Transactions: From Local Atomicity to Atomicity in the Cloud

David Lomet

Microsoft Research
Redmond, WA 98052 USA

Abstract. Atomicity is a fundamental concept in computer science. Initially it was defined as an abstraction to be used in a local context. But over time, its use has expanded or scaled as application programmers have come to rely on it. This reliance is based on atomicity's ability to make concurrent systems understandable and applications much simpler to program. Atomicity has multiple origins, but it can be fairly said that Brian Randell's Reliability Project at the University of Newcastle in the 1970's played a significant early role in defining the atomicity abstraction and building an early prototype to realize it. This paper starts by sketching the Newcastle contribution and goes on to explore how atomicity has been stretched to deal with clusters of processors. The challenge today is to deal well with systems of vast scale, as exemplified by the enormous data centers of current cloud services providers. We sketch a new and promising approach to this challenge.

1 Atomicity and Transactions: A Personal Perspective

The atomic action (or serializable transaction) abstraction has multiple roots. Among those roots was work that was outside of the database area that is usually thought of when thinking about transactions (the database folks coined the term "transaction" for this abstraction and connected it with earlier work, e.g. IMS program isolation). The reliable computing (fault tolerance) world also played a key role in the emergence of atomicity as a way of building and understanding systems. And no place played a larger role in this than the University of Newcastle and the Reliability Project of Brian Randell.

1.1 ACID Properties

Serializable transactions are the gold standard for controlling and understanding concurrent execution [12]. The properties of these transactions make systems as well as user application programs understandable. As well, using serializable transactions in situations where concurrent activity is in progress makes the programming chore substantially easier. The reason for this can be stated in a number of ways.

In my early work, I characterized atomic actions (serializable transactions) as presenting to users "a consistent view of the data, i.e. one in which each of them appears to be the sole user of the system" [14]. Thus, within the boundaries of an atomic action, if an "action" is correct when executed serially or in isolation, it will be correct when executed in the presence of concurrent activity.

Subsequently, Haerder and Reuter [10] characterized transactions via the "ACID" acronym, where the letters of "ACID" characterized the properties of transactions. We list the ACID properties of serializable transactions in Table 1. That characterization is useful as it teases apart properties that have subsequently been seen to be separable, but that atomic actions (serializable transactions) possess in their entirety.

While ACID properties were described after the fact, they can also serve as a framework for understanding the early evolution of the atomicity concept. Unfortunately, the "A" in "ACID" is used to denote what is called the "atomic" property, and as a result is overloaded. In our discussion of ACID, we will use the phrase "All or nothing" for the "A" of ACID, and indeed, that captures the intent more precisely.

Table 1. ACID properties of serializable transactions

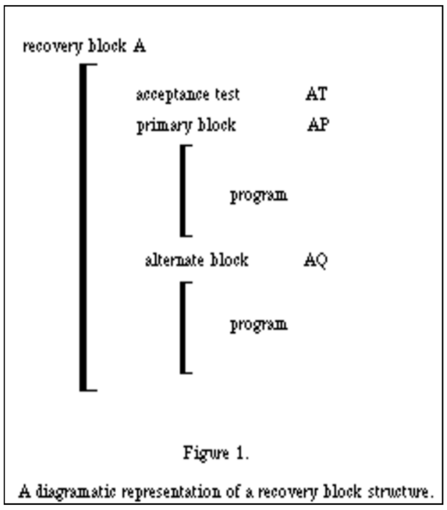
A	All or nothing
C	Consistent
I	Isolated
D	Durable

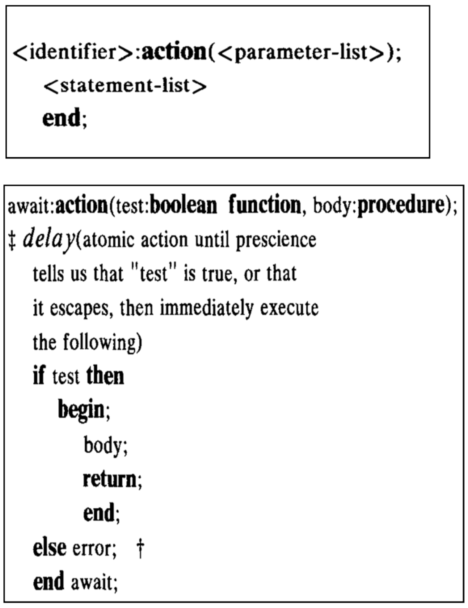
I would like to make a couple of comments about ACID transactions. First, the ACID properties are separable. One can have the all-or-nothing "A" without having the "I" of isolation. One can have both "A" and "I" without having durability (the "D"). Indeed, while durability is essential, for transactions that are provided to database application programmers, in the case of system level activity, durability may not serve a useful purpose. Further, consistency (the "C") is actually provided by the programmer using the atomic abstraction. The system supporting the atomic action abstraction provides the "AID" properties.

1.2 Architectural Approach to Reliability

Fundamental to the role that the Newcastle Reliability Project played in the development of atomicity was Brian Randell's vision of reliability as resulting from the use of system architectural mechanisms [18]. This was in stark contrast to the view of reliability as a consequence of the writing of correct programs. While few would dispute that reliability increases as the bug count drops, in the real world, large systems and large applications are never "correct". Indeed, it is usually impossible to fully state the correctness conditions, much less provide a proof that the system meets them. Further, hardware fails from time to time, so even perfect programs would not prevent system failures. What was needed, and this is Brian Randell's key insight, is an architectural mechanism that can be used to cope with the inevitable failures that will always occur. Brian was pursuing this “fault tolerant systems” approach at the emergence of fault tolerant systems as a separate technical area.

As part of the Reliability Project work, the Newcastle team had invented a notion called “Recovery Blocks”. A recovery block [11] is illustrated in Figure 1 (a), taken from the original paper. A recovery block permitted a program, having failed in its execution of a block of code, to return cleanly to an earlier state, and try again, perhaps executing a different program block. Tom Anderson and Ron Kerr at Newcastle implemented recovery blocks using what they called a “recovery cache” [3], a mechanism that incrementally saved program state as of entry to the recovery block and would restore it should a failure occur. Recovery blocks were thought of as “backward error handling” in which, instead of going forward in an effort to find a new correct state (forward error handling), a program returned to a prior (hopefully correct) state to try and straighten things out. Recovery blocks and the recovery cache that realized them captured the “A” in ACID transactions (all or nothing). That is, the recovery block either executed successfully to completion or undid its effects and tried something else.





```
<identifier>:action(<parameter-list>);
   <statement-list>
   end;
```

```
await:action(test:boolean function, body:procedure);
‡ delay(atomic action until prescience
   tells us that "test" is true, or that
   it escapes, then immediately execute
   the following)
   if test then
      begin;
         body;
         return;
         end;
   else error;  †
   end await;
```

Fig. 1. (a) Nested recovery blocks. (b) Atomic action and “await” statement.

1.3 Dealing with Concurrent Execution

My involvement in “fault tolerant computing” arose as a result of a sabbatical I took from IBM Research in Yorktown. I was working in programming lanaguages at this time and was investigating how programming languages might help in writing understandable concurrent programs. Les Belady suggested that

Brian Randell at Newcastle was doing interesting work in the area of reliability, and, after a rather involved negotiation, I went off to Newcastle to participate in Brian's Reliability Project and to interact with Brian and his collaborator, Michael Melliar-Smith.

I was entranced with the recovery block idea. I decided I would figure out what was needed for the recovery block notion to work in the presence of concurrently executing activites. This is how atomic actions came to be [14]. Atomic actions are similar to procedures in how I chose to define them, as illustrated in Figure 1(b). And in an isolated setting, they behave just like procedures. When concurrent activity is present, the body of an atomic action continues to behave as it did in isolation, while an ordinary procedure, if it accesses shared state, may behave very differently. Figure 1 also illustrates how it is possible to include a synchronizing condition with an atomic action without violating its isolation property.

This work was done at Newcastle, and was greeted with great enthusiasm by both Brian and Michael. And it changed my research career. Atomic actions added the "I" of ACID (isolation) and with recovery blocks hence provided both "A" and "I". Since a user provides the C (consistency), we now had everything but the "D" (durability) of ACID transactions. Given the system setting (where we were not providing durability guarantees to user transactions) this captured the salient parts of atomicity. Though we did not think about it in these terms, this was really the start of transactional memory (of the software kind).

The term transactional memory was first used in a paper by Herlihy and Moss [9], where they described a hardware oriented approach that relied on exploiting processor cache and optimistic concurrency control. Processor caches were still new in 1976, and optimistic concurrency control had not yet been invented. So we had no thought of using these techniques in the way Herlihy and Moss suggested. Rather, we suggested using two phase locking to provide isolation for atomic actions [14] and recovery blocks for "all or nothing" execution. But subsequent to the Herlihy and Moss paper, and recognizing the great value of the atomicity abstraction, many folks have explored pure software implementations for transactional memory. Newcastle had all the pieces for software transactional memory in 1976.

2 Database Connection

2.1 Early Transactions

Brian and the Newcastle Reliability Project did the earliest work of which I am aware that exploited architectural and framework engineering approaches to reliable systems in the fault tolerant computing area. However, overlapping with this, researchers in the database community were wrestling with how to provide concurrent access to a large data store without application programmers needing to use subtle reasoning to get programs concurrently accessing a database to operate in a well-behaved manner. This led (at around the same time) to the notion of database transaction.

IBMs System R team associated the word "transaction" with atomicity, and established it as the key database abstraction in a 1975 paper [8], hence overlapping with the Newcastle effort, but focused on database state, not program state. Databases added durability to the definition of transactions (the "D" of ACID). Durability was essential to support "committed database state change", a promise to a user that the system would not forget his business transaction, e.g. the purchase of an airline ticket, even were the system to subsequently fail.

The key to why database transactions were so successful is the following: an application programmer could disregard whatever any other application was doing once the relevant part of a program was wrapped within transaction brackets, just as was the case with atomic actions. The result was as if an application instance were the only program executing on database state. This is where the "C" of ACID transactions comes into play. If a transaction, operating all by itself could transform an earlier consistent state into a later consistent state in ISOLATION (in the absence of other executing programs) then during concurrent execution, that consistency would continue to hold when the transaction executed.

When I found database systems using transactions, I switched fields (from programming languages) to take part in the research enterprise of implementing and exploring the uses of transactions.

2.2 Ever More Data

Databases and their transactions are all about data. And because data volumes become larger and larger, users want to have their transactional systems scale to handle ever more data. So scalability has been and is a big issue within the database world.

It is possible to ride the technology curve with ever larger single systems with their faster processors, larger memories, and bigger disks. And this has been a boon to the database world. However, the demand for data "knows no bounds", and so databases now support multi-computer deployments. Instead of only "scaling up" with processor power, database systems "scale out" to exploit multiple computer systems. Database scale out takes two forms, shared nothing and data sharing. Figure 2 illustrates these architectures, which are described in the next subsections.

Shared Nothing. A shared nothing database system executes in isolation from other database systems. It is solely responsible for the execution of queries, the modification of data, its caching, its transactions, etc. No coordination with another system is required. It usually works in the context of directly attached disks.

To scale out such systems requires that we partition data among a number of systems. This can work well, and especially so when all transactions can be assigned to and exploit data stored at a single shared nothing DBMS instance.

However, it is typical that no partitioning succeeds in targetting every transaction of an application to a single node. And complexity increases when transactions can span multiple DBMS instances. This is when two phase commit (2PC)

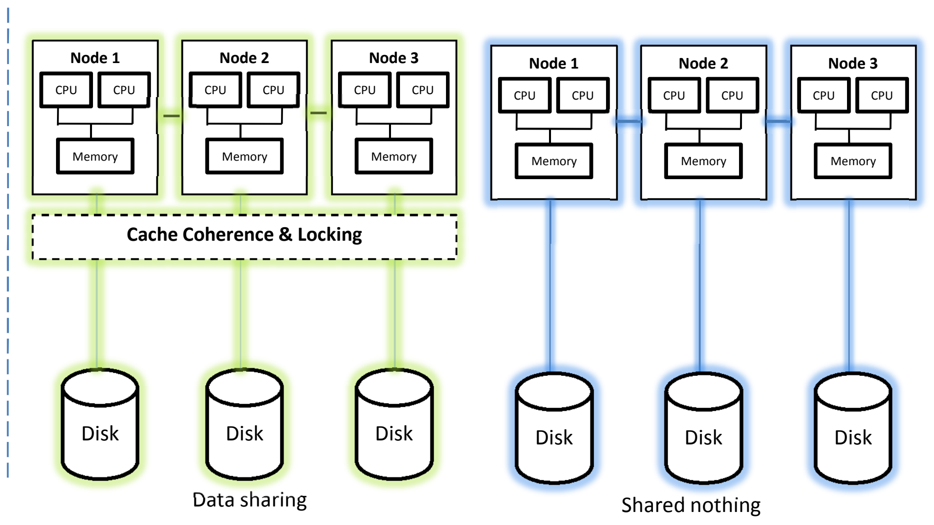


Fig. 2. Two cluster architectures for scaling out to handle more data

is needed to coordinate a distributed transaction. The 2PC protocol requires that every node involved in the transaction agree to its being committed, and involves extra rounds of messages between nodes. This becomes a larger problem as the number of nodes hosting data for an application increases. The result can be that more and more nodes need to be involved in a transaction, leading to the potential for network partitions to disrupt service and block 2PC protocol completion.

In addition to the potential for blocking, the partitioned approach has other difficulties. Two phase commit introduces extra latency which, while not itself a problem, can reduce throughput. Also, load imbalance, where some nodes become saturated while others are under-utilized, can become an issue. To balance the load typically takes some time and planning, and this makes it difficult to do such balancing in real time.

Data Sharing. Data sharing systems are characterized by multiple nodes being able to “share” common data, i.e. operate on it as if it were local, with the ability to cache the data, update it, and store it back to the stable database. The technology price for this is the requirement to handle distributed cache coherence, distributed locking, and coordinated access in some fashion to the transaction log. And this extra technology not only makes data sharing systems more complicated than shared nothing systems, but also usually incurs higher execution overhead.

While both architectural simplicity and performance are compromised, data sharing systems do exhibit decent scalability up to a modest number of nodes, perhaps in the low tens. And this scalability is achieved without the need to partition the data, which makes administering data sharing systems simpler than shared nothing systems. However, data sharing system scalability does not come close to dealing effectively with the huge data sets that are of current interest. Further, data sharing works best when all nodes can directly access the disks, which interferes with exploiting commodity hardware and reaping its economic advantage.

3 The Cloud

We now have a new platform, the cloud, that is the focus of much business and technical interest. This platform presents new technology and business opportunities, and makes possible data storage and possible integration on a scale that we have not really witnessed in the past. What we have is enormous data centers supporting thousands of machines, attached by high speed, relatively low latency communications. Figure 3 is a picture of one such data center, built in a modular way, in locations where costs are low and communications are good. This is truly a concentration of data storage and compute power that we have not seen before.

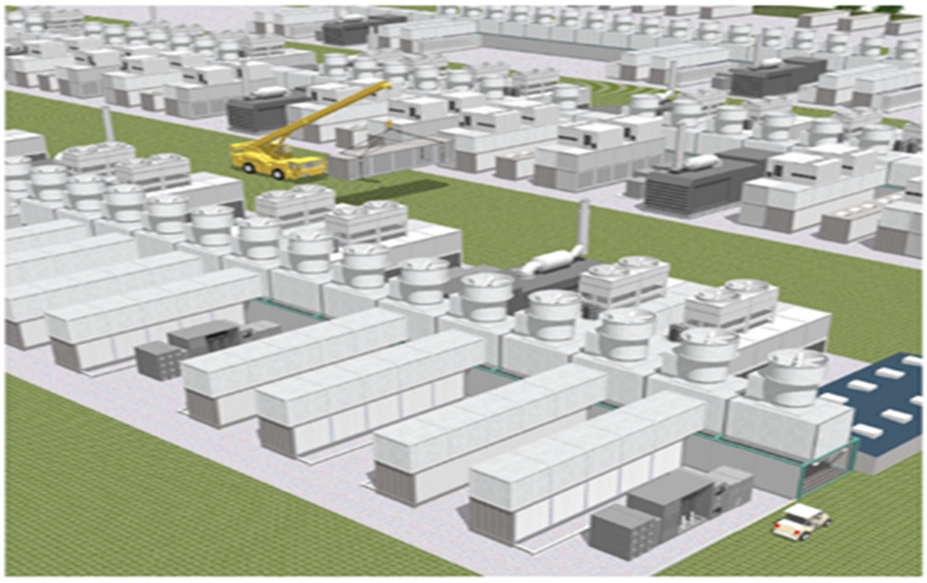


Fig. 3. Microsoft's data center "architecture"

3.1 Economic Imperative

It is not an accident that the cloud has attracted serious attention. The economics of the cloud are compelling, said by some to be a factor of six or seven cheaper than alternative infrastructure. The last time we have seen such a decisive cost factor was when PC based servers displaced mainframe and mini-computer based servers.

Typically such a cloud data center is located where power is cheap and land costs low. The hardware is purchased in bulk at rock bottom prices or specially assembled from even cheaper components. Operations are automated, and multi-tenancy, where customers pay for what they use instead of provisioning for their maximum load, is offered at enormously attractive prices. In addition, cloud providers have found, despite execution on cheap, and occasionally flaky hardware and disks, that they can offer their customers excellent availability via data replication, which is really essential to attracting "bet your business" applications.

3.2 Distributed Systems

Because data centers offer a large set of nodes with communication interconnects, it is natural to think of such a data center as a distributed system. Because data can be distributed over multiple nodes, database users want transactions to work across such systems. Historically, this led to the development of distributed commit protocols, and a collection of variations on this, e.g. two phase commit, three phase commit, optimizations like presumed abort, presumed commit, nested, switch of coordination, timestamping protocols, etc.

Every distributed infrastructure standard has started by defining a "standard" two phase commit (2PC). This includes OSF, DCE, X/Open, OSI, CORBA, OMG, DCOM, etc. But 2PC is rarely used for wide area distributed systems, and "never" crosses administrative domains. This effectively rules out using 2PC for the web. Everyone will coordinate, but no one will participate because 2PC is a blocking protocol and message latencies can be substantial.

Thus, while the intent has been the optimistic vision of data anywhere joined in transactions, what has been delivered is much more limited. Mostly distributed commit protocols are used to commit transactions on a cluster of machines in the same or nearby machine rooms, and all within one administrative domain. The idea has been to handle larger databases "locally" with each node of the system being a simple "shared nothing" system participating in transactions via 2PC when transactions crossed database partiton boundaries.

So the question here is whether this distributed system approach can or should apply to the enormous number of processors in cloud data centers.

3.3 CAP Caution

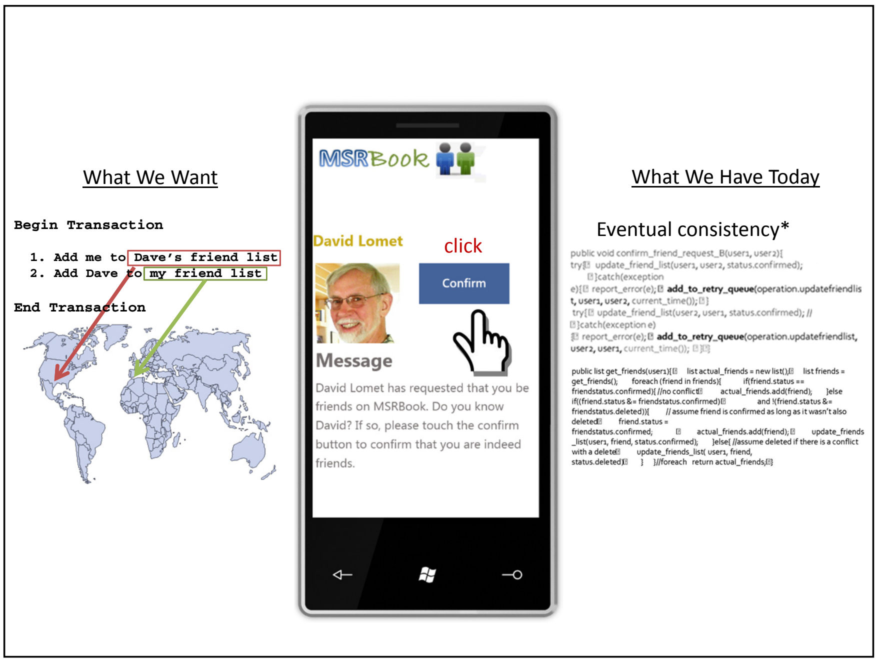
How cloud providers and their customers view data centers colors what kind of functionality will be offered and/or used. If one views a data center as simply another instance of a distributed system, then one needs to pause and take a deep breath when considering the transactional functionality that might be provided or requested.

Brewer's CAP theorem [7] states that you cannot simultaneously have consistency (the "C"), availability (the "A"), and partition resilience (the "P") in a distributed system. This theorem has made cloud providers careful about the transactional functionality that they support as transactions provide the consistency. Cloud providers knew that they needed to give their users high availability, so other aspects were relegated to secondary consideration.

Early on, when companies like Amazon, Google, Yahoo, and Microsoft first rolled out their cloud platforms, they did not include transactions. The view [17] was that, given the CAP theorem, providing transactions was more or less infeasible, especially if you wanted good performance as well as scalability and availability. That CAP included network partitioning and, at least within a data center, that should not really have been an issue, did not prevent this view from becoming wide spread, at least for a while.

3.4 Eventual Consistency and More

Cloud providers hoped that “eventual consistency” would be sufficient. “Eventual consistency” provides the guarantee that the changes that your program makes will eventually show up everywhere, including at all replicas, so long as you can wait sufficiently long. The difficulty with “eventual consistency” is exactly in the lengths that application programmers need to go to ensure that that consistency is achieved. Figure 4 illustrates the extent to which these concerns impact code complexity and understandability [2].



```
Begin Transaction

   1. Add me to Dave's friend list
   2. Add Dave to my friend list

End Transaction
```

Message

David Lomet has requested that you be friends on MSRBook. Do you know David? If so, please touch the confirm button to confirm that you are indeed friends.

```
public void confirm_friend_request_B(user1, user2){
try{□  update_friend_list(user1, user2, status.confirmed);
     □}catch(exception
e){□ report_error(e);□ add_to_retry_queue(operation.updatefriendlis
t, user1, user2, current_time());□}
 try{□ update_friend_list(user2, user1, status.confirmed); //
□}catch(exception e)
{□ report_error(e);□ add_to_retry_queue(operation.updatefriendlist,
user2, user1, current_time()); □}□}

public list get_friends(user1){□    list actual_friends = new list();□    list friends =
get_friends();      foreach (friend in friends){          if(friend.status ==
friendstatus.confirmed){ //no conflict□        actual_friends.add(friend);      }else
if((friend.status &= friendstatus.confirmed)□              and !(friend.status &=
friendstatus.deleted)){        // assume friend is confirmed as long as it wasn't also
deleted□       friend.status =
friendstatus.confirmed;          □       actual_friends.add(friend);□       update_friends
_list(user1, friend, status.confirmed);      }else{ //assume deleted if there is a conflict
with a delete□       update_friends_list( user1, friend,
status.deleted)□      }     }//foreach   return actual_friends;□}
```

Fig. 4. Applications want transactions, not eventual consistency

But things are currently in flux. Google and Microsoft both now provide local transactions without any caveats [4,6]. Amazon is following with Oracle support in the cloud [1]. So local transactions are now seen as not only feasible but to provide real value to users. Indeed, in the database case, that value has never been doubted. And if cloud based databases are to become accepted, transactions are a must. Database users both expect and demand transactional consistency– though they may not always know what form of transaction they are executing as many do not, in fact, execute serializable transactions.

Google even provides distributed transactions via two phase commit in the cloud [4], though they come with a suggestion not to use this capability– or at least not use it often. But distributed transactions local to a data center have been supported in the past [19] with decent performance, and I believe we are moving to a world where this support will become more robust over time.

3.5 A New Platform

Data centers are an entirely new platform. A data center is **not** the web. Data centers that consist of thousands of machines connected by a high speed communication infrastructure surely have failures of parts. But a data center link is no more likely to go down than is a data center processor, perhaps less likely. One way to think about a data center is to consider it as similar to an enormous NUMA machine, where the advantage of local execution and local data access is strong, but where remote execution and remote access is possible as well.

A way to think about all the data center disks is as if they were one large SAN, and treat them accordingly. These analogies are not precise, but each captures an aspect of large data centers, each stressing in a different way, that processing and data access within a data center is fundamentally different from wide area networks of computers and their associated storage. Further, an application hosted at a data center, regardless of where its data is located within the center, has no need to cross administrative domains to access it.

Why should we think that the CAP theorem applies to such data centers, however large. Yet transaction support has been very limited. However, care is required as data can be spread over a very large number of machines.

4 New Look at Transactions on Distributed Data

4.1 How to Exploit Data Centers

While data centers are not the web, they are also not a simple SMP machine, not even of the NUMA variant. Users with large quantities of data typically need to partition it among perhaps a large number of the servers of the data center.

The potential wide spread of data over nodes in a cloud data center currently means that application designers have to be very careful about how their data is partitioned between nodes. The problem with conventional approach is that every node with data involved in the transaction needs to be a two phase commit participant. At the end of the transaction, when you would like to “discharge” the transaction, release its locks, and move on to processing work for other transactions, the system must "pause" to coordinate the distributed transactions among all nodes containing data. This requires extra visits to all the nodes involved at the end of the transaction, before the transaction is committed and data is accessible once again. One visit is to prepare the transaction, the second to commit it. The prepare phase is synchronous, and, absent potential optimizations which do not always apply, include waiting for the write of at least one record to the log.

So how might we view transactions with distributed data? Consider that **we do not do two phase commit with disks** when data is distributed across several disks. Why is this? All that is expected of the disk is that it be an atomic page store (that is not quite true, as we need to take measures to at least detect when a disk fails in that role). But we do not need to get the permission of a disk to commit a transaction, except perhaps the disk on which the log resides. So one way of handling distributed data in the cloud is to use virtual disks provided by the cloud infrastructure, and commit the transaction elsewhere (or perhaps at one of the virtual disks handling the log).

While that is an interesting view, it too has its difficulties. When using the shared nothing strategy, this requires that all caching be done at a single node, even though the quantity of data can be truly enormous. So, while data capacity scales, single node caching restricts application scalability. Further, we need to force flush our log before sending the data to the cloud disk infrastructure. To do that requires that we read all data first, so that we can do the usual undo/redo logging. But cloud latencies and communication overhead make this costly and limit optimization opportunities.

4.2 Deuteronomy

We cannot scale our ability to cache updatable data without permitting multiple machines to participate. This has traditionally forced us to choose between the shared nothing and the data sharing architectures, both of which have difficulties we want to avoid. Shared nothing systems require two phase commit that introduces extra latency at the end of each transaction to execute this protocol. Data sharing introduces a cache coherence problem that has required some form of distributed locking. What we want is distributed and partitioned caching, hence avoiding distributed cache coherence, but without requiring 2PC. This is what the Deuteronomy architecture provides [15,13].

A Deuteronomy based system divides a database engine into transaction component (TC) and data component (DC), isolating transactional functionality (TC) from the rest of data access management (DC). These functions have traditionally been tightly integrated, with many database folks convinced that they were not separable. We beg to differ. By enforcing the right contract between system parts, we can achieve the partitioning into TC and DC, as illustrated in Figure 5.

The Deuteronomy approach, not surprisingly, has its own set of issues. Among them is how to efficiently deal with the interaction of distribution with

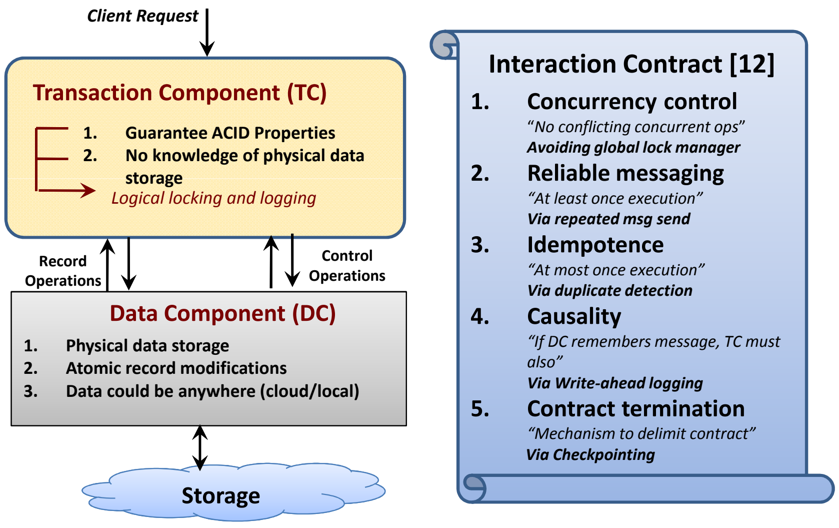

Client Request
Transaction Component (TC)
1. Guarantee ACID Properties
2. No knowledge of physical data storage
Logical locking and logging
Record Operations
Control Operations
Data Component (DC)
1. Physical data storage
2. Atomic record modifications
3. Data could be anywhere (cloud/local)
Storage
Interaction Contract [12]
1. Concurrency control
"No conflicting concurrent ops"
Avoiding global lock manager
2. Reliable messaging
"At least once execution"
Via repeated msg send
3. Idempotence
"At most once execution"
Via duplicate detection
4. Causality
"If DC remembers message, TC must also"
Via Write-ahead logging
5. Contract termination
"Mechanism to delimit contract"
Via Checkpointing

Fig. 5. Deuteronomy Architecture and "Contract"

transaction logging, i.e. to enforce the write-ahead log protocol and provide for checkpointing of the log. Naively, one might think that the log needs to be forced to disk at the TC prior to sending the operation request to the DC. This would lead to excessive forced logging. Instead, the TC:DC contract permits the TC to control when the DC makes operation results stable, permitting the TC to lazily flush its transaction log. Further, the TC cannot checkpoint the log without DC agreement, permitting the DC to be similarly lazy in its posting of results back to the persistent database (e.g. on disk). These are captured in the TC:DC *interaction contract.*

4.3 Deuteronomy Scaling

We do not perform 2PC with our disks, we merely expect them to be atomic page stores. We do not perform 2PC with DCs, we merely expect them to be atomic record stores that support a contract enabling lazy logging and lazy cache management. Thus, the Deuteronomy architecture enables effective transaction processing over data center hosted data that is distributed across very large numbers of machines. Together with the traditional stateless application servers, also perhaps hosted in the data center, this permits transaction processing at "data center scale". This is illustrated in Figure 6. It is even possible to provide transactional functionality for data stored at multiple cloud providers since no 2PC protocol agreement is needed.

There is effectively no limit to data scalability in this picture, and no limit to application logic scalability as well. The TC does limit transaction scalability, i.e. the rate at which transactions and their operations can be executed. However, the TC, with its very modest computational requirements, suggests that transaction

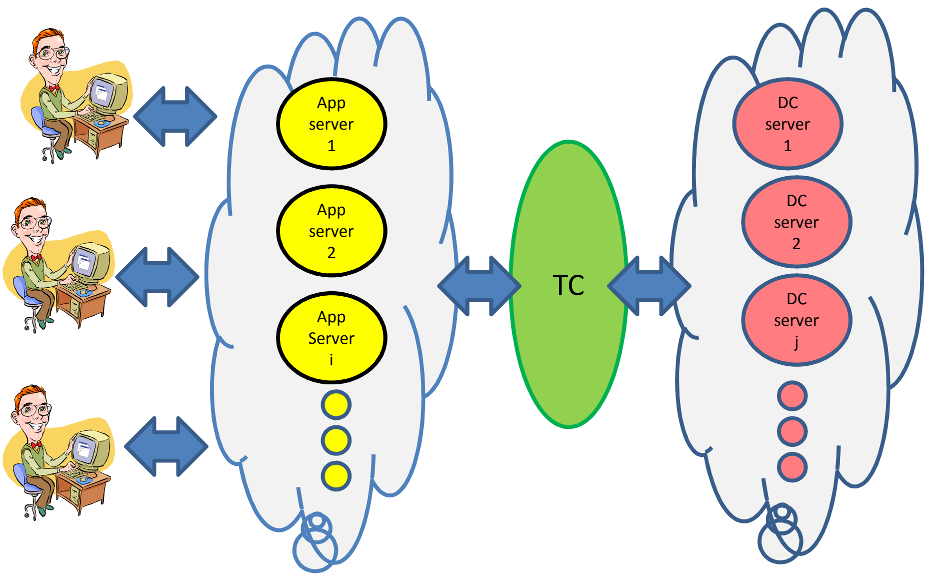


Fig. 6. Scaling that exploits a separate TC

"bandwidth" at the TC hosting node should be very high, even though it is the limiting scalability factor. Very high transaction rates should be achievable. Our prototype implementation provides credible performance with further large improvement opportunities [13].

Transactional scalability can be enhanced by increasing the number of TC's involved in supporting an application. This will require two phase commit among the TC's. However, note that 2PC is not required for every node hosting application data, only with nodes hosting a TC. This breaks the link between where data is stored and who is involved in the 2PC protocol.

5 Summary

Atomic actions and transactions were originally conceived as a local architectural mechanism for handling concurrency control and recovery in systems that were subsequently explored in the fault tolerant and the database fields. Over time, the atomicity mechanism has been stretched to handle more situations, including database systems with ever larger amounts of data spread over ever larger numbers of nodes. The abstraction has shown great resilience, a tribute to the ever increasing cleverness of its supporting mechanisms, produced by large numbers of researchers over many years.

The current cloud environment poses a new challenge, one of unprecedented scale. But, given the incredible value that transactions bring to application programmers, I believe that ways will be found to realize effective and performant atomicity mechanisms in the cloud. Indeed, though I am hardly objective in this, I believe that the Deuteronomy approach holds the promise of making transactional programming the defacto way of accessing cloud data.

Atomicity now has a long history in computer science. An atomic action (serializable transaction) is an architectural abstraction that makes concurrent programs both easier to write and easier to understand. This was and continues to be Brian Randell's vision for how to build and understand large, fault tolerant, and concurrent systems. I want to thank Brian for inviting me to visit on my IBM sabbatical so many years ago and for structuring the environment where the emergence of atomicity was inevitable, both by laying the groundwork with recovery blocks, and by fostering a collaborative environment where so many of us could prosper.

References

1. Amazon: Oracle and AWS, `http://aws.amazon.com/solutions/global-solution-providers/oracle/`
2. Agrawal, D., Abbadi, A.E., Das, S.: Big Data and Cloud Computing: New Wine or just New Bottles? In: VLDB (2010), tutorial
3. Anderson, T., Kerr, R.: Recovery blocks in action: A system supporting high reliability. In: ICSE 1976, pp. 447–457 (1976)
4. Baker, J., Bond, C., Corbett, J., Furman, J.J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In: CIDR 2011, pp. 223–234 (2011)
5. Barga, R.S., Lomet, D.B., Shegalov, G., Weikum, G.: Recovery Guarantees for Internet Applications. ACM Trans. Internet Techn. 4(3), 289–328 (2004)
6. Bernstein, P., Cseri, I., Dani, N., Ellis, N., Kakivaya, G., Kalhan, A., Lomet, D., Manne, R., Novik, L., Talius, T.: Adapting Microsoft SQL Server for Cloud Computing. In: ICDE 2011, pp. 1255–1263 (2011)
7. Brewer, E.A.: Towards Robust Distributed Systems Distributed Systems. PODC Keynote (July 19, 2000)
8. Gray, J., Lorie, R.A., Putzolu, G.R., Traiger, I.L.: Granularity of Locks in a Large Shared Data Base. In: VLDB 1975, pp. 428–451 (1975)
9. Maurice Herlihy, J., Moss, E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: ISCA 1993, pp. 289–300 (1993)
10. Haerder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys 15(4), 287–317 (1983)
11. Horning, J.J., Lauer, H.C., Melliar-Smith, P.M., Randell, B.: A program structure for error detection and recovery. In: Symposium on Operating Systems 1974, pp. 171–187 (1974)
12. Jones, C.B., Lomet, D.B., Romanovsky, A.B., Weikum, G.: The Atomic Manifesto. J. UCS 11(5), 636–651 (2005)
13. Levandoski, J.J., Lomet, D.B., Mokbel, M.F., Zhao, K.: Deuteronomy: Transaction Support for Cloud Data. In: CIDR 2011, pp. 123–133 (2011)
14. Lomet, D.B.: Process Structuring, Synchronization, and Recovery Using Atomic Actions. In: Language Design for Reliable Software, pp. 128–137 (1977)
15. Lomet, D.B., Fekete, A., Weikum, G., Zwilling, M.J.: Unbundling Transaction Services in the Cloud. In: CIDR (2009)

16. Mohan, C., Haderle, D.J., Lindsay, B.G., Pirahesh, H., Schwarz, P.M.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Trans. Database Syst. 17(1), 94–162 (1992)
17. Ramakrishnan, R., Cooper, B., Silberstein, A.: Cloud Data Management @ Yahoo! In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA 2010. LNCS, vol. 5981, pp. 2–2. Springer, Heidelberg (2010)
18. Randell, B.: System Structure for Software Fault Tolerance. IEEE Trans. Software Eng. 1(2), 221–232 (1975)
19. Tandem Database Group: NonStop SQL, A Distributed, High-Performatlce, High-Availability Implementation of SQL. Tandem Technical Report 87.4 (April 1987)