failed and is being replaced), all machines must agree on the membership of the new configuration before allowing object mutations. FaRM *reconfiguration protocol* is responsible for moving the system from one configuration to the next.

Using one-sided RDMA operations is important to achieve good performance but it imposes new requirements on the reconfiguration protocol. For example, a common technique to achieve consistency is to use leases : servers check if they hold a lease for an object before replying to requests to access the object. If a server is evicted from the configuration, the system guarantees that the objects it stores cannot be mutated until after its lease expires. FaRM uses this technique when servicing requests from external clients that communicate with the system using messages. But since machines in the FaRM configuration read objects using RDMA reads without involving the remote CPU, the server's CPU cannot check if it holds the lease. Current NIC hardware does not support leases and it is unclear if it will in the future.

When a machine receives a NEW-CONFIG message with a configuration identifier greater than its own, it *prepares* to move to the new configuration by updating its current configuration identifier and copy of the cached region mapping, and allocates space for any new regions assigned to it. "From this point, it does not issue new requests to machines that are not in the configuration and it rejects read responses and write acks from those machines. It also starts blocking requests from external clients." Machines reply to the configuration manager with a NEW-CONFIG-ACK. Once acks have been received from all machines in the new configuration, the coordinator *commits* the new configuration by sending a NEW-CONFIG-COMMIT message. Members then unblock previously blocked external client requests, and initiate transaction recovery.

Fault detection and Recovery

We provide durability for all committed transactions even if the entire cluster fails or loses power: all committed state can be recovered from regions and logs stored in non-volatile DRAM.We ensure durability even if at most f replicas per object lose the contents of non-volatile DRAM. FaRM can also maintain availability with failures and network par-

titions provided a partition exists that contains a majority of the machines which remain connected to each other and to a majority of replicas in the Zookeeper service, and the partition contains at least one replica of each object.

Failure recovery has five phases:

1. **Failure detection**, which uses *very short* leases (5-10ms). Expiry of any lease triggers recovery. A 90-node cluster can use 5ms leases with no false positives. "Achieving short leases under load required careful implementation" !

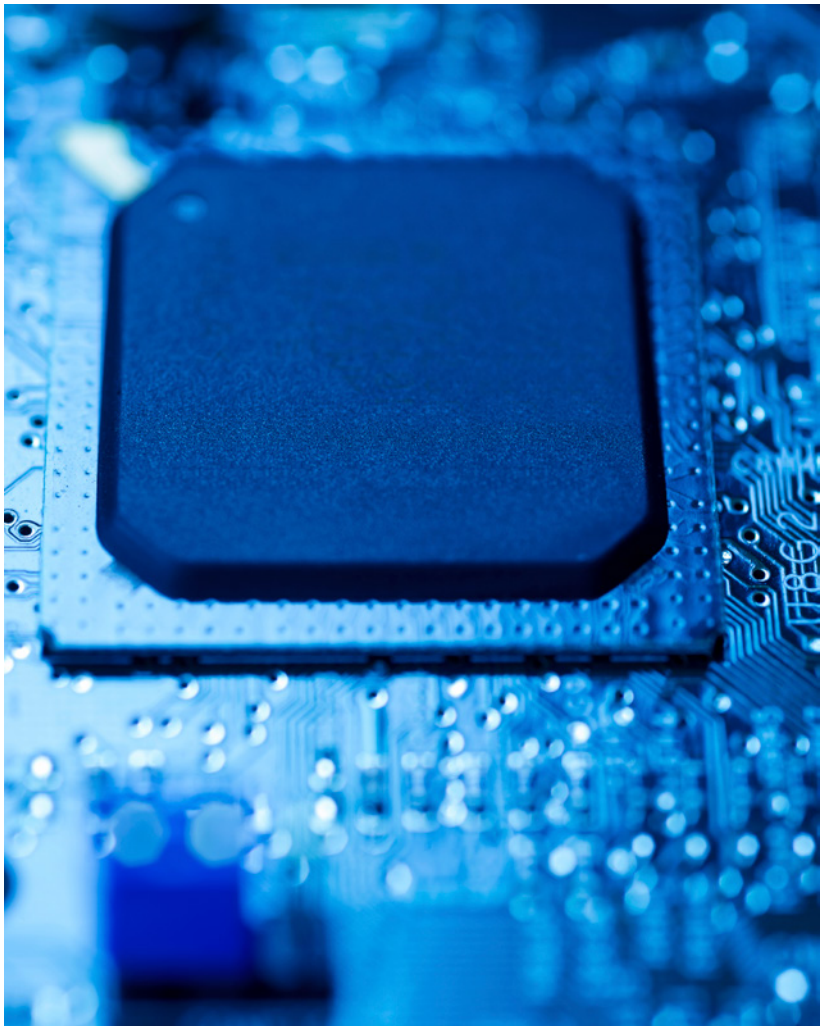
2. **Reconfiguration**, using the reconfiguration protocol briefly described above.

3. **Transaction state recovery**, using the logs distributed across the replicas of objects modified by the transaction being recovered. First access is blocked to recovering regions until all write locks for recovering transactions have been acquired. Then the logs are drained to ensure that all relevant records are processed during recovery. During the log draining process the set of recovering transactions is determined (those whose commit phase spans configuration changes). Information about these transactions is exchanged via NEED-RECOVERY messages. Then locks are acquired, log records replicated so that every participant has a full set, and the coordinator drives a commit protocol to decide whether to commit or abort. The coordinator for a given transaction is determined by consistent hashing.

4. **Bulk data recovery**. Data recovery (re-replicating data to ensure that f replica failures can be tolerated in the future) is not necessary to resume normal operation, so it is delayed until all regions become active to minimize impact on latency-critical lock recovery.

5. **Allocator state recovery** – the FaRM allocator splits regions into 1MB blocks used as slabs for allocating small objects. Free lists need to be recovered on the new primary by scanning the objects in the region, which is parallelized across all threads on the machine.

In the TPC-C test, the system regains most throughput in less than 50ms after the loss of a node. Full data recovery takes just over 4 minutes.



ALL FILE SYSTEMS ARE NOT CREATED EQUAL

On the Complexity of Crafting Crash Consistent Applications

Pillai et al. 2014

Last week we looked at Reducing Crash Recoverability to Reachability for file system-based applications. Today's choice predates that work and investigates the semantics of real world file systems and how this affects applications that run on top of them. It's a real eye-opener, and essential reading if you're building (or operating!) anything that you expect to be able to recover from crashes and that interacts with a file system. The bottom line: there are lots of vulnerabilities out there, and an application that works correctly with one file system may not do so with another one.

Many important applications, including databases such as SQLite and key-value stores such as LevelDB, are currently implemented on top of (these) file systems instead of directly on raw disks. Such data management applications must also be crash consistent, but achieving this goal atop modern file systems is challenging for two fundamental reasons...

1. The guarantees provided by file systems are unclear and underspecified. While the POSIX standard specifies the effect of a system call in memory, specifications of how disk state is mutated in the event of a crash are widely misunderstood and debated. Each file system does it slightly differently, and each has a plethora of options that can modify its behaviour.
2. In a quest for performance, most applications implement complex update protocols.

BOB, the Block Order Breaker, is used to find out what behaviours are exhibited by a number of modern file systems that are relevant to building crash consistent applications. ALICE, the Application Level Intelligent Crash Explorer is then used to explore the crash recovery behaviour of a number of applications on top of these file systems.

We use ALICE to study and analyze the update protocols of eleven important applications: LevelDB, GDBM, LMDB,

SQLite, PostgreSQL, HSQLDB, Git, Mercurial, HDFS, ZooKeeper, and VMWare Player... Overall, we find that application-level consistency in these applications is highly sensitive to the specific persistence properties of the underlying file system. In general, if application correctness depends on a specific file system persistence property, we say the application contains a crash vulnerability; running the application on a different file system could result in incorrect behavior. We find a total of 60 vulnerabilities across the applications we studied; several vulnerabilities have severe consequences such as data loss or application unavailability. Using ALICE, we also show that many of these vulnerabilities (roughly half) manifest on current file systems such as Linux ext3, ext4, and btrfs.

Exploring File System Behaviours with BOB

BOB explores the persistence properties of modern file systems. In particular it examines atomicity – does the update arising from a system call happen all at once – and ordering – can system calls be reordered? Consider this program fragment:

```
write(f1,"pp")

write(f2,"qq")
```

All sorts of fun things might happen in the event of a crash. If the write operation is not atomic for example the file size could be updated but the content might not be written (file contains garbage), or just one "p" might be written into f1. Or if the calls are persisted out of order, f2 might contain "qq", but f1 might not contain "pp". Any combination of these is possible.

Ext2, ext3, ext4, xfs, btrfs, and reiserfs are studied with a variety of configurations (16 in total).

BOB collects the block I/O generated by the workload, and then re-orders the collected blocks, selectively writing some of them to disk to generate a new legal disk state (disk barriers are obeyed). In this manner, BOB generates a number of unique disk images corresponding to possible on-disk states after a system crash. BOB then runs file-system recovery on each resulting disk image, and checks whether various persistence properties hold (e.g., if writes were atomic).

BOB can find failures but can't prove their absence. Here's what BOB finds for the configurations under test – an x indicates that the property does not hold for the file system in question.

Persistence Property	File system															
	ext2	ext2-sync	ext3-writeback	ext3-ordered	ext3-datajournal	ext4-writeback	ext4-ordered	ext4-nodelalloc	ext4-datajournal	btrfs	xfs	xfs-wsync	reiserfs-nolog	reiserfs-writeback	reiserfs-ordered	reiserfs-datajournal
Atomicity																
Single sector overwrite																
Single sector append	×		×			×								×		
Single block overwrite	×	×	×	×		×	×	×			×	×	×	×	×	
Single block append	×	×	×			×							×	×		
Multi-block append/writes	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Multi-block prefix append	×	×	×			×							×	×		
Directory op	×	×											×			
Ordering																
Overwrite → Any op	×		×	×		×	×	×			×	×	×	×	×	
[Append, rename]→ Any op	×		×			×							×	×		
O_TRUNC Append → Any op	×		×			×							×	×		
Append → Append (same file)	×		×			×							×	×		
Append → Any op	×		×			×	×			×	×		×	×		
Dir op → Any op	×									×			×			

Table 1: **Persistence Properties.** *The table shows atomicity and ordering persistence properties that we empirically determined for different configurations of file systems. X → Y indicates that X is persisted before Y. [X,Y] → Z indicates that Y follows X in program order, and both become durable before Z. A × indicates that we have a reproducible test case where the property fails in that file system.*

From Table 1, we observe that persistence properties vary widely among file systems, and even among different configurations of the same file system. The order of persistence of system calls depends upon small details like whether the calls are to the same file or whether the file was renamed. From the viewpoint of an application developer, it is risky to assume that any particular property will be supported by all file systems. (emphasis mine).

Exploring Application Level Behaviours with ALICE

ALICE finds the generic persistence properties required for an application to behave correctly. It does this by constructing file system states directly from the system call trace of an application workload.

The user first supplies ALICE with an initial snapshot of the files used by the application (typically an entire directory), and a workload script that exercises the application (such as performing a transaction). The user also supplies a checker script corresponding to the workload that verifies whether invariants of the workload are maintained (such as atomicity of the transaction). ALICE runs the checker atop different crash states, i.e., the state of files after rebooting from a system crash that can occur during the workload. ALICE produces a logical representation of the update protocol executed during the workload, vulnerabilities in the protocol and their associated source lines, and persistence properties required for correctness.

To capture intermediate crash states, ALICE breaks logical operations down into micro-operations – the smallest atomic operations that can be performed on each logical entity. These

> **A common heuristic to prevent data loss on file systems with delayed allocation is to persist all data of a file before subsequently renaming – the bad news is that this tactic only fixes a small minority (3) of vulnerabilities.**

are: write block, change file size, create dir entry, delete dir entry, and adding messages to terminal output (stdout). An Application Persistence Model for a given run specifies atomicity constraints by defining how logical operations map into micro-operations.

ALICE selects different sets of the translated micro-ops that obey the ordering constraints. A new crash state is constructed by sequentially applying the micro-ops in a selected set to the initial state (represented as logical entities). For each crash state, ALICE then converts the logical entities back into actual files, and supplies them to the checker. The user-supplied checker thus verifies the crash state.

What ALICE revealed

34 different configuration options were tested across the 11 applications listed earlier.

Application	Types									Unique static vulnerabilities
	Across-syscalls atomicity	Atomicity			Ordering			Durability		
		Appends and truncates	Single-block overwrites	Renames and unlinks	Safe file flush	Safe renames	Other	Safe file flush	Other	
Leveldb1.10	1‡	1		1	2	1	3	1		10
Leveldb1.15	1	1		1	1		2			6
LMDB			1							1
GDBM	1			1			1		2	5
HSQLDB		1		2	1		3	2	1	10
Sqlite-Roll									1	1
Sqlite-WAL										0
PostgreSQL			1							1
Git	1			1	2	1	3		1	9
Mercurial	2	1		1		1	4		2	10
VMWare				1						1
HDFS				1			1			2
ZooKeeper			1				1	2		4
Total	**6**	**4**	**3**	**9**	**6**	**3**	**18**	**5**	**7**	**60**

(a) Types.

Application	Silent errors	Data loss	Cannot open	Failed reads and writes	Other
Leveldb1.10	1	1	5	4	
Leveldb1.15	2		2	2	
LMDB					read-only open†
GDBM		2*	3*		
HSQLDB	2	3	5		
Sqlite-Roll		1*			
Sqlite-WAL					
PostgreSQL			1†		
Git		1*	3*	5*	3#*
Mercurial		2*	1*	6*	5 dirstate fail*
VMWare			1*		
HDFS			2*		
ZooKeeper		2*	2*		
Total	**5**	**12**	**25**	**17**	**9**

(b) Failure Consequences.

ALICE finds 60 static vulnerabilities in total, corresponding to 156 dynamic vulnerabilities. Altogether, applications failed in more than 4000 crash states. Table 3(a)shows the vulnerabilities classified by the affected persistence property, and 3(b) shows the vulnerabilities classified by failure consequence. Table 3(b) also separates out those vulnerabilities related only to user expectations and not to documented guarantees, with an asterik (many of these correspond to applications for which we could not find any documentation of guarantees.

We find many vulnerabilities have severe consequences such as silent errors or data loss. Seven applications are affected by data loss, while two (both LevelDB versions and HSQLDB) are affected by silent errors. The cannot open failures include failure to start the server in HDFS and ZooKeeper, while the failed reads and writes include basic commands (e.g., git-log, git-commit) failing in Git and Mercurial. A few cannot open failures and failed reads and writes might be solvable by application experts, but we believe lay users would have difficulty recovering from such failures (our checkers invoke standard recovery techniques).

Four applications require atomicity across system calls, and three require appends to be content-atomic: the appended portion should contain actual data. LMDB, PostgreSQL and ZooKeeper require small writes (Applications are extremely vulnerable to system calls being persisted out of order; we find 27 vulnerabilities.

A common heuristic to prevent data loss on file systems with delayed allocation is to persist all data of a file before subsequently renaming – the bad news is that this tactic only fixes a small minority (3) of vulnerabilities.

The vulnerabilities exposed vary based on the file system, and thus testing applications on only a few file systems does not work.

NOT-QUITE-SO-BROKEN TLS

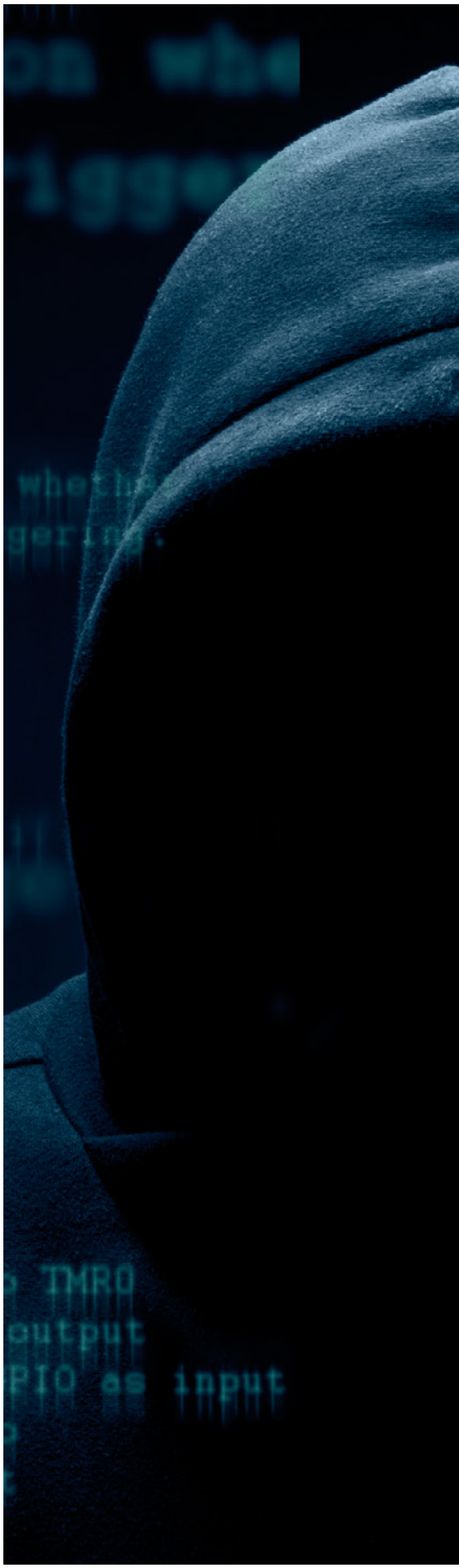
Lessons in re-engineering a security protocol specification and implementation

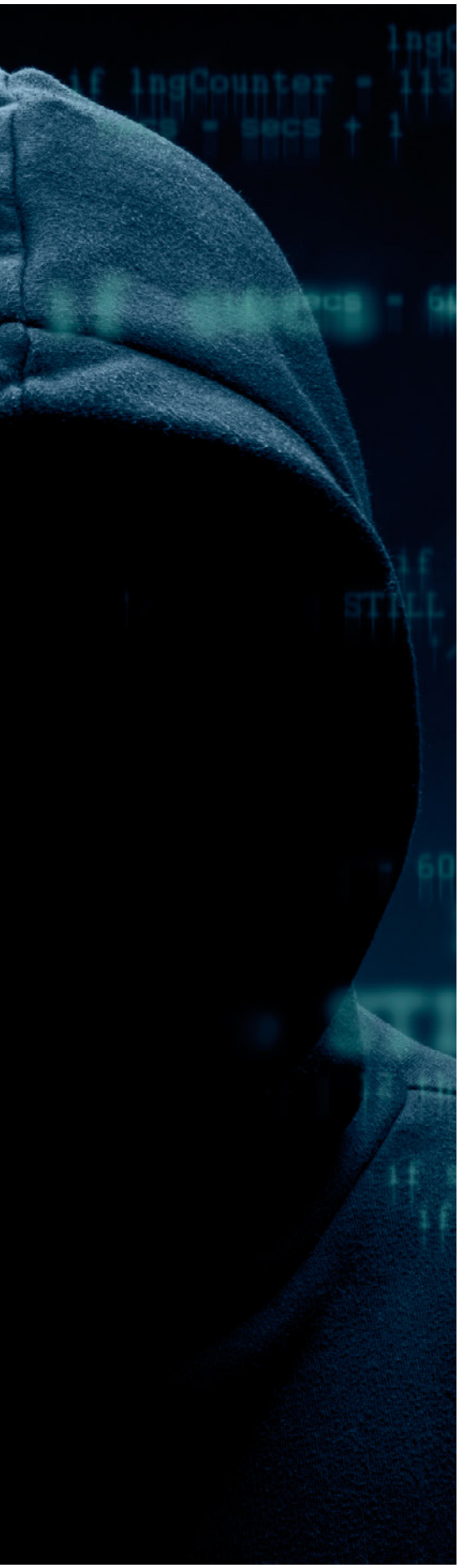
Kaloper-Meršinjak et al. 2015

"Transport Layer Security (TLS) is the most widely deployed security protocol on the Internet, used for authentication and confidentiality, but a long history of exploits shows that its implementations have failed to guarantee either property."

On the surface this is a paper about a TLS implementation, but the really interesting story to me is the attempt to 'do it right,' and the techniques and considerations involved in that process. The IT landscape is littered with bug-ridden and vulnerable software – surely we can do better? And if we're going to make a serious attempt at that, where better than something like a TLS stack – because bugs and vulnerabilities there also expose everything that relies on it – i.e. pretty much the whole of the internet. The authors certainly don't pull any punches when it comes to explaining what they think of the typical way of building such software...

Current mainstream engineering practices for specifying and implementing security protocols are not fit for purpose: as one can see from many recent compromises of sensitive services, they are not providing the security we need. Transport Layer Security (TLS)





is the most widely deployed security protocol on the Internet, used for authentication and confidentiality, but a long history of exploits shows that its implementations have failed to guarantee either property. Analysis of these exploits typically focusses on their immediate causes, e.g. errors in memory management or control flow, but we believe their root causes are more fundamental.

And what might those root causes be? Poor choice of programming languages, that tend to encourage errors rather than protect against them; poor software engineering – especially a lack of attention on a clean separation of concerns and modularity; and an insistence on ambiguous and untestable prose specifications. All well and good, but what should we do instead? How about:

- A precise and testable specification (in this case for TLS) that unambiguously determines the set of behaviours it allows (and hence also what it does not). The specification should also be executable as a test oracle, to determine whether or not a given implementation is compliant.

We specify TLS as a collection of pure functions over abstract datatypes. By avoiding I/O and shared mutable state, these functions can be considered in isolation and each is deterministic, with errors returned as explicit values. The top-level function takes an abstract protocol state and an incoming message, and calculates the next state and any response messages... In building our specification, to resolve the RFC ambiguities, we read other implementations and tested interoperability with them; we thereby capture the practical de facto standard.

- Reuse between specification and implementation, to the extent possible.

The same functions form the main part of our implementation, coupled with code for I/O and to provide entropy. Note that this is not an

"executable specification" in the conventional sense: our specification is necessarily somewhat loose, as the server must have the freedom to choose a protocol version and cipher suite, and the trace checker must admit that, while our implementation makes particular choices.

- Cleanly separating concerns in a modular structure:

This focus on pure functional descriptions also enables a decomposition of the system (both implementation and specification) into strongly separated modules, with typed interfaces, that interact only by exchanging pure values.

- Choosing a programming language and style that guarantees memory and type-safety.

The structure we describe above could be implemented in many different programming languages, but guarantees of memory and type safety are desirable to exclude many common security flaws (lack of memory safety was the largest single source of vulnerabilities in various TLS stacks throughout 2014, as shown in our §3 vulnerability analysis), and expressive statically checked type and module systems help maintain the strongly separated structure that we outlined. Our implementation of nqsb-TLS uses OCaml, a memory-safe, statically typed programming language that compiles to fast native code with a lightweight, embeddable runtime.

Let's look a little deeper at how these choices helped the nqsb-TLS team avoid some of the common vulnerabilities found in other TLS stacks (based on reported CVEs in 2014)...

In the past 13 months (January 2014 to January 2015), 54 CVE security advisories have been published for 6 widely used TLS implementations... These vulnerabilities have a wide range of causes. We classify them into broad families below, identifying root causes for each and discussing how nqsb-TLS avoids flaws of each kind.

Memory Safety Violations (15/54)

Most of these bugs, 15 in total, are memory safety issues: out-of-bounds reads, out-of-bounds writes and NULL pointer dereferences. A large group has only been demonstrated to crash the hosting process, ending in denial-of-service, but some lead to disclosure of sensitive information.

A now-notorious example of this class of bugs is Heartbleed in OpenSSL (CVE-2014-0160)... nqsb-TLS avoids this class of issues entirely by the choice of a programming language with automated memory management and memory safety guarantees: in OCaml, array bounds are always checked and it is not possible to access raw memory; and our pure functional programming style rules out reuse of mutable buffers.

Certificate Parsing Bugs (3/54)

TLS implementations need to parse ASN.1, primarily for decoding X.509 certificates...

Since ASN.1 is a large and complex standard, and TLS needs only a subset, hand-built parsers are common:

Unsurprisingly, ASN.1 parsing is a recurrent source of vulnerabilities in TLS and related software, dating back at least to 2004... This class of errors is due to ambiguity in the specification, and ad-hoc parsers in most TLS imple-

mentations. nqsb-TLS avoids this class of issues entirely by separating parsing from the grammar description (we created a library for declaratively describing ASN.1 grammars in OCaml, using a functional technique known as combinatory parsing).

Certificate Validation Bugs (5/54)

Closely related to ASN.1 parsing is certificate validation. X.509 certificates are nested data structures standardised in three versions and with various optional extensions, so validation involves parsing, traversing, and extracting information from complex compound data... Many implementations interleave the complicated X.509 certificate validation with parsing the ASN.1 grammar, leading to a complex control flow with subtle call chains. This illustrates another way in which the choice of programming language and style can lead to errors: the normal C idiom for error handling uses goto and negative return values, while in nqsb-TLS we return errors explicitly as values and have to handle all possible variants. OCaml's type checker and pattern-match exhaustiveness checker ensures this at compile time.

Algebraic data types are used "as the control flow of functions that navigate this structure (the parse tree) is directed by the type checker." The 7000 lines of text in the RFC end up being encoded in 314 lines of easily reviewable code.

State Machine Errors (10/54)

TLS consists of several subprotocols that are multiplexed at the record level... There were 10 vulnerabilities in this class. Some led to denial-of-service conditions caused (for example) by NULL-pointer dereferences on receipt of an unexpected message, while others lead to a breakdown of the TLS security guarantees... A prominent example is Apple's "goto fail" (CVE-2014-1266), caused by a repetition of a goto statement targeting the cleanup block of the procedure responsible for verifying the digital signature of the ServerKeyExchange message.

In nqsb-TLS, the state machine is explicitly encoded.

This construction [pure composable functions] and the explicit encoding of state-machine is central to maintaining the state-machine invariants and preserving the coherence of state representation. It is impossible to enter a branch dedicated to a particular transition without the pair of values representing the appropriate state and appropriate input message, and, as intermediate data is directly obtained from the state value, it is impossible to process it without at the same time requiring that the state-machine is in the appropriate state. It is also impossible to manufacture a state-representation ahead of time, as it needs to contain all of the relevant data.

Protocol Bugs (2/54)

Two separate issues in the SSL v3 protocol itself were discovered in 2014. While the nsqd-TLS engineering approach does not claim to prevent or solve such bugs, the actual implementation focuses on a modern subset not including SSL v3 so nqsb-TLS is not vulnerable to these bugs.

Timing side-channel leaks (2/54)

Two vulnerabilities were related to timing side-channel leaks, where the observable duration of cryptographic operations depended on cryptographic secrets. To mitigate timing side-channels, which a memory managed programming language might further ex-

Of the examined bugs, 10 were not in TLS implementations themselves, but in the way the client software used them. These included the high profile anonymisation software Tor, the instant messenger Pidgin and the widely used multi-protocol data transfer tool cURL

pose, we use C implementations of the low level primitives.

A second reason C is used at this level is that symmetric encryption and hashing are the most CPU-intensive operations in TLS, and so performance concerns motivate the use of C.

Such treatment creates a potential safety problem in turn: even if we manage to prevent certain classes of bugs in OCaml, they could occur in our C code. Our strategy to contain this is to restrict the scope of C code: we employ simple control flow and never manage memory in the C layer. C functions receive pre-allocated buffers, tracked by the runtime, and write their results there.

More complex cryptographic constructions are implemented in OCaml on top of the C-level primitives.

Misuse of TLS libraries (10/54)

Of the examined bugs, 10 were not in TLS implementations themselves, but in the way the client software used them. These included the high profile anonymisation software Tor, the instant messenger Pidgin and the widely used multi-protocol data transfer tool cURL... The root cause of this error class is the large and complex legacy APIs of contemporary TLS stacks. nqsb-TLS does not mirror those APIs, but provides a minimal API with strict validation by default. This small API is sufficient for various applications we developed. OpenBSD uses a similar approach with their libtls API.

So how confident can we be in the nqsb-TLS implementation?

We assess the security of nqsb-TLS in several ways: the discussion of the root causes of many classic vulnerabilities and how we avoid them; mitigation

of other specific issues; our state machine was tested; random testing with the Frankencert fuzzing tool; a public integrated system protecting a bitcoin reward; and analysis of the TCB size of that compared with a similar system built using a conventional stack.

I'm going to highlight just the bitcoin pinata and TCB size comparison here.

To demonstrate the use of nqsb-TLS in an integrated system based on MirageOS, and to encourage external code-review and penetration testing, we set up a public bounty, the Bitcoin Pinata. This is a standalone MirageOS unikernel containing the secret key to a bitcoin address, which it transmits upon establishing a successfully authenticated TLS connection. The service exposes both TLS client and server on different ports, and it is possible to bridge the traffic between the two and observe a successful handshake and the encrypted exchange of the secret.

As of June 2105 there were more than 230,000 accesses from more than 50,000 unique IP addresses.

A detailed analysis of the captured traces showed that most of the flaws in other stacks have been attempted against the Pinata.

To the best of my knowledge, the bitcoins are still there...

The TCB [Trusted Compute Base] size is a rough quantification of the attack surface of a system. We assess the TCB of our Pinata, compared to a similar traditional system using Linux and OpenSSL. Both systems are executed on the same hardware and the Xen hypervisor, which we do not consider here. The TCB sizes of the two systems are shown in Table 2 (using cloc)... The trusted computing base of the traditional system is 25 times larger than ours. Both systems provide the same service to the outside world and are hardly distinguishable for an external observer.

I guess you may also be wondering about performance: the handshake performance is the same as OpenSSL, and achieves 73-84% of the bulk throughput.

So can we do better?

I opened this write-up by suggesting that surely we can do better than the majority of buggy and vulnerable implementations of systems software that proliferate in our industry. What this paper nicely illustrates is that there is no one silver bullet. But by making careful choices, being aware of the implications of those choices, and using a variety of validation techniques (in this case, including the test oracle and bitcoin pinata), yes we can indeed do a lot better than the state of the practice.

HYPERLOGLOG IN PRACTICE

Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm

Heule et al. 2013

Continuing on the theme of approximations from yesterday, today's paper looks at what must be one of the best known approximate data structures after the Bloom Filter, HyperLogLog. It's HyperLogLog with a twist though – a paper describing HyperLogLog in practice at Google and a number of enhancements they made to the base algorithm to make it work even better.

At Google, various data analysis systems such as Sawzall, Dremel and PowerDrill estimate the cardinality of very large data sets every day, for example to determine the number of distinct search queries on google.com over a time period. Such queries represent a hard challenge in terms of computational resources, and memory in particular: For the PowerDrill system, a non-negligible fraction of queries historically could not be computed because they exceeded theavailable memory. In this paper we present a series of improvements to the HyperLogLog algorithm by Flajolet et. al. that estimates cardinalities efficiently. Our improvements decrease memory usage as well as increase the accuracy of the estimate significantly for a range of important cardinalities.

In PowerDrill (and other database systems), a user can count the number of distinct elements in a data set by issuing a count distinct query. Such a query often contains a group-by clause

"At Google, various data analysis systems such as Sawzall, Dremel and PowerDrill estimate the cardinality of very large data sets every day, for example to determine the number of distinct search queries on google.com over a time period"

meaning that a single query can lead to many count distinct computations being carried out in parallel. PowerDrill performs about 5 million such computations, 99% of these yielding a result of 100 or less. About 100 computations a day yield a result greater than 1 billion though. The key requirements at Google are therefore:

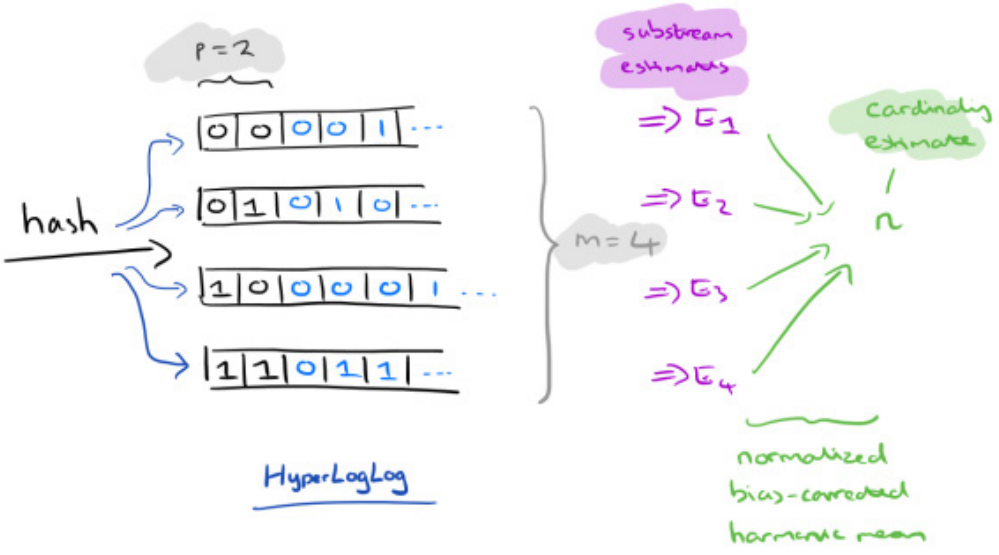
- As accurate an estimate as possible within a fixed memory budget, and for small cardinalities the result should be near exact.
- Efficient use of memory
- The ability to estimate large cardinalities (1 billion+) with reasonable accuracy
- The algorithm should be (easily) implementable and maintainable

As a quick refresher, the basis of HyperLogLog is hashing every item to be counted, and remembering the maximum number of leading zeros that occur for each hash value. If the bit pattern starts with d zeros, then a good estimate of the size of the multiset is 2d+1. Instead of relying on a single measurement, variability is reduced by splitting the input stream of data elements S into m substreams of roughly equal size by using the first p bits of the hash values (m=2p). Therefore we count the number of leading zeros after the first p bits. The final cardinality estimate is produced from the substream estimates as the normalised bias corrected harmonic mean:

$$E := \alpha_m \cdot m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]} \right)$$

where

$$\alpha_m := \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1}$$



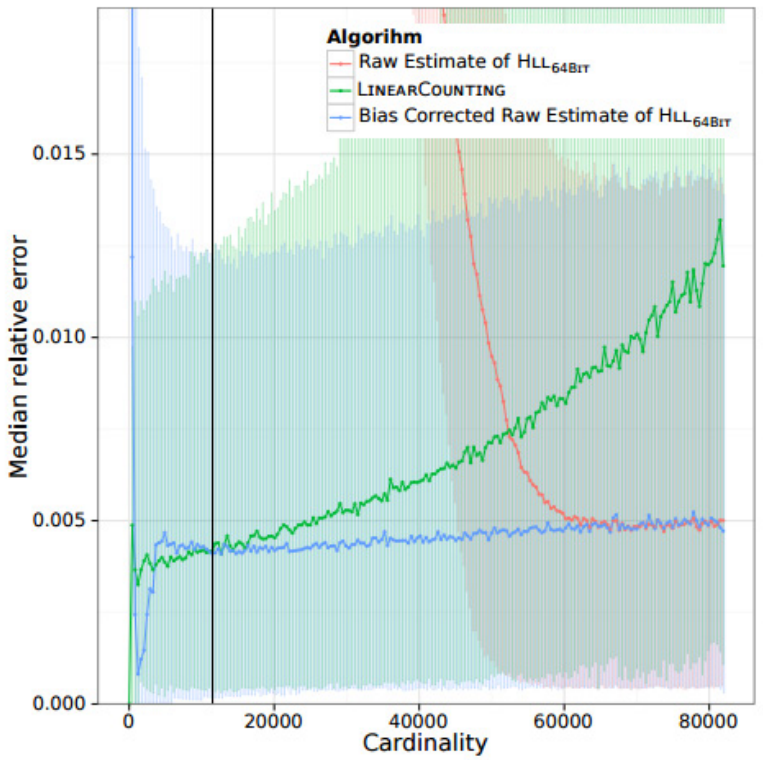


Figure 3: The median error of the raw estimate, the bias-corrected raw estimate, as well as LinearCounting for $p = 14$. Also shown are the 5% and 95% quantiles of the error. The measurements are based on 5000 data points per cardinality.

For n < 5m/2 nonlinear distortions appear that need to be corrected, so in this range linear counting (m log(m/V), where V is the number of zeros) is used instead.

Google improve the base HyperLogLog algorithm in three areas: ability to estimate very large cardinalities; improved accuracy for small cardinalities; and reduction in the amount of space required.

Estimating Large Cardinalities

A hash function of L bits can distinguish at most 2L values, and as the cardinality n approaches 2L hash collisions become more and more likely and accurate estimation becomes impossible. Google's solution here is rather straightforward – use a 64-bit hash function in place of a 32-bit one. We don't actually need to store the hashes themselves, just the maximum size of the number of leading zeros + 1, so 64-bit hashes only increases the memory requirement from 52p bits to 62p bits.

Improving Accuracy for Small Cardinalities

Recall that for n < 5m/2 the raw estimate of HyperLogLog is inaccurate, and linear counting is used instead. In the chart below, the red line shows the raw HLL estimate, and the green line shows the more accurate estimate produce by linear counting for small n.

The blue line on the above chart shows an improved estimate for small n that gives even better accuracy. This estimate is produced by directly correcting for bias in the original raw estimate.

In simulations, we noticed that most of the error of the raw estimate is due to bias; the algorithm overestimates the real cardinality for small sets. The bias is larger for smaller n, e.g., for n = 0 we already mentioned that the bias is about 0.7m. The statistical variability of the estimate, however, is small compared to the bias. Therefore, if we can correct for the bias, we can hope to get a better estimate, in particular for small cardinalities.

To determine the bias, the authors calculate the mean of all raw esti-

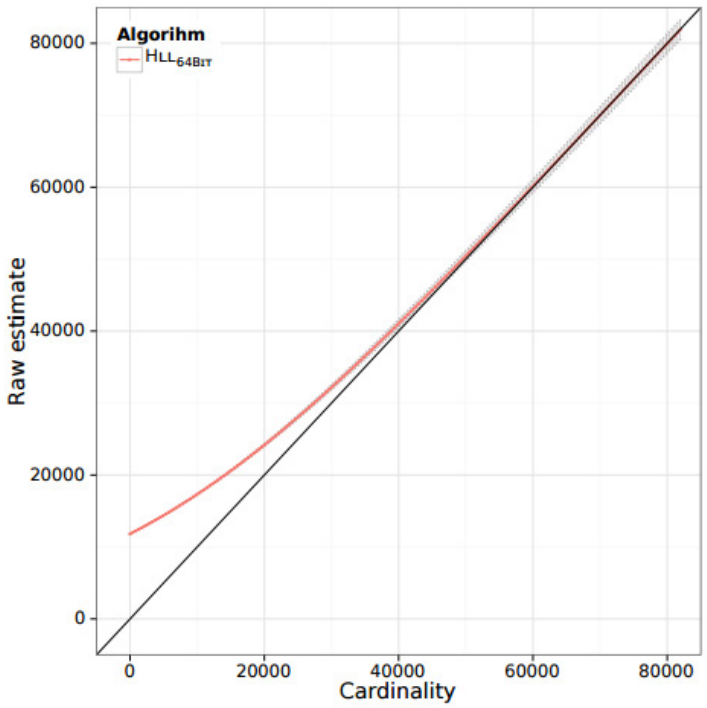


Figure 2: The average raw estimate of the HLL64Bit algorithm to illustrate the bias of this estimator for $p = 14$, as well as the 1% and 99% quantiles on 5000 randomly generated data sets per cardinality. Note that the quantiles and the median almost coincide for small cardinalities; the bias clearly dominates the variability in this range.

mates for a given cardinality, minus that cardinality.

With this data, for any given cardinality we can compute the observed bias and use it to correct the raw estimate. As the algorithm does not know the cardinality, we record for every cardinality the raw estimate as well as the bias so that the algorithm can use the raw estimate to look up the corresponding bias correction. To make this practical, we choose 200 cardinalities as interpolation points, for which we record the average raw estimate and bias. We use nearest neighbor interpolation to get the bias for a given raw estimate (for k = 6).

Notice in the first figure of this section, that linear counting still beats the bias-corrected estimate for small n. For precision 14 (p=14) the error curves intersect at around n = 11,500. Therefore linear counting is used below 11,500, and bias corrected raw estimates above.

The end result is that for an important range of cardinalities – roughly between 18,000 and 61,000 when p=14 , the error is much less than that of the original HyperLogLog.

Saving Space

In the original algorithm we maintain an array M of registers of size m = 2p. Each registers holds 1 + the maximum number of leading zeros seen so far for the corresponding hash prefix. If n << m then most of the registers are never used. In this case we can use a sparse representation that stores (index,register-value) pairs. If the list of pairs requires more memory than the dense representation of the registers (the array M), then the list can be converted to the dense representation.

In our implementation, we represent (index, register-value) pairs as a single integer by concatenating the bit patterns... Our implementation then maintains a sorted list of such integers. Furthermore, to enable quick insertion, a separate set is kept where new elements can be added quickly without keeping them sorted. Periodically, this temporary set is sorted and merged with the list.

When only the sparse representation is used, accuracy can be increased by choosing a different precision argument p' > p. If the sparse representation gets too large and needs to be converted to a dense representation, then it is possible to fall back to the lower precision p.

In the sparse implementation, we can use a more efficient representation for the list itself too...

We use a variable length encoding for integers that uses a variable number of bytes to represent integers, depending on their absolute value. Furthermore, we use a difference encoding, where we store the difference between successive elements in the list. That is, for a sorted list a1,a2, a3, ...

we would store a1, a2-a1, a3-a2, ... The values in such a list of differences have smaller absolute values, which makes the variable length encoding even more efficient.

The cutoff point between the sparse and dense representations is such that we know that the linear counting method will always be used to produce the cardinality estimation. This fact can be taken advantage of to yield a further space saving – see section 5.3.3 in the paper for details.

Comparison

The final Google algorithm with all the improvements is denoted HLL++. The following chart compares the accuracy of the original HLL algorithm with HLL++.

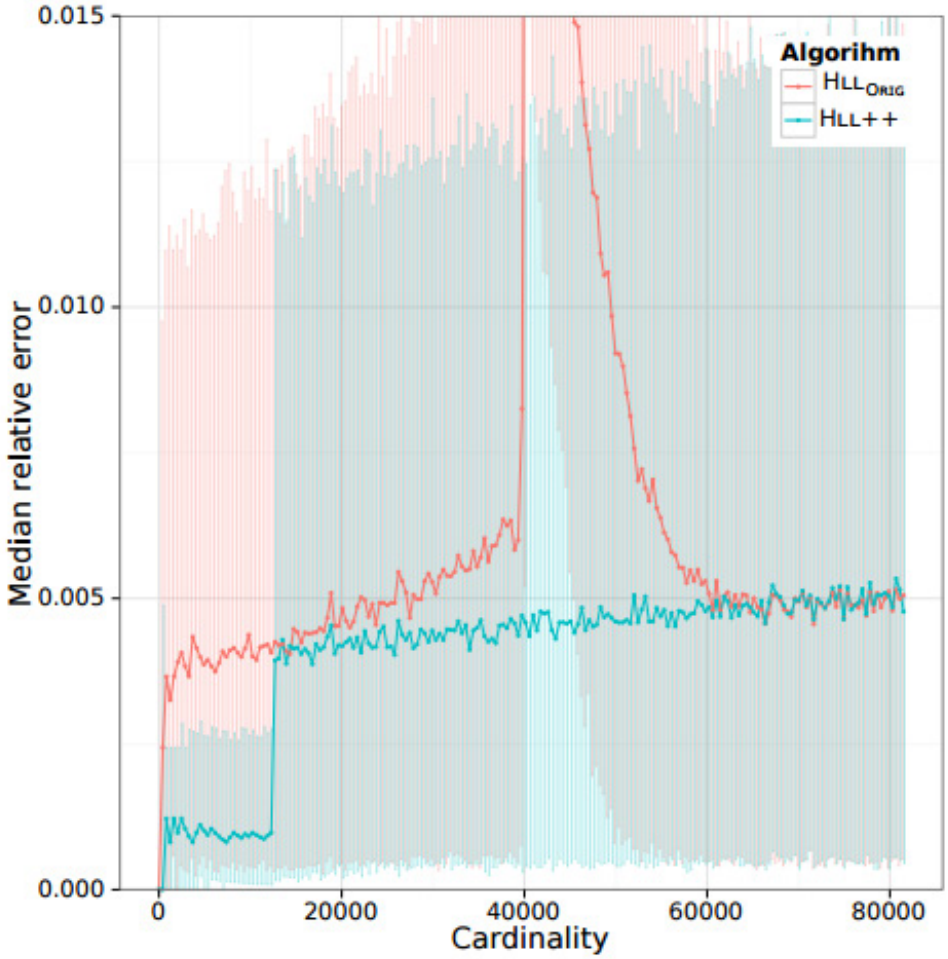


Figure 8: Comparison of HLL$_{ORIG}$ with HLL++. The mean relative error as well as the 5% and 95% quantiles are shown. The measurements are on 5000 data points per cardinality.

All of these changes can be implemented in a straight-forward way and we have done so for the PowerDrill system. We provide a complete list of our empirically determined parameters at http://goo.gl/iU8Ig to allow easier reproduction of our results.

DEEP DIVES ON HOT TOPICS

INFOQ EMAGS & MINI-BOOKS