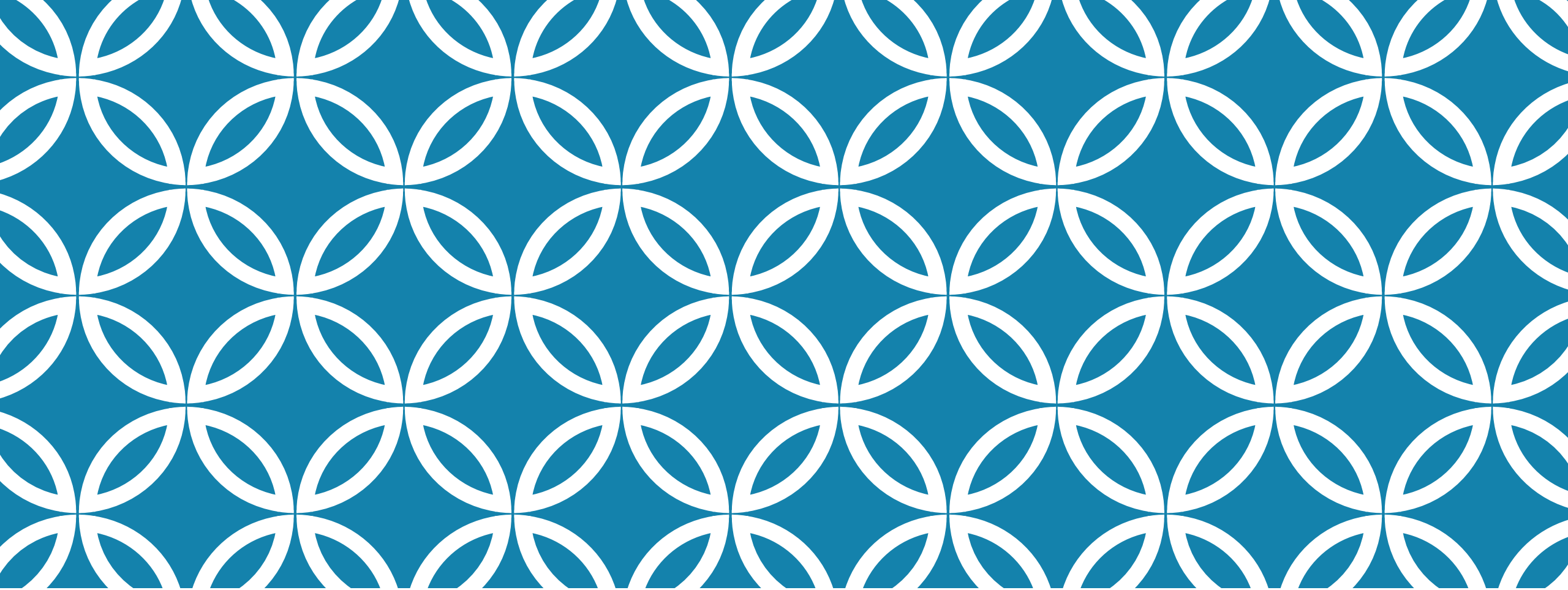
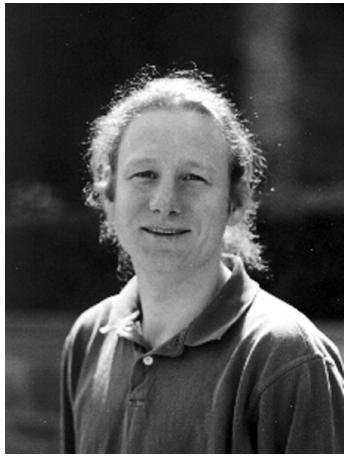


LEVERAGING RDMA

Professor Ken Birman

CS4414/5416 Lecture 27

A BRIEF HISTORY OF RDMA



… it all started here at Cornell!

In 1995, Werner Vogels (now CTO at Amazon) was working as a Cornell researcher and offered to help Thorsten Von Eicken (now a tech entrepreneur) and two students develop a new programming model for computer networking

CONTEXT: EVOLUTION OF NETWORK INTERFACE CARDS

When the U-Net project started, computer networks were based on standard ethernet and accessed through network adaptors that handled data transfers

Some of these NICs began to offer programmability: it was possible to replace the firmware on the card with new firmware coded in C. The firmware controlled DMA transfers between the network and host memory, just like for a disk controller

KEY IDEA ONE: GET THE KERNEL OFF THE PATH

The insight the U-Net team started with was that for many tasks, the kernel was more of a bottleneck than a helper

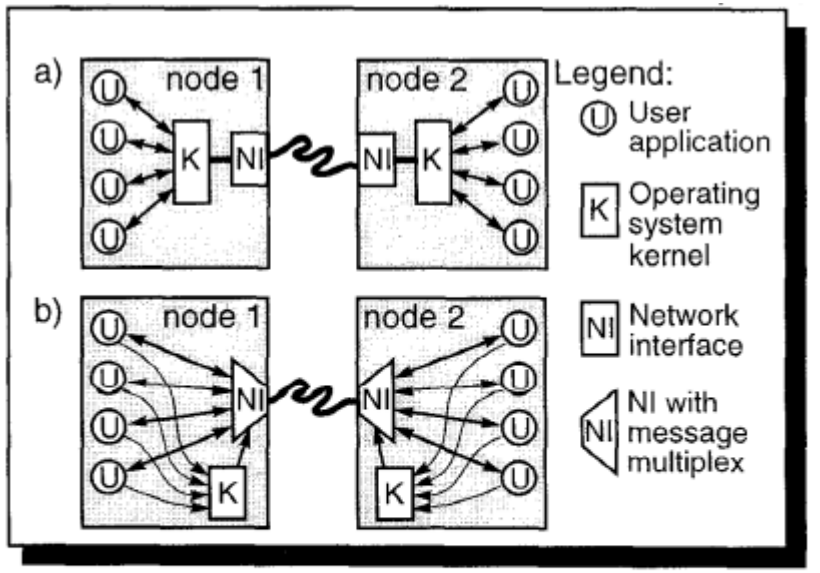
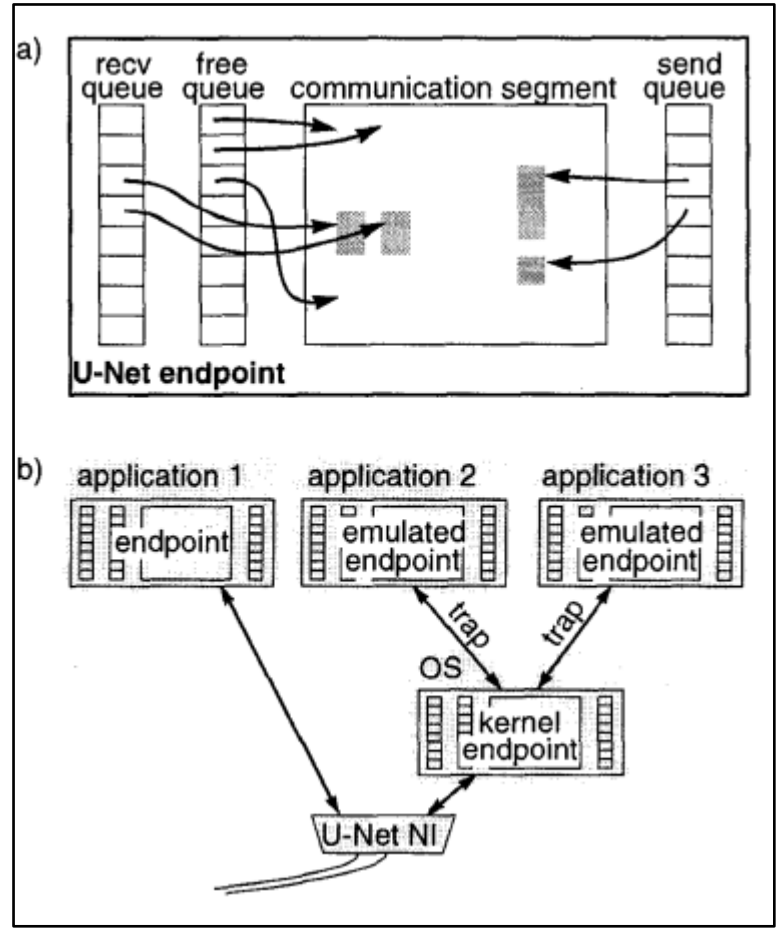


Figure 1: The traditional networking architecture (a) places the kernel in the path of all communication. The U-Net architecture (b) only uses a simple multiplexing/demultiplexing agent—that can be implemented in hardware—in the data communication path and uses the kernel only for set-up.

They argued that for very high speed networking of the kind used in HPC supercomputers, we should bypass the kernel

KEY IDEA TWO: MOVE DATA USING DMA



At very high speeds, interrupts seem slow too, so they proposed "queue pairs"

The U-Net NIC would monitor for send requests on the send queue. Seeing a send, U-Net ran a simple protocol to match against a pending receive request on the target machine, then would do the transfer and move the two completed requests to a free queue

KEY IDEA THREE: MODIFIED USER CODE

Rather than using the POSIX TCP connection model, U-Net favored a direct send/receive model.

The remote user would poll, watching for an incoming RPC request, then respond instantly

This required changing user code to build messages in memory regions "pinned and mapped" where the RDMA NIC could see the data (and where Linux wouldn't page it out)

IT WORKED!

U-Net was about 100x faster than standard TCP for simple RPC requests and ran on standard PC servers. It was far faster than the world's fastest parallel supercomputers.

A consortium was formed to standardize the concept and this led to modern Infiniband networks and RDMA

INFINIBAND VERSUS ETHERNET

Both run on the same optical networking layer.

But as we learned in Lecture 16, TCP has an approach in which senders send faster and faster until loss occurs: linear speedup but multiplicative backoff, yielding the famous saw-tooth

Infiniband uses a “credit” concept

INFINIBAND CREDIT

It works hop by hop. Consider one switch, router or NIC sending and another receiving.

The receiver advertises available memory capacity to the sender: “credit” to send.

The sender waits for credit and won’t ever overrun the receiver

ROUTED INFINIBAND?

It works the identical way, with credit hop by hop.

Ideally, devices don't need to store in memory before they forward… but they always have that option!

This turns out to drive loss rates to near zero in modern datacenter settings, similar to a NUMA computer memory bus.

RDMA IS A NEW U-NET BUT LAYERED OVER INFINIBAND

It uses the same credit-based scheme at the Infiniband layer

The end-to-end abstraction uses “bound queue-pairs”, just like U-Net: one for requests and the other for completions

Requests can be send, receive, and also some fancy ones: remote read and remote write

REMOTE READ AND WRITE

These require pre-granted permission and setup

But the idea is that one machine, call it R, grants permission for a second machine, call it S, to directly access its memory.

Now S can create some object in its own memory and read or write it directly into the memory of R.

COMPARISON?

Send and Receive are a lot like TCP operations, but using the U-Net model. The receive buffer needs to be large enough for the Send operation.

Remote read and write don't require any posted operation by the receiver (R in our example). S literally is writing directly into R's memory, or reading from it (the operation is like **memcpy** but over a network).

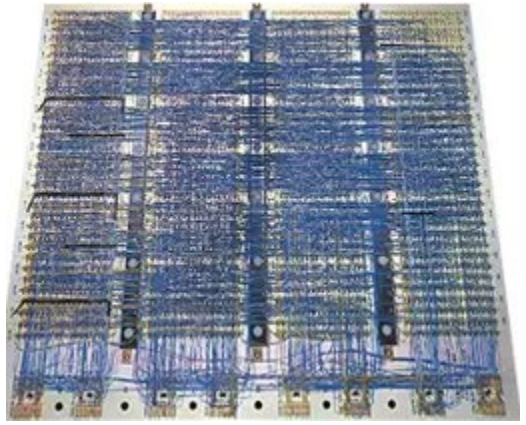
RDMA ON ETHERNET: RoCE

There was a lot of interest in using RDMA in cloud computing, but not in moving to Infiniband: too different/disruptive.

So a "converged" model was proposed, supporting both RDMA and Infiniband at the same time, sharing the ethernet

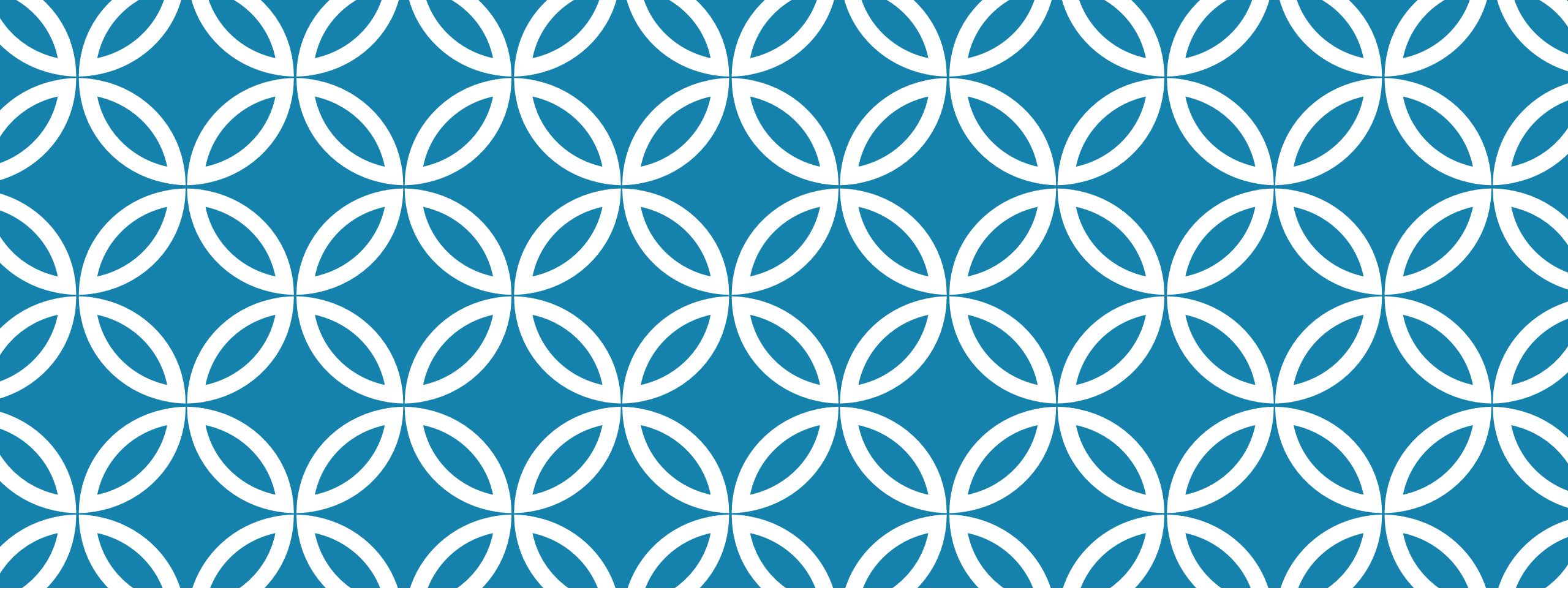
This is called RoCE. Pronounced "rocky".

TODAY?



All cloud computing systems offer HPC clusters with Infiniband "wiring" as well as standard Ethernet, and RDMA support on Infiniband.

Internally many cloud operators have RoCE deployments, but for their own use only. They limit it for use by infrastructure tools



MICROSOFT'S FARM KEY-VALUE STORE

Professor Ken Birman

CS4414 Lecture 25

IDEA MAP FOR THIS PART OF OUR LECTURE

Modern applications often work with big data

By definition, big data means "you can't fit it on your machine"
Reminder: Shared storage servers accessed over a network.

Reminder: Key Value Storage

Concept: Transactions. Applying this concept to a key-value store.

RDMA hardware accelerator for TCP and remote memory access. How FaRM leveraged this.

Ken spent a sabbatical at MSRC

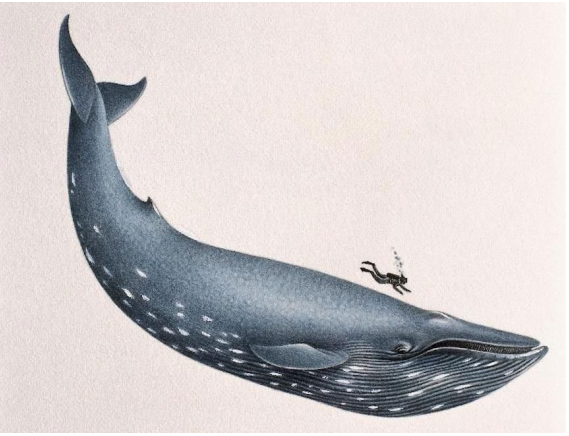
Kings College in Cambridge

MICROSOFT'S GOALS WITH FARM

In the Microsoft "bing" search engine, they had a need to store various kinds of objects that represent common searches and results. Most of these objects are small, but they have so many of them that in total, it represents a big-data use case.

The Microsoft research FaRM project (based in Cambridge England) was asked to help solve this problem.

BIG OBJECTS



When they do have big objects, they break them into smaller chunks (very small, in fact: 60 bytes plus a 4-byte header)

Farm is used via a **put**/**get** API, not with general transactions. But because of this issue of objects being large, we do have a form of transaction: one done to update a chunked object.

We'll see how they do this

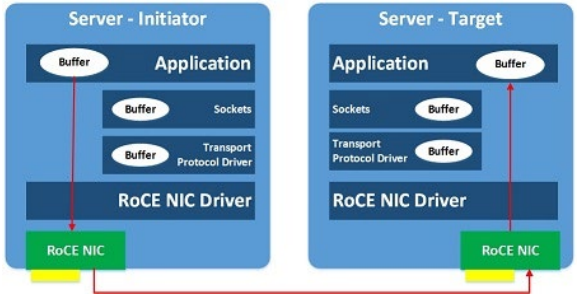
A FEW OBSERVATIONS THEY MADE

The felt they should try to leverage new kinds of hardware.

- The specific option that interested them was *remote direct memory access* networking, also called RDMA.
- RDMA makes the whole data center into a large NUMA system. All the memory on every machine can potentially be shared over the RDMA network and accessed from any other machine.

RDMA had never been used outside of supercomputers

THEIR APPROACH



Purchase a new hardware unit from Mellanox that runs RDMA on RoCE cables. Microsoft didn't want to use Infiniband.

Create a pool of FaRM servers, which would hold the storage. Stands for "Fast Remote Memory"

The servers don't really do very much work. The clients do the real work using RDMA for reading and writing.

… BUT COMPLICATIONS ENSUED

A "rocky" road!

The hardware didn't work very well, at first.

- Getting RDMA to work on normal ethernet was unexpectedly hard. RoCE is pronounced "rocky", perhaps for this reason.
- Solving the problem involved major hardware upgrades to the datacenter routers and switches, which now have to carry both RDMA and normal TCP/IP packets.

It cost millions of dollars, but now Microsoft has RDMA everywhere.

IT IS EASY TO LOSE THE BENEFIT OF RDMA

One idea was to build a protocol like GRPC over RDMA.

When Microsoft's FaRM people tried this, it added too much overhead. RDMA lost its advantage.

To leverage RDMA, we want server S to say to its client, A, "you may read and write directly in my memory". Then A can just put data into S's memory, or read it out.

DUE TO INTEREST FROM USERS, THEY ADDED A KEY-VALUE "DISTRIBUTED HASH TABLE"

The Bing developers requested a **put**/**get** model, which seems like a match with RDMA except for the object sizes involved.

They didn't request transactions on multiple distinct objects (multiple keys), but they do need **put** and **get** atomicity.

THEY DECIDED TO IMPLEMENT A FAST SHARED MEMORY

The plan was to use the direct-memory access form of RDMA.

- **Write(addr,object)** by A on server S would just reach into the memory of S and write the object there.
- **Read(addr)** would reach out and fetch the object.

A FaRM **address** is a pair: 32-bits to identify the server, and 32-bits giving the offset to a cche line inside its memory (64 bytes).

HOW DO WE KNOW WHERE THESE CHUNKS LIVE?

Farm has a meta-data service, like the one in Ceph.

The object key lets us look up the list of servers and cache lines where the small chunks are hosted, including the address of each in that server's memory.

To select this address Farm uses a form of O(1) search for free cache line cells in a bit vector of "in use" bits, one for each server.

SO…

When an object is first written, the creator process asks the metadata service to find space for it. Then the first write and subsequent reads will use this list of servers and cache lines.

If an object changes size, the previous meta data would have to be deleted and then a new meta data record can be created. (Probably FaRM then notifies any process that knew the old mapping in case it cached it for reuse.)

… THEY ALSO HANDLE COMPLEX OBJECTS

This approach can handle complicated objects, like a JSON file with many fields.

We simply serialize the object as a byte vector.

So in this lecture, we focus on chunks, but arbitrary KV objects can be hosted in FaRM.

A PROBLEM ARISES!

What if two processes on different machines access the same data? One might be updating it when the other is reading it.

Or they might both try to update the object at the same time.

These issues are rare, but we can't risk buggy behavior. FaRM needs a form of critical section.

… MORE PROBLEMS

They also will turn out to have some (infrequent) cases where A and B end up writing to the same locations

They want to avoid “A smashes B” scenarios if the objects are in fact not the same

LOCKING OR MONITORS?

In a C++ process with multiple threads, we use mutex locks and monitors for cases like these.

But in FaRM the processes are on different machines.

Distributed locking is just too expensive. They do support it, but decided to avoid needing to use it.

DOWN TO BASICS: BING DOES NEED ATOMICITY

We learned about C++ atomics. Atomicity can be defined for a set of updates, too:

- We need them to occur in an all or nothing manner. [**ordering**]
- If two threads try to update the same thing, one should run before the other (and finish) before the other can run. [**isolation**]
- Data shouldn't be lost if something crashes. [**durability**]

CONCEPT: A TRANSACTIONAL WRITE

We say that an operation is an *atomic transaction* if it combines a series of reads and updates into a single indivisible action.

The transaction has multiple steps (the individual reads and writes, or get and puts). Software creates an illusion that they occur all at once.

Readers always see the system as if no updates were underway. Updates seem to occur one by one.

FARM TRANSACTIONS

They decided to support two kinds of transactions

1. An atomic read that will read a series of key-value pairs all as a single atomic operation.

2. An atomic update that will replace a series of key-value pairs with a new set of key-value pairs.

FARM VARIANT

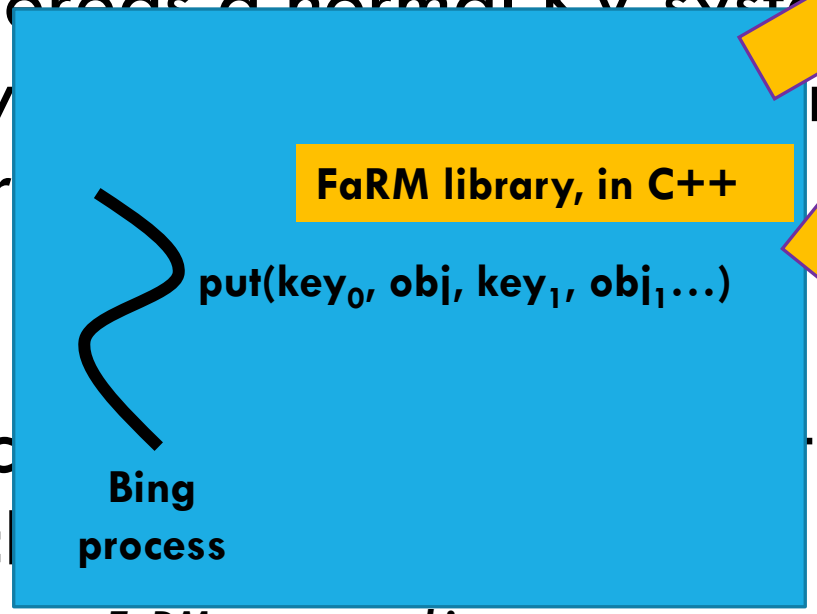
Whereas a normal KV system needs to find its data in each server each time, and requires software help for this, FaRM can short circuit that.

Once an object is placed, it remembers the location of each of its chunk, so it can just read or write chunks directly.

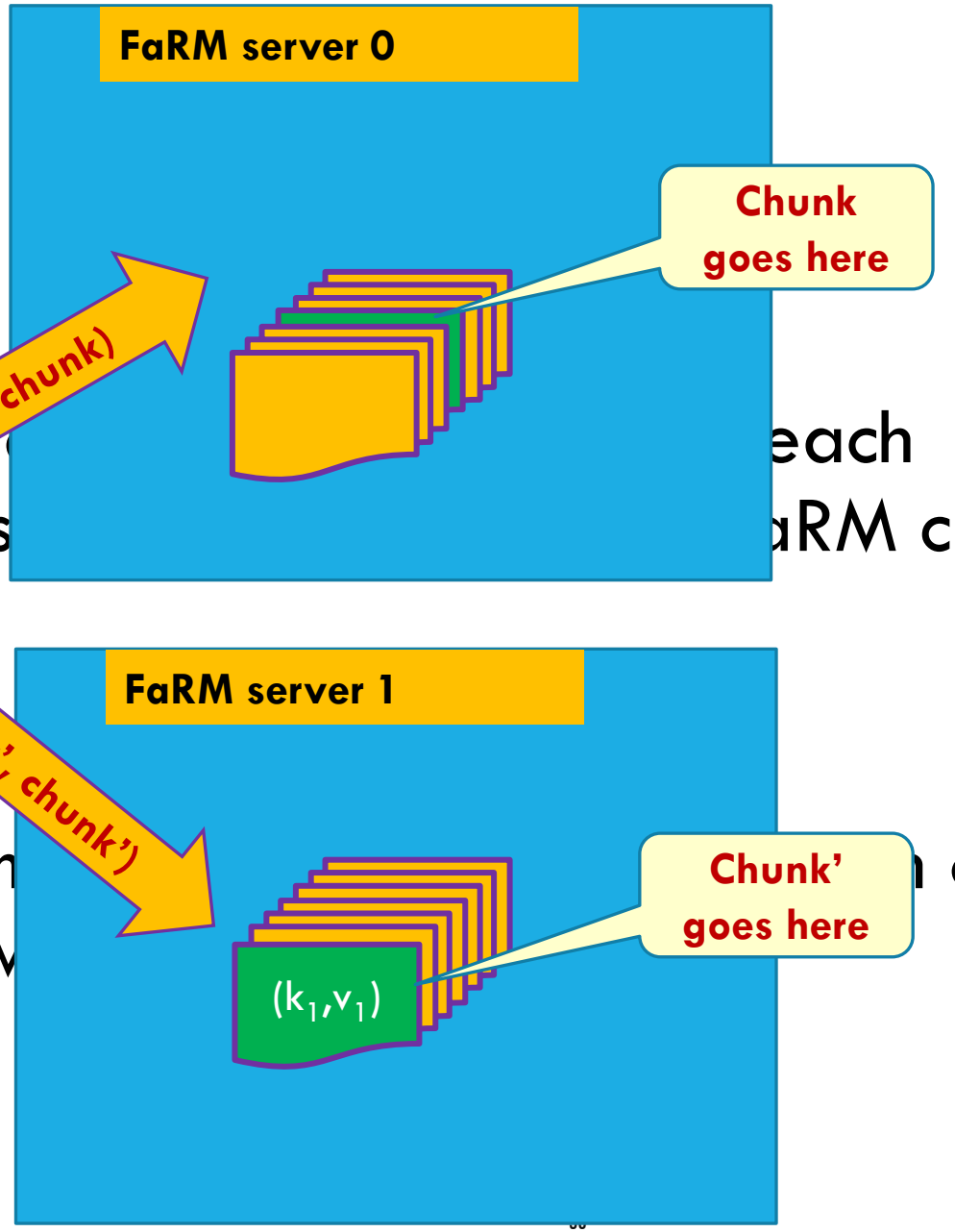
FARM VARIANT

Whereas a normal KV syste… each serv… res s… aRM can shor…

Onc… rem… of its cl… or w…



FaRM on my machine



COMPLICATING FACTORS THEY CONSIDERED

Lots of processes running concurrently that share the FaRM

Distributed locking is complex and slow, so they are only using it for transactions that need strong atomicity.

They wanted to optimize FaRM for their expected workload

HOW THEY SOLVED THIS

Microsoft came up with a way to write large objects without needing locks.

They also found a hashed data structure that has very limited needs for locking.

FIRST, THE TRANSACTION MODEL

Let's first deal with the "many updates done as an atomic transaction" aspect.

Then we will worry about how multiple machines can safely write into a server concurrently without messing things up.

THE TRANSACTIONAL WRITE IDEA

Suppose that some object requires *k* updates. FaRM breaks the data into chunks of size 60 bytes, adds a 4-byte "version id" to each update, in a hidden field (a "header"), obtaining 64-byte records.

version update_0 ***version*** update_1 …. ***version*** update_k

64 bytes 64 bytes…

In FaRM the version number is a hash of who did the write and the transaction id. The end user will never see this number.

THE TRANSACTION WRITE IDEA, CONTINUED

The basic algorithm FaRM uses is this:

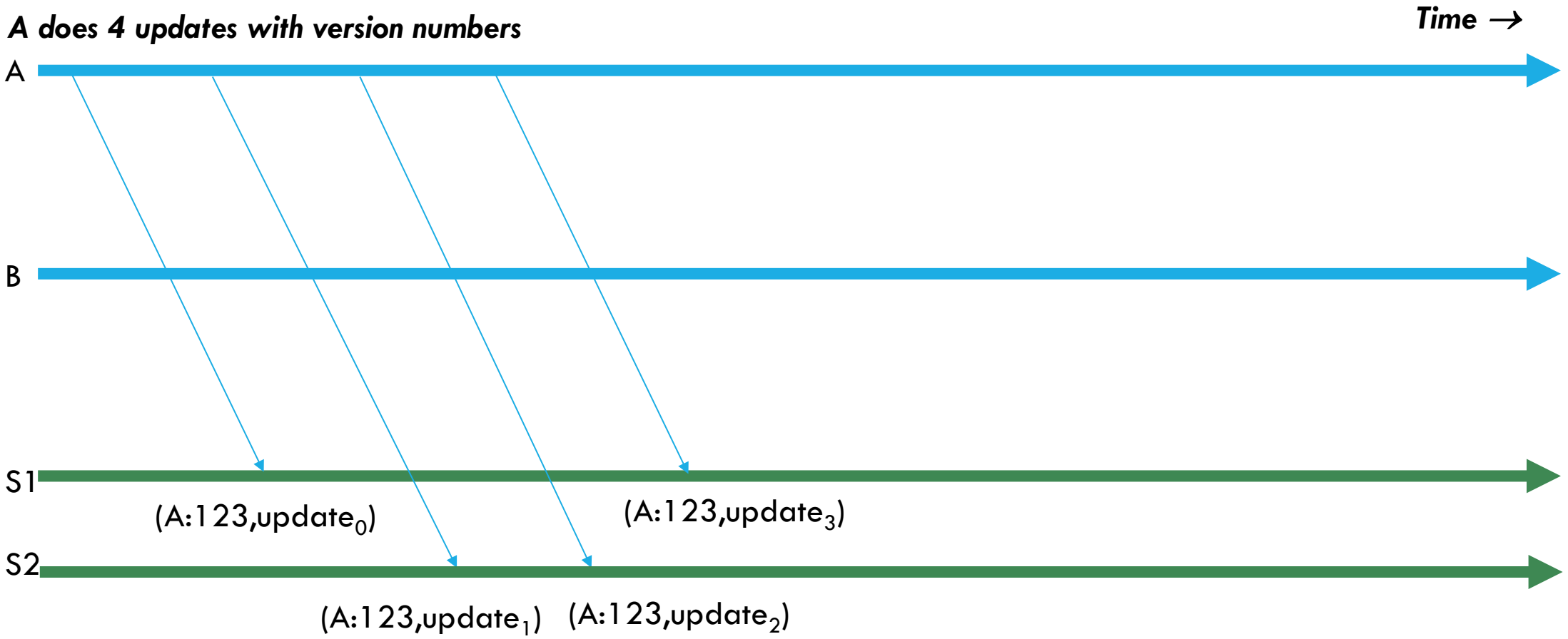
To write X, A first tags each update with a version number

Now it writes all its updates.

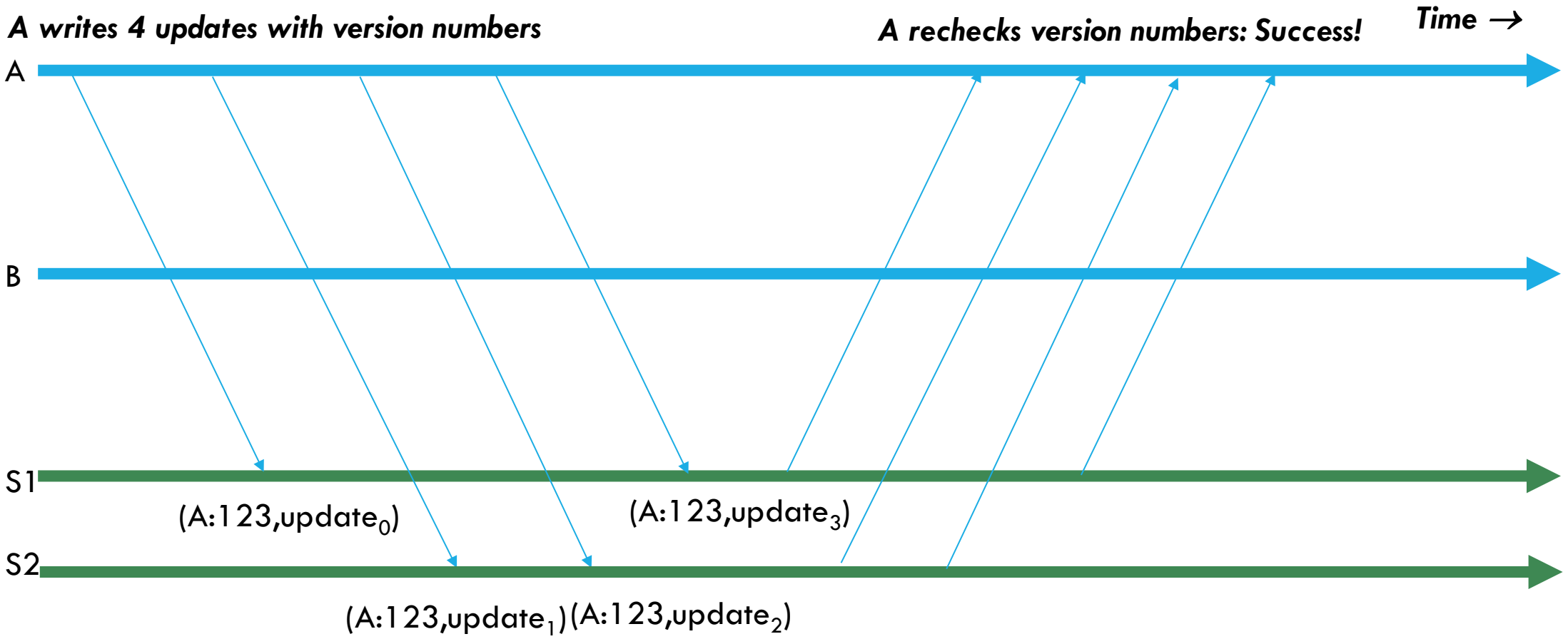
Then A goes back to check that the version numbers at the end are the same as the version numbers it wrote

Same if some process reads the data

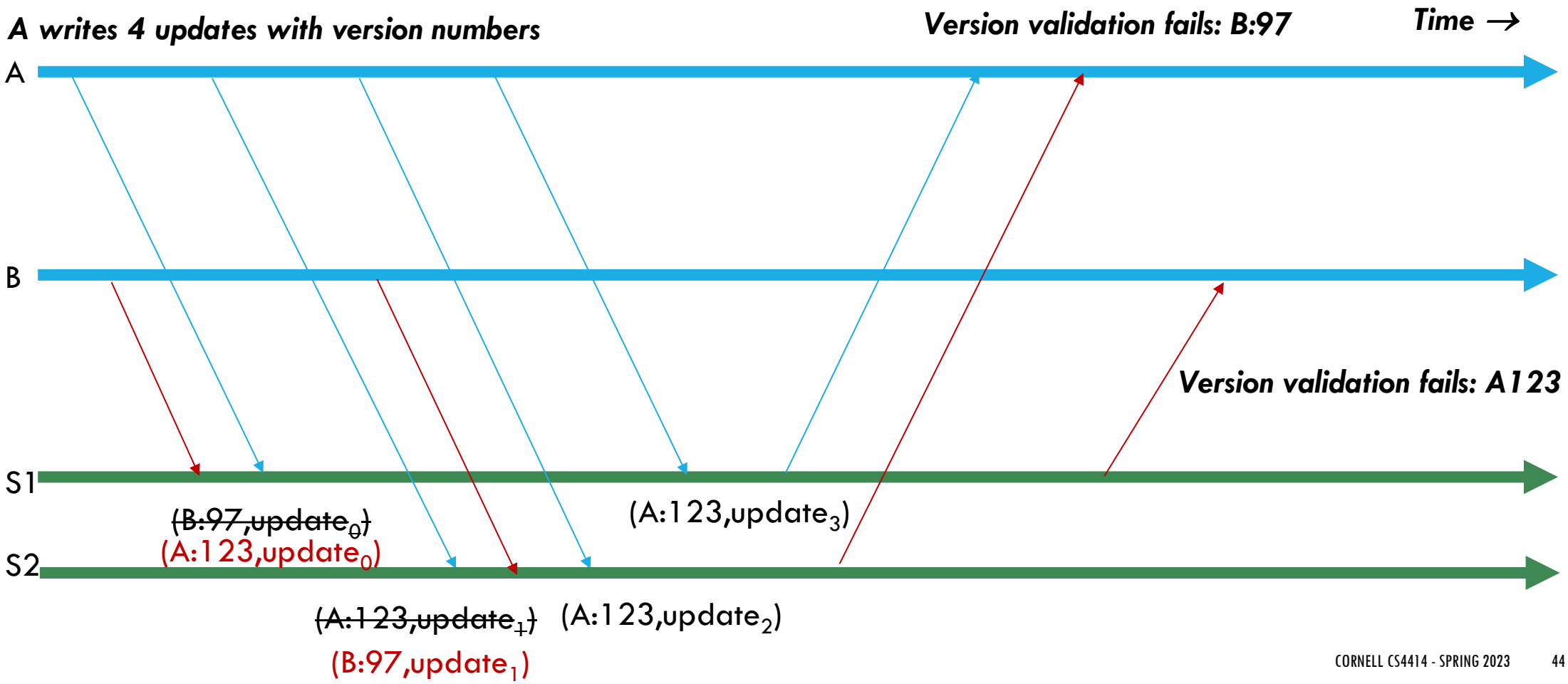
A TRANSACTIONAL WRITE



A TRANSACTIONAL WRITE: NORMAL CASE



A TRANSACTIONAL WRITE: CONFLICT CASE



BASIC RULE

If the version numbers all match, the object is intact.

If the version numbers don't match, something disrupted A's read or write, and it must retry the same request.

WHY COULD THEY FAIL TO MATCH?

If A and B write at the same time, they use different version numbers. There are three possible cases:

1. A writes first, then B. Version numbers match but are the ones B wrote.
2. B writes first, then A. Same, but now we see A's versions.
3. They overlapped, leaving some unmatched version numbers.

IF A OR B IS "SURPRISED" BY THE RESULT

Pick a random small number, delay by this amount of time.

They will pause by different amounts: A "random backoff". Perhaps, A doesn't pause at all, but B waits 2us

FaRM should rarely see conflict. When a conflict does occur, A retries instantly. B pauses, then later will wake up and do its write.

WHY ISN'T LOCKING NEEDED?

FaRM does not require any locks for reads or writes provided that conflicts occur rarely.

This is because the validation step will usually succeed, and the backoff step will rarely even need to run.

But there are a few objects where conflicts are more common.

LOCKING

For heavily contended-for objects, or for cases where an overwrite should not be allowed, they implemented per-key locking using a new RDMA test-and-set feature.

These “lock objects” allow Bing to get true mutual exclusion if genuinely needed, but Bing developers were urged to use them very rarely – they harm performance otherwise.

HOW A UPDATES SOME RECORD IN S

A prepares the 64-byte object.

Now A figures out which server to talk to: a first hashing step.

And then A figures out which array element to access: a *second* hash and modulus operation, modulo the size of the table in S. A uses RDMA to read or write directly into this memory region.

WHAT IF A AND B "COLLIDE" AT THIS STEP?

For a single 64-byte record, RDMA itself handles atomicity.

Either A will "win" and go first, and B will run second, or vice versa. The hardware does it and FaRM has no need for any kind of special logic.

If the update is part of a multi-write transaction, the transaction logic we saw on slides 32-25 will resolve any conflicts.

HASH COLLISIONS

Another puzzle is this.

Suppose that A and B are accessing S *using different keys*. The objects are different. No conflict has arisen here.

Yet those keys could hash to the identical slot in memory. This is called a *hash collision*. We don't want the objects to overwrite one-another: we need a way to keep both!

HOPSCOTCH HASHING

Hopscotch hashing is done by the metadata service when FaRM initially places an object, after it knows which storage servers will hold the chunks of a particular (key,update) pair.

FaRM hashes to find the slot an item should go into, but first reads the slot to see if it is full. It asks: are we updating a value or inserting some other key?

Replacement can occur in place. To insert a different key,FaRM scans the region "around" the slot. The new item goes into the closest empty slot.

HOW TO DO A READ OR WRITE?

Once FaRM decides where the chunks will live, it doesn't have to repeat the hopscotch search again.

By keeping that information in the metadata service, any user of that object can find the chunks.

This is really exactly like Ceph and its way of striping data over the storage server layer.

OUTCOME?

Now A and B can treat the pool of storage servers as a NUMA memory unit. In fact keys are like memory addresses.

S helps with RDMA setup, but then A and B can read/write concurrently without help from S.

The algorithm is lock-free except for high-contention objects. Those use the mutex locks.

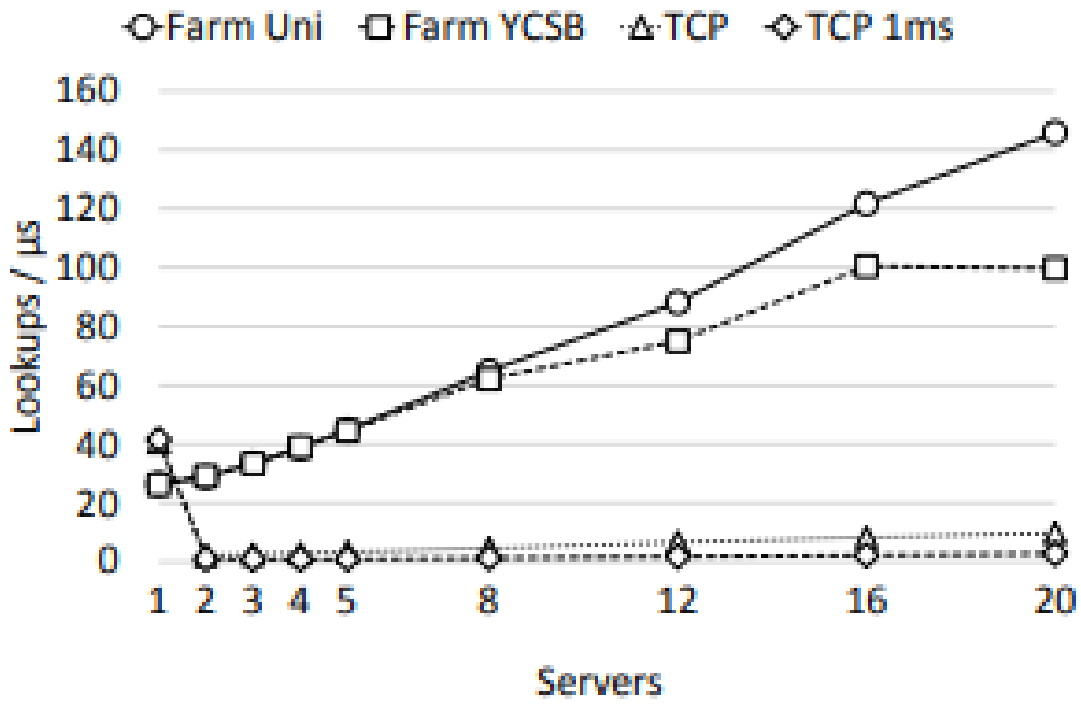
FARM PERFORMANCE: LOOKUP RATE

FaRM Uni: FaRM with 120M (k,v) pairs, accessed uniformly at random.

FaRM YCSB: Items accessed according to the YCSB benchmark popularity pattern.

TCP: FaRM running over a hand-tuned TCP-based protocol, similar to GRPC

TCP 1ms: FaRM on TCP with a 1ms response time guarantee



https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-dragojevic.pdf

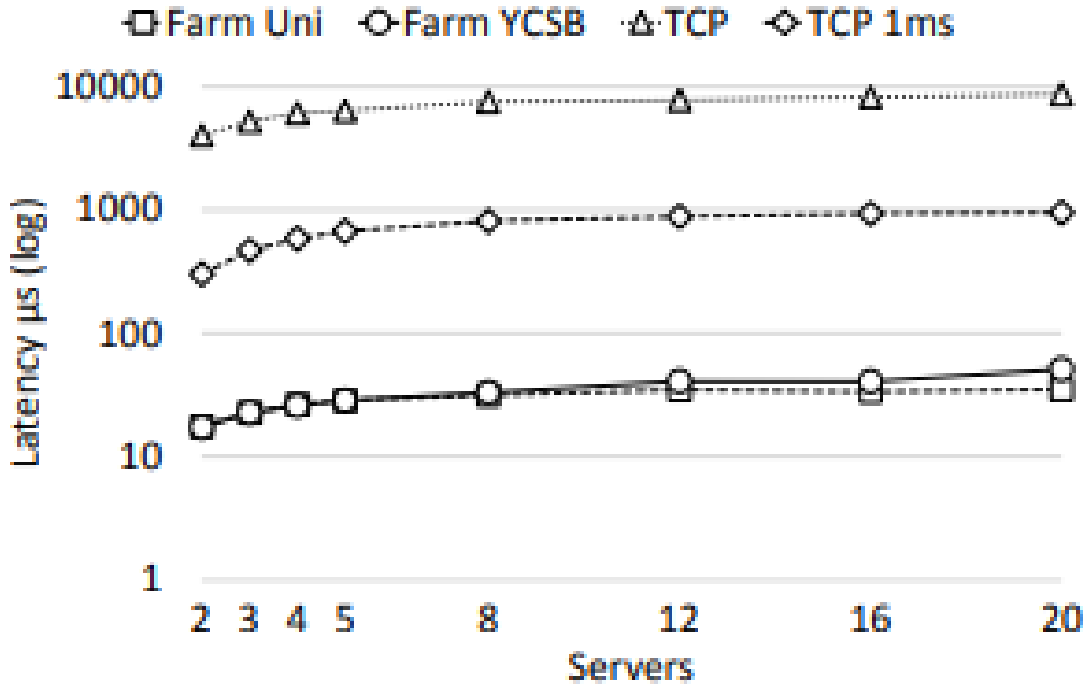
FARM PERFORMANCE: DELAY TO DO LOOKUP

FaRM Uni: FaRM with 120M (k,v) pairs, accessed uniformly at random.

FaRM YCSB: Items accessed according to the YCSB benchmark popularity pattern.

TCP: FaRM running over a hand-tuned TCP-based protocol, similar to GRPC

TCP 1ms: FaRM on TCP with a 1ms response time guarantee



FARM PERFORMANCE: TRANSACTIONAL READ

FaRM Uni: FaRM with 120M (k,v) pairs, accessed uniformly at random.

FaRM YCSB: Items accessed according to the YCSB benchmark popularity pattern.

TCP: FaRM running over a hand-tuned TCP-based protocol, similar to GRPC

TCP 1ms: FaRM on TCP with a 1ms response time guarantee

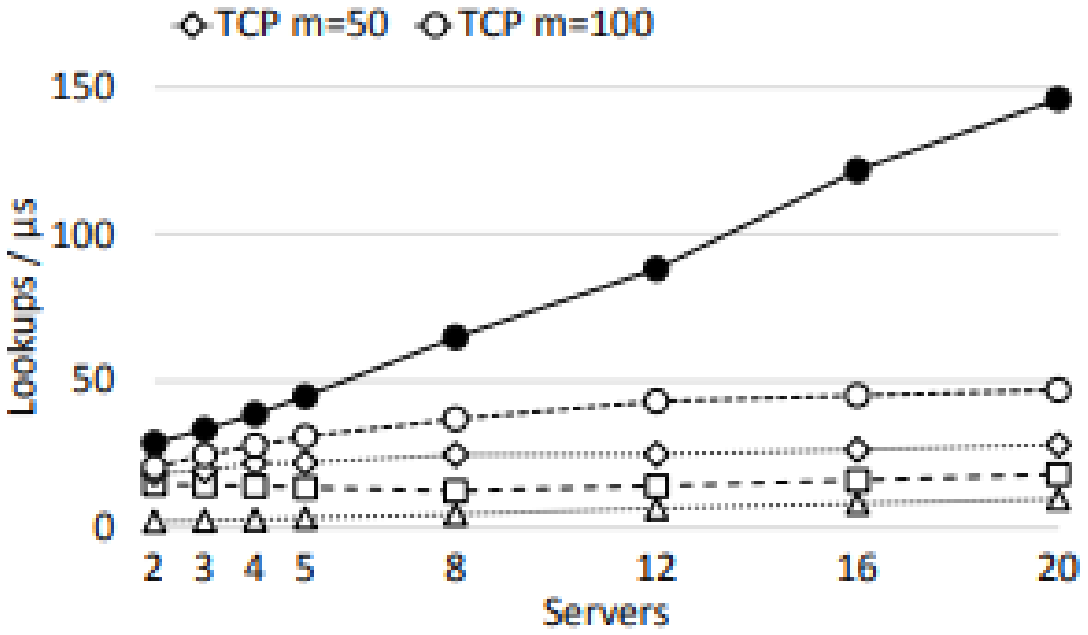


Figure 13: Key-value store: multi-get scalability

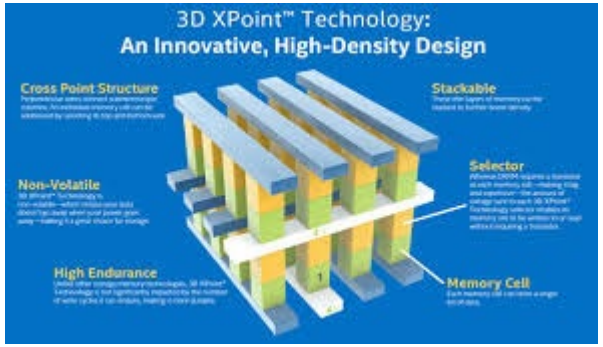
BACKUP

Bing also needed some form of backup.

Objects can be lost, rarely, but for some special objects Bing wants the data to be there after a server crashes, then restarts.

For these cases, Microsoft decided to build an ultra fast write-ahead log. Speed matters because it could slow down updates.

3-D XPOINT MEMORY



Normal memory will be lost in the event of a crash. Replication only helps if the backup doesn't crash, too. Some Bing uses need more.

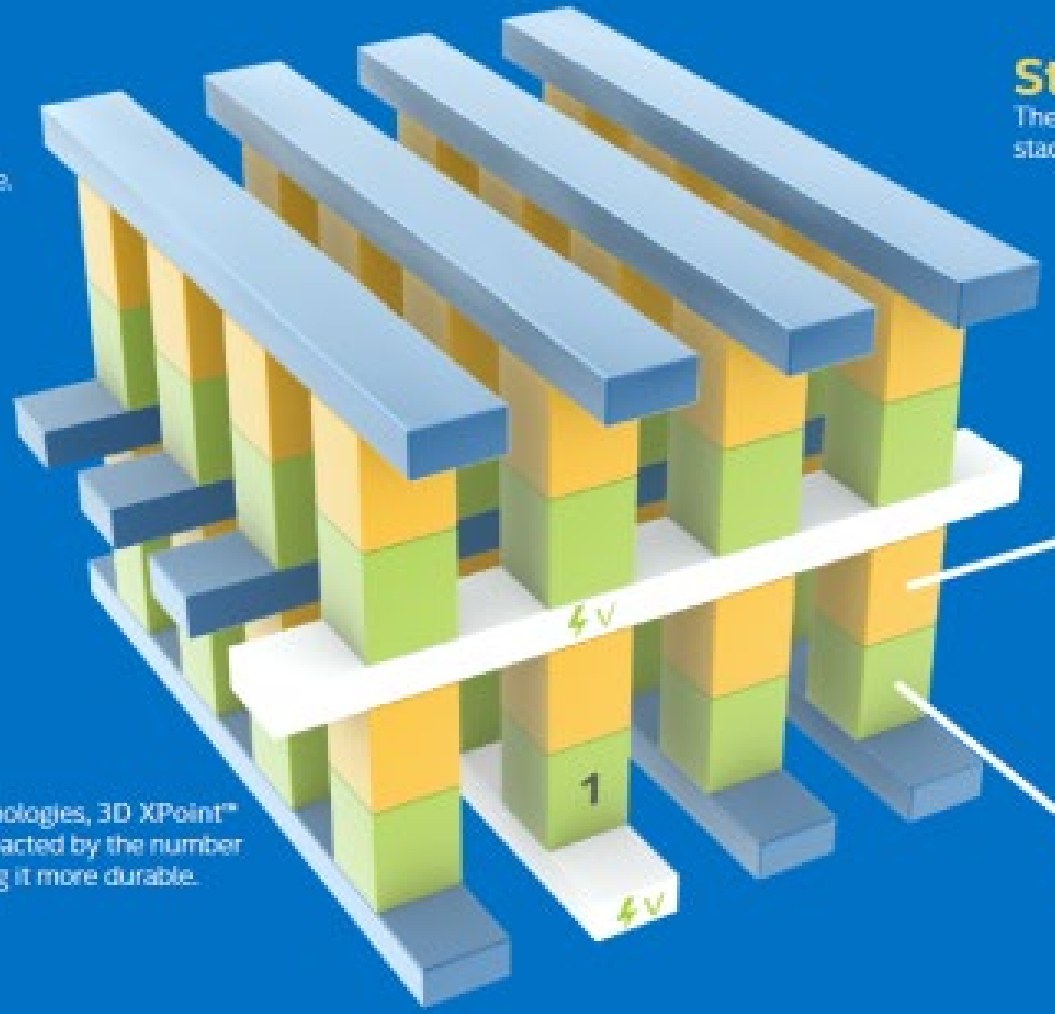
The FaRM project decided to experiment with a new form of memory called 3-D XPoint, based on phase-change memory hardware (a novel layered semiconductor technology)

The idea was that anything in the FaRM memory would survive crashes and just be "in memory" on restart!

3D XPoint™ Technology:
An Innovative, High-Density Design

Cross Point Structure
Perpendicular wires connect submicroscopic columns. An individual memory cell can be addressed by selecting its top and bottom wire.

Stackable
These thin layers of memory can be stacked to further boost density.



Non-Volatile
3D XPoint™ Technology is non-volatile—which means your data doesn't go away when your power goes away—making it a great choice for storage.

Selector
Whereas DRAM requires a transistor at each memory cell—making it big and expensive—the amount of voltage sent to each 3D XPoint™ Technology selector enables its memory cell to be written to or read without requiring a transistor.

High Endurance
Unlike other storage memory technologies, 3D XPoint™ Technology is not significantly impacted by the number of write cycles it can endure, making it more durable.

Memory Cell
Each memory cell can store a single bit of data.

… NO LUCK

3-D XPoint took years longer than expected to reach the market.

Sample units were much slower than expected when used as memory, although they were impressive as disks. (Slow DMA transfer rates)

Even today, this technology can't just be used like normal memory!

FALLBACK: WRITE-BEFORE LOG ON SOME FORM OF DISK

FaRM was forced to use a fallback: storage drives based on a fast form of flash memory (similar to a USB)

The FaRM servers watch for updated portions of the key-value store and log them to the server in a continuous stream.

FALLBACK: WRITE-BEFORE LOGGING

For sensitive data that cannot be lost, FaRM has a way to pause the writer to wait until the log is written to disk.

Called “write-before logging”: First log the update, then finalize it and allow readers to see the data.

This way, a writer can be sure the data won’t be lost.

IT WAS TOO SLOW!

250TB for only $500,000…

NAND storage speeds are surprisingly variable: some operations are fast, some slow.

Microsoft realized that write-ahead logging would emerge as a bottleneck if they didn't find a solution to this.

THEY ADDED TWO OPTIONS

Some Bing uses only need high availability, not persistence. For these, they replicated FaRM.

Data is still held purely in memory, but now there is a copy in an active replica server: two copies of each item. If that machine doesn't depend on the same hardware in any way, it shouldn't crash even if the primary copy goes down.

BATTERY-BACKED CACHE FOR FLASH DRIVES

A second option was to use a NAND storage unit that has a battery-backed RAM cache in front of the flash-memory storage.

A DMA write first goes into the battery-backed memory: a very fast transfer. Now the data is "safe".

The write to flash memory occurs soon after, but no need to pause unless the RAM memory cache fills up. Even if power is lost, the battery-backup will allow the device to finish the writes.

THIS IDEA IS OPTIMIZED FOR LOAD BURSTS

With a steady rate of persisted writes, the solution would saturate when the DRAM buffer fills up and would be as slow as the flash memory.

But FaRM workloads are bursts and writes that need logging are rare. So there is time to persist the write ahead records in the pauses between requests to the unit.

We end up with fault tolerance but RDMA memory speeds!

BING REALLY USES FARM. COPILOT WILL BE USING IT TOO (OR ALREADY IS).

Not every "academic research" project has impact on big corporate products like Bing. FaRM is unusually impactful.

But normal Azure users can't access FaRM (yet) and RDMA is used only by special services like FaRM – not you and me!

NOTICE THE SIMILARITY TO C++ IDEAS IN A SINGLE COMPUTER!

Many of these same concepts are relevant to C++ with threads.

In fact, this is deliberate. The FaRM team members are C++ developers and wanted FaRM to feel natural.

People always prefer the same ideas in new settings… not new ideas, unless there is no choice!

SUMMARY

Modern applications often run into big-data uses

Microsoft created FaRM as an RDMA-enabled shared memory. Later it evolved for Bing: the developers preferred a key-value model.

The extreme efficiency of FaRM comes from the mapping to RDMA hardware, and inspired Derecho, which we heard about in lecture 14. But getting the full benefit of this cutting-edge hardware was hard!