ACM SIGACT News Distributed Computing Column 34

Distributed Computing in the Clouds

Idit Keidar
Dept. of Electrical Engineering, Technion
Haifa, 32000, Israel
idish@ee.technion.ac.il

It seems like "computation clouds" are cropping up everywhere nowadays... well, except perhaps, actually "in the clouds", as a recent April Fool's joke by Amazon suggested[1]. While there is no commonly agreed-upon definition of what exactly constitutes a cloud, it is clear that there are some pretty interesting mega-scale distributed computing environments out there. Such environments require, and already deploy, many distributed services and applications. This column examines distributed computing research that seeks to develop new solutions for clouds, as well as to improve existing ones.

Our main contribution is by Ken Birman, Gregory (Grisha) Chockler, and Robbert van Renesse, who identify a research agenda for cloud computing, based on insights gained at the 2008 LADIS workshop. They question whether contemporary research in distributed computing, which sometimes targets cloud environments, is indeed relevant for cloud computing. Some researchers will be disappointed by (and might disagree with) the conclusions they reach. They then proceed to define a new agenda for cloud computing research. Their article, however, does not consider issues of security and trust. This perhaps stems from the fact that the paper is written from the perspective of cloud service *providers*, rather than *users*, whereas trust is a concern for the latter. In the next contribution, Christian Cachin, Yours Truly, and Alexander (Alex) Shraer examine the *trust* that users have (or can have) in cloud services where they store their data, surveying risks as well as solutions that are being proposed to address them.

The column then turns to a more applied perspective. The next contribution, by Edward (Eddie) Bortnikov from Yahoo! Research, surveys open source technologies that are used for web-scale computing, highlighting some technology transfer from the research community to actual implementations. The column concludes with an announcement, provided by Roger Barga and Jose Bernabeu-Auban from Microsoft, about a Cloud Computing tutorial that will be given at *DISC'2009* in September, in Elche, Spain.

Many thanks to Ken, Grisha, Robbert, Christian, Alex, Eddie, Roger, and Jose for their contributions!

[1] http://aws.typepad.com/aws/2009/03/up-up-and-away-cloud-computing-reaches-for-the-sky.html

Toward a Cloud Computing Research Agenda

Ken Birman
Cornell University
Ithaca, NY, USA
ken@cs.cornell.edu

Gregory Chockler
IBM Research
Haifa, Israel
chockler@il.ibm.com

Robbert van Renesse
Cornell University
Ithaca, NY, USA
rvr@cs.cornell.edu

Abstract

The 2008 LADIS workshop on Large Scale Distributed Systems brought together leaders from the commercial cloud computing community with researchers working on a variety of topics in distributed computing. The dialog yielded some surprises: some hot research topics seem to be of limited near-term importance to the cloud builders, while some of their practical challenges seem to pose new questions to us as systems researchers. This brief note summarizes our impressions.

1 Workshop Background

LADIS is an annual workshop focusing on the state of the art in distributed systems. The workshops are by invitation, with the organizing committee setting the agenda. In 2008, the committee included ourselves, Eliezer Dekel, Paul Dantzig, Danny Dolev, and Mike Spreitzer. The workshop website [1] includes the detailed agenda, white papers, and slide sets [2]; proceedings are available electronically from the ACM Portal web site [21].

2 LADIS 2008 Topic

The 2008 LADIS topic was Cloud Computing, and more specifically:

- Management infrastructure tools (examples would include Chubby [4], Zookeeper [25], Paxos [22], [19], Boxwood [23], Group Membership Services, Distributed Registries, Byzantine State Machine Replication [6], etc),
- Scalable data sharing and event notification (examples include Pub/Sub platforms, Multicast [31], Gossip [30], Group Communication [8], DSM solutions like Sinfonia [1], etc),

[1] http://www.cs.cornell.edu/projects/ladis2008
[2] http://www.cs.cornell.edu/projects/LADIS2008/presentations.htm

- Network-Level and other resource-managed technologies (Virtualization and Consolidation, Resource Allocation, Load Balancing, Resource Placement, Routing, Scheduling, etc),
- Aggregation, Monitoring (Astrolabe [29], SDIMS [32], Tivoli, Reputation).

In 2008, LADIS had three keynote speakers, one of whom shared his speaking slot with a colleague:

- Jerry Cuomo, IBM Fellow, VP, and CTO for IBM's Websphere product line. Websphere is IBMs flagship product in the web services space, and consists of a scalable platform for deploying and managing demanding web services applications. Cuomo has been a key player in the effort since its inception.
- James Hamilton, at that time a leader within Microsoft's new Cloud Computing Initiative. Hamilton came to the area from a career spent designing and deploying scalable database systems and clustered data management platforms, first at Oracle and then at Microsoft. (Subsequent to LADIS, he joined Amazon.com.)
- Franco Travostino and Randy Shoup, who lead eBay's architecture and scalability effort. Both had long histories in the parallel database arena before joining eBay and both participated in eBay's scale-out from early in that company's launch.

We won't try and summarize the three talks (slide sets for all of them are online at the LADIS web site, and additional materials such as blogs [3] and videotaped talks [4]. Rather, we want to focus on three insights we gained by comparing the perspectives articulated in the keynote talks with the cloud computing perspective represented by our research speakers:

- We were forced to revise our "definition" of cloud computing.
- The keynote speakers seemingly discouraged work on some currently hot research topics.
- Conversely, they left us thinking about a number of questions that seem new to us.

3 Cloud Computing Defined

Not everyone agrees on the meaning of cloud computing. Broadly, the term has an "outward looking" and an "inward looking" face. From the perspective of a client outside the cloud, one could cite the Wikipedia definition:

> Cloud computing is Internet (cloud) based development and use of computer technology (computing), whereby dynamically scalable and often virtualized resources are provided as a service over the Internet. Users need not have knowledge of, expertise in, or control over the technology infrastructure "in the cloud" that supports them.

[3] Perspectives: James Hamiltons blog: http://perspectives.mvdirona.com

[4] Randy Shoup on eBay's Architectural Principles:
http://www.infoq.com/presentations/shoup-ebay-architectural-principles

The definition is broad enough to cover everything from web search to photo sharing to social networking. Perhaps the key point is simply that cloud computing resources should be accessible by the end user anytime, anywhere, and from any platform (be it a cell phone, mobile computing platform or desktop). The outward facing side of cloud computing has a growing set of associated standards. By and large:

- Cloud resources are accessed from browsers, "minibrowsers" running JavaScript/AJAX or similar code, or at a program level using web services standards. For example, many cloud platforms employ SOAP as a request encoding standard, and HTTP as the preferred way to actually transmit the SOAP request to the cloud platform, and to receive a reply.

- The client thinks of the cloud as a single entity. Of course, this is just an illusion: in reality, the implementation typically requires one or more data centers, composed of potentially huge numbers of service instances running on a large amount of hardware. Inexpensive commodity PCs structured into clusters are popular. A typical data center has an outward facing bank of servers with which client systems interact directly. Cloud systems implement a variety of DNS and load-balancing/routing mechanisms to control the routing of client requests to actual servers, in a manner that masks the structure of the cloud from its users.

- Further isolating the clients of a cloud system from its internal structure, the external servers may actually be the only ones that a client can access directly. This occurs because those servers often run in a "demilitarized zone" (outside any firewall), and are limited to executing stateless "business logic." This typically involves extracting the client request and parallelizing it within some set of services that do the work and maintain any state associated with the cloud or the transaction. The external server collects replies, combines them into a single "result," and sends it back to the client.

There is also an inside facing perspective:

- A cloud service is implemented by some sort of pool of servers that either share a database subsystem or replicate data [13]. The replication technology is very often supported by some form of scalable, high-speed update propagation technology, such as publish/subscribe message bus (in web services, the term Enterprise Service Bus or ESB is a catch-all for such mechanisms).

- Cloud platforms are highly automated: management of these server pools (including such tasks as launching servers, shutting them down, load balancing, failure detection and handling) are performed by standardized infrastructure mechanisms.

- A cloud system will often provide its servers with some form of shared global file system, or in-memory store services. For example, Google's GFS [15], Yahoo!'s HDFS [3], Amazon.com's S3 [2], memcached [18], and Amazon Dynamo [12] are widely cited. These are specific solutions; the more general statement is simply that servers share files, databases, and other forms of content.

- Server pools often need ways to coordinate when shared configuration or other shared state is updated. In support of this many cloud systems provide some form of locking or atomic multicast mechanism with strong properties [4], [25]. Some very large-scale services use tools like Distributed Hash Tables (DHTs) to rapidly find information shared within a pool of servers, or even as part of a workload partitioning scheme (for example, Amazon's shopping-cart service uses a DHT to spread the shopping cart function over a potentially huge number of server machines).

We've focused the above list on the interactive side of a data center, which supports the clustered server pools with which clients actually interact. But these in turn will often depend upon "back office" functionality: activities that run in the background and prepare information that will be used by the servers actually handling client requests. At Google, these back office roles include computing search indices. Examples of widely known back-office supporting technologies include:

- Scheduling mechanisms that assign tasks to machines, but more broadly, play the role of provisioning the data center as a whole. As we'll see below, this aspect of cloud computing is of growing importance because of its organic connection to power consumption: both to spin disks and run machines, but also because active machines produce heat and demand cooling. Scheduling, it turns out, comes down to "deciding how to spend money."
- Storage systems that include not just the global file system but also scalable database systems and other scalable transactional subsystems and middleware such as Google's BigTable [7], which provides an extensive (conceptually unlimited) table structure implemented over GFS.
- Control systems for large-scale distributed data processing like MapReduce [11] and DryadLINQ [33].
- Archival data organization tools, applications that compress information or compute indexes, applications that look for duplicate versions of objects, etc.

In summary, cloud computing lacks any crisp or simple definition. Trade publications focus on cloud computing as a realization of a form of ubiquitous computing and storage, in which such functionality can be viewed as a new form of cyber-supported "utility". One often reads about the cloud as an analog of the electric power outlet or the Internet itself. From this perspective, the cloud is defined not by the way it was constructed, but rather by the behavior it offers. Technologists, in turn, have a tendency to talk about the components of a cloud (like GFS, BigTable, Chubby) but doing so can lose track of the context in which those components will be used — a context that is often very peculiar when compared with general enterprise computing systems.

4 Is the Distributed Systems Research Agenda Relevant?

We would like to explore this last point in greater detail. If the public perception of the cloud is largely oblivious to the implementation of the associated data centers, the research community can seem oblivious to the way mechanisms are used. Researchers are often unaware that cloud systems have overarching design principles that guide developers towards a cloud-computing mindset quite distinct from what we may have been familiar with from our work in the past, for example on traditional client/server systems or traditional multicast protocols. Failing to keep the broader principles in mind can have the effect of overemphasizing certain cloud computing components or technologies, while losing track of the way that the cloud uses those components and technologies. Of course if the use was arbitrary or similar enough to those older styles of client/server system, this wouldn't matter. But because the cloud demands obedience to those overarching design goals (either because the cloud was built with tools that only support certain styles of system, or because operators such as eBay or Microsoft impose and enforce "rules of practice", as we discuss further below), what might normally seem like mere application-level detail instead turns out to be dominant and to have all sorts of lower level implications.

Just as one could criticize the external perspective ("ubiquitous computing") as an oversimplification, LADIS helped us appreciate that when the research perspective overlooks the roles of our technologies, we

can sometimes wander off on tangents by proposing "new and improved" solutions to problems that actually run contrary to the overarching spirit of the cloud mechanisms that will use these technologies.

To see how this can matter, consider the notion of distributed systems consistency. The research community thinks of consistency in terms of very carefully specified models such as the transactional database model, atomic broadcast, Consensus, etc. We tend to reason along the following lines: Google uses Chubby (a locking service) and Chubby uses State Machine Replication based on Paxos. Thus Consensus, an essential component of State Machine Replication, should be seen as a legitimate cloud computing topic: Consensus is "relevant" by virtue of its practical application to a major cloud computing infrastructure. We then generalize: research on Consensus, new Consensus protocols and tools, alternatives to Consensus are all "cloud computing topics".

While all of this is true, our point is that Consensus, for Google, wasn't the goal. Sure, locking matters in Google, this is why they built a locking service. But the bigger point is that even though large data centers need locking services, if one can trust our keynote speakers, application developers are under huge pressure not to use them. We're reminded of the old story of the blind men touching the elephant. When we reason that "Google needed Chubby, so Consensus as used to support locking is a key cloud computing technology," we actually skip past the actual design principle and jump directly to the details: this way of building a locking service versus that one. In doing so, we lose track of the broader principle, which is that distributed locking is a bad thing that must be avoided!

This particular example is a good one because, as we'll see shortly, if there was a single overarching theme within the keynote talks, it turns out to be that strong synchronization of the sort provided by a locking service must be avoided like the plague. This doesn't diminish the need for a tool like Chubby; when locking actually can't be avoided, one wants a reliable, standard, provably correct solution. Yet it does emphasize the sense in which what we as researchers might have thought of as the main point ("the vital role of consistency and Consensus") is actually secondary in a cloud setting. Seen in this light, one realizes that while research on Consensus remains valuable, it was a mistake to portray it as if it was research on the most important aspect of cloud computing.

Our keynote speakers made it clear that in focusing overly narrowly, the research community of ten misses the bigger point. This is ironic: most of the researchers who attended LADIS are the sorts of people who teach their students to distinguish a problem statement from a solution to that problem, and yet by overlooking the reasons that cloud platforms need various mechanisms, we seem to be guilty of fine-tuning specific solutions without adequately thinking about the context in which they are used and the real needs to which they respond — aspects that can completely reshape a problem statement. To go back to Chubby: once one realizes that locking is a technology of last resort, while building a great locking service is clearly the right thing to do, one should also ask what research questions are posed by the need to support applications that can safely avoid locking. Sure, Consensus really matters, but if we focus too strongly on it, we risk losing track of its limited importance in the bigger picture.

Let's look at a second example just to make sure this point is clear. During his LADIS keynote, Microsoft's James Hamilton commented that for reasons of autonomous control, large data centers have adopted a standard model resembling the well-known Recovery-Oriented Computing (ROC) paradigm [24, 5]. In this model, every application must be designed with a form of automatic fault handling mechanism. In short, this mechanism suspects an application if any other component complains that it is misbehaving. Once suspected by a few components, or suspected strenuously by even a single component, the offending application is rebooted — with no attempt to warn its clients or ensure that the reboot will be graceful or transparent or non-disruptive. The focus apparently is on speed: just push the reboot button. If this doesn't clear the problem, James described a series of next steps: the application might be automatically reinstalled

on a fresh operating system instance, or even moved to some other node — again, without the slightest effort to warn clients.

What do the clients do? Well, they are forced to accept that services behave this way, and developers code around the behavior. They try and use idempotent operations, or implement ways to resynchronize with a server when a connection is abruptly broken.

Against this backdrop, Hamilton pointed to the body of research on transparent task migration: technology for moving a running application from one node to another without disrupting the application or its clients. His point? Not that the work in question isn't good, hard, or publishable. But simply that cloud computing systems don't need such a mechanism: if a client can (somehow) tolerate a service being abruptly restarted, reimaged or migrated, there is no obvious value to adding "transparent online migration" to the menu of options. Hamilton sees this as analogous to the end-to-end argument: if a low level mechanism won't simplify the higher level things that use it, how can one justify the complexity and cost of the low level tool?

Interestingly, although this wasn't really our intention when we organized LADIS 2008, Byzantine Consensus turned out to be a hot topic. It was treated, at least in passing, by surprisingly many LADIS researchers in their white papers and talks. Clearly, our research community is not only interested in Byzantine Consensus, but also perceives Byzantine fault tolerance to be of value in cloud settings.

What about our keynote speakers? Well, the quick answer is that they seemed relatively uninterested in Consensus, let alone Byzantine Consensus. One could imagine many possible explanations. For example, some industry researchers might be unaware of the Consensus problem and associated theory. Such a person might plausibly become interested once they learn more about the importance of the problem. Yet this turns out not to be the case for our four keynote speakers, all of whom have surprisingly academic backgrounds, and any of whom could deliver a nuanced lecture on the state of the art in fault-tolerance.

The underlying issue was quite the opposite: the speakers believe themselves to understand something we didn't understand. They had no issue with Byzantine Consensus, but it just isn't a primary question for them. We can restate this relative to Chubby. One of the LADIS attendees commented at some point that Byzantine Consensus could be used to improve Chubby, making it tolerant of faults that could disrupt it as currently implemented. But for our keynote speakers, enhancing Chubby to tolerate such faults turns out to be of purely academic interest. The bigger — the overarching — challenge is to find ways of transforming services that might seem to need locking into versions that are loosely coupled and can operate correctly without locking [17] — to get Chubby (and here we're picking on Chubby: the same goes for any synchronization protocol) off the critical path.

The principle in question was most clearly expressed by Randy Shoup, who presented the eBay system as an evolution that started with a massive parallel database, but then diverged from the traditional database model over time. As Shoup explained, to scale out, eBay services started with the steps urged by Jim Gray in his famous essay on terminology for scalable systems [13]: they partitioned the enterprise into multiple disjoint subsystems, and then used small clusters to parallelize the handling of requests within these. But this wasn't enough, Shoup argued, and eventually eBay departed from the transactional ACID properties entirely, moving towards a decentralized convergence behavior in which server nodes are (as much as possible) maintained in loosely consistent but transiently divergent states, from which they will converge back towards a consistent state over time.

Shoup argued, in effect, that scalability and robustness in cloud settings arises not from tight synchronization and fault-tolerance of the ACID type, but rather from loose synchronization and self-healing convergence mechanisms.

Shoup was far from the only speaker to make this point. Hamilton, for example, commented that when

a Microsoft cloud computing group wants to use a strong consistency property in a service... his executive team had the policy of sending that group home to find some other way to build the service. As he explained it, one can't always completely eliminate strong ACID-style consistency properties, but the first principle of successful scalability is to batter the consistency mechanisms down to a minimum, move them off the critical path, hide them in a rarely visited corner of the system, and then make it as hard as possible for application developers to get permission to use them. As he said this, Shoup beamed: he has the same role at eBay.

The LADIS audience didn't take these "fighting words" passively. Alvisi and Guerraoui both pointed out that Byzantine fault-tolerance protocols are more and more scalable and more and more practical, citing work to optimize these protocols for high load, sustained transaction streams, and to create optimistic variants that will terminate early if an execution experiences no faults [10], [20]. Yet the keynote speakers pushed back, reiterating their points. Shoup, for example, noted that much the same can be said of modern transaction protocols: they too scale well, can sustain extremely high transaction rates, and are more and more optimized for typical execution scenarios. Indeed, these are just the kinds of protocols on which eBay depended in its early days, and that Hamilton "cut his teeth" developing at Oracle and then as a technical leader of the Microsoft SQL server team. But for Shoup performance isn't the reason that eBay avoids these mechanisms. His worry is that no matter how fast the protocol, it can still cause problems.

This is a surprising insight: for our research community, the prevailing assumption has been that Byzantine Protocols would be used pervasively if only people understood that they no longer need to be performance limiting bottlenecks. But Shoup's point is that eBay avoids them for a different reason. His worry involves what could be characterized as "spooking correlations" and "self-synchronization". In effect, any mechanism capable of "coupling" the behavior of multiple nodes even loosely would increase the risk that the whole data center might begin to thrash.

Shoup related stories about the huge effort that eBay invested to eliminate convoy effects, in which large parts of a system go idle waiting for some small number of backlogged nodes to work their way through a seemingly endless traffic jam. Then he spoke of feedback oscillations of all kinds: multicast storms, chaotic load fluctuations, thrashing. And from this, he reiterated, eBay had learned the hard way that any form of synchronization must be limited to small sets of nodes and used rarely.

In fact, the three of us are aware of this phenomenon from projects on which we've collaborated over the years. We know of many episodes in which data center operators have found their large-scale systems debilitated by internal multicast "storms" associated with publish/subscribe products that destabilized on a very large scale, ultimately solving those problems by legislating that UDP multicast would not be used as a transport. The connection? Multicast storms are another form of self-synchronizing, destructive behavior that can arise when coordinated actions (in this case, loss recovery for a reliable multicast protocol) are unleashed on a large scale.

Thus for our keynote speakers, "fear of synchronization" was an overarching consideration that in their eyes, mattered far more than the theoretical peak performance of such-and-such an atomic multicast or Consensus protocol, Byzantine-tolerant or not. In effect, the question that mattered wasn't actually performance, but rather the risk of destabilization that even using mechanisms such as these introduces.

Reflecting on these comments, which were echoed by Cuomo and Hamilton in other contexts, we find ourselves back in that room with the elephant. Perhaps as researchers focused on the performance and scalability of multicast protocols, or Consensus, or publish/subscribe, we're in the position of mistaking the tail of the beast for the critter itself. Our LADIS keynote speakers weren't naive about the properties of the kinds of protocols on which we work. If anything, we're the ones being naive, about the setting in which those protocols are used.

To our cloud operators, the overarching goal is scalability, and they've painfully learned one overarching principle of scale: decoupling. The key is to enable nodes to quietly go about their work, asynchronously receiving streams of updates from the back-office systems, synchronously handling client requests, and avoiding even the most minor attempt to interact with, coordinate with, agree with or synchronize with other nodes. However simple or fast a consistency mechanism might be, they still view such mechanisms as potential threats to this core principle of decoupled behavior. And thus their insistence on asynchronous convergence as an alternative to stronger consistency: yes, over time, one wants nodes to be consistent. But putting consistency ahead of decoupling is, they emphasized, just wrong.

5 Towards a Cloud Computing Research Agenda

Our workshop may have served to deconstruct some aspects of the traditional research agenda, but it also left us with elements of a new agenda — and one not necessarily less exciting than the one we are being urged by these leaders to shift away from. Some of the main research themes that emerge are:

1. Power management. Hamilton was particularly emphatic on this topic, arguing that a ten-fold reduction in the power needs of data centers may be possible if we can simply learn to build systems that are optimized with power management as their primary goal, and that this savings opportunity may be the most exciting way to have impact today [14]. Examples of ideas that Hamilton floated were:

    - Explore ways to simply do less during surge load periods.
    - Explore ways to migrate work in time. The point here was that load on modern cloud platforms is very cyclical, with infrequent peaks and deep valleys. It turns out that the need to provide acceptable quality of service during the peaks inflates costs continuously: even valley time is made more expensive by the need to own a power supply able to handle the peaks, a number of nodes adequate to handle surge loads, a network provisioned for worst-case demand, etc. Hamilton suggested that rather than think about task migration for fault-tolerance (a topic mentioned above), we should be thinking about task decomposition with the goal of moving work from peak to trough. Hamilton's point was that in a heavily loaded data center coping with a 95% peak load, surprisingly little is really known about the actual tasks being performed. As in any system, a few tasks probably represent the main load, so one could plausibly learn a great deal — perhaps even automatically. Having done this, one could attack those worst-case offenders. Maybe they can precompute some data, or defer some work to be finished up later, when the surge has ended. The potential seems to be very great, and the topic largely unexplored.
    - Even during surge loads, some machines turn out to be very lightly loaded. Hamilton argued that if one owns a node, it should do its share of the work. This argues for migrating portions of some tasks in space: breaking overloaded services into smaller components that can operate in parallel and be shifted around to balance load on the overall data center. Here, Hamilton observed that we lack software engineering solutions aimed at making it easy for the data center development team to delay these decisions until late in the game. After all, when building an application it may not be at all clear that, three years down the road, the application will account for most of the workload during surge loads that in turn account for most of the cost of the data center. Thus, long after an application is built, one needs ways to restructure it with power management as a goal.

- New models and protocols for convergent consistency, perhaps along the lines of [28] or [5]. As noted earlier, Shoup energetically argued against traditional consistency mechanisms related to the ACID properties, and grouped Consensus into this technology area. But it was not so clear to us what alternative eBay would prefer, and in fact we see this as a research opportunity.
- We need to either adapt existing models for convergent behavior (self-stabilization, perhaps, or the forms of probabilistic convergence used in some gossip protocols) to create a formal model that could capture the desired behavior of loosely coupled systems. Such a model would let us replace "loose consistency" with strong statements about precisely when a system is indeed loosely consistent, and when it is merely broken!
- We need a proof methodology and metrics for comparison, so that when distinct teams solve this new problem statement, we can convince ourselves that the solutions really work and compare their costs, performance, scalability and other properties.
- Conversely, the Byzantine Consensus community has value on the table that one would not wish to sweep to the floor. Consider the recent, highly publicized, Amazon.com outage in which that company's S3 storage system was disabled for much of a day when a corrupted value slipped into a gossip-based subsystem and was then hard to eliminate without fully restarting the subsystem — one needed by much of Amazon, and hence a step that forced Amazon to basically shut down and restart. The Byzantine community would be justified, we think, in arguing that this example illustrates not just a weakness in loose consistency, but also a danger associated with working in a model that has never been rigorously specified. It seems entirely feasible to import ideas from Byzantine Consensus into a world of loose consistency; indeed, one can imagine a system that achieves "eventual Byzantine Consensus." One of the papers at LADIS (Rodrigues et al. [26], [27]) presented a specification of exactly such a service. Such steps could be fertile areas for further study: topics close enough to today's hot areas to publish upon, and yet directly relevant to cloud computing.

2. Not enough is known about stability of large-scale event notification platforms, management technologies, or other cloud computing solutions. As we scale these kinds of tools to encompass hundreds or thousands of nodes spread over perhaps tens of data centers, worldwide, we as researchers can't help but be puzzled: how do our solutions work today, in such settings?

   - Very large-scale eventing deployments are known to be prone to destabilizing behavior — a communications-level equivalent of thrashing. Not known are the conditions that trigger such thrashing, the best ways to avoid it, the general styles of protocols that might be inherently robust or inherently fragile, etc.
   - Not very much is known about testing protocols to determine their scalability. If we invent a solution, how can we demonstrate its relevance without first taking leave of our day jobs and signing on at Amazon, Google, MSN or Yahoo? Today, realistically, it seems nearly impossible to validate scalable protocols without working at some company that operates a massive but proprietary infrastructure.
   - Another emerging research direction looks into studying subscription patterns exhibited by the nodes participating in a large-scale publish/subscribe system. Researchers (including the authors of this article) are finding that in real-world workloads, the subscription patters associated with individual nodes are highly correlated, forming clusters of nearly identical or highly similar subscriptions. These structures can be discovered and exploited (through e.g., overlay network

clustering [9], [16], or channelization [31]). LADIS researchers reported on opportunities to amortize message dissemination costs by aggregating multiple topics and nodes, with the potential of dramatically improving scalability and stability of a pub/sub system.

3. Our third point leads to an idea that Mahesh Balakrishnan has promoted: we should perhaps begin to treat virtualization as a first-class research topic even with respect to seemingly remote questions such as the scalability of an eventing solution or a tolerating Byzantine failures. The point Mahesh makes runs roughly as follows:

   - For reasons of cost management and platform management, the data center of the future seems likely to be fully virtualized.
   - Today, one assumes that it makes no sense to talk about a scalable protocol that was actually evaluated on 200 virtual nodes hosted on 4 physical ones: one presumes that internal scheduling and contention effects could be more dominant than the scalability of the protocols per se. But perhaps tomorrow, it will make no sense to talk about protocols that aren't designed for virtualized settings in which nodes will often be co-located. After all, if Hamilton is right and cost factors will dominate all other decisions in all situations, how could this not be true for nodes too?
   - Are there deep architectural principles waiting to be uncovered — perhaps even entirely new operating systems or virtualization architectures — when one thinks about support for massively scalable protocols running in such settings?

4. In contrast to enterprise systems, the only economically sustainable way of supporting Internet scale services is to employ a huge hardware base consisting entirely of cheap off-the-shelf hardware components, such as low-end PC's and network switches. As Hamilton pointed out, this reflects simple economies of scale: i.e., it is much cheaper to obtain the necessary computational and storage power by putting together a bunch of inexpensive PC's than to invest into a high-end enterprise level equipment, such as a mainframe. This trend has important architectural implications for cloud platform design:

   - Scalability emerges as a crosscutting concern affecting all the building blocks used in cloud settings (and not restricted to those requiring strong consistency). Those blocks should be either redesigned with scalability in mind (e.g., by using peer-to-peer techniques and/or dynamically adjustable partitioning), or replaced with new middleware abstractions known to perform well when scaled out.
   - As we scale systems up, sheer numbers confront us with growing frequency of faults within the cloud platform as a whole. Consequently, cloud services must be designed under assumption that they will experience frequent and often unpredictable failures. Services must recover from failures autonomously (without human intervention), and this implies that cloud computing platforms must offer standard, simple and fast recovery procedures [17]. We pointed to a seeming connection to recovery oriented computing (ROC) [24], yet ROC was proposed in much smaller scale settings. A rigorously specified, scalable form of ROC is very much needed.
   - Those of us who design protocols for cloud settings may need to think hard about churn and handling of other forms of sudden disruptions, such as sudden load surges. Existing protocols are too often prone to destabilized behaviors such as oscillation, and this may prevent their use

in large data centers, where such events run the risk of disrupting even applications that don't use those protocols directly.

We could go on at some length, but these points already touch on the highlights we gleaned from the LADIS workshop. Clearly, cloud computing is here to stay, and poses tremendously interesting research questions and opportunities. The distributed systems community, up until now at least, owns just a portion of this research space (indeed, some of the topics mentioned above are entirely outside of our area, or at best tangential).

6 LADIS 2009

In conclusion, LADIS 2008 seems to have been an unqualified success, and indeed, a far more thought-provoking workshop than we three have attended in some time. The key was that LADIS generated spirited dialog between distributed systems researchers and practitioners, but also that the particular practitioners who participated shared so much of our background and experience. When researchers and system builders meet, there is often an impedance mismatch, but in the case of LADIS 2008 we managed to fill a room with people who share a common background and way of thinking, and yet see the cloud computing challenge from very distinct perspectives.

LADIS 2009 is now being planned running just before the ACM Symposium on Operating Systems in October 2009, at Big Sky Resort in Utah. In contrast to the two previous workshops, this year's event is officially sponsored by ACM SIGOPS, and the papers are being solicited through both an open Call For Papers, and targeted solicitation. If SOSP 2009 isn't already enough of an attraction, we would hope that readers of this essay might consider LADIS 2009 to be absolutely irresistible! You are most cordially invited to submit a paper and attend the workshop. More information about the upcoming LADIS workshop can be found on its website [5] and in the call for papers [6]. The first LADIS was held at IBM Haifa Research Lab in 2007 [7].

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of SOSP'07*, pages 159–174, Stevenson, WA, 2007.

[2] Amazon.com. Amazon simple storage service (Amazon S3). 2009. http://aws.amazon.com/s3.

[3] Apache.org. HDFS architecture. 2009. http://hadoop.apache.org/core/docs/current/hdfs_design.html.

[4] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, 2006. USENIX Association.

[5] http://www.sigops.org/sosp/sosp09/workshops/cfp/ladis09/cfp.pdf
[6] http://www.cs.cornell.edu/projects/ladis2009
[7] http://www.haifa.ibm.com/conferences/ladis2007/index.html

[5] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *DSN'09: the thirty-ninth Annual International Conference on Dependable Systems and Networks*, Lisbon, Portugal, 2009. IEEE/IFIP.

[6] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *OSDI '99: Proceedings of the third Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, 1999. USENIX Association.

[7] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, Wallach D.A., M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, Seattle, WA, November 2006.

[8] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[9] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication. In *DEBS '07: Proceedings of the 2007 inaugural International Conference on Distributed Event-Based Systems*, pages 14–25, Toronto, ON, 2007.

[10] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. BFT: the time is now. In LADIS'08 [21]. http://doi.acm.org/10.1145/1529974.1529992.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, 2004. USENIX Association.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of SOSP '07*, pages 205–220, Stevenson, WA, 2007.

[13] B. Devlin, J. Gray, B. Laing, and G. Spix. Scalability terminology: Farms, clones, partitions, and packs: RACS and RAPS. Technical Report MS-TR-99-85, Microsoft Research, 1999. ftp://ftp.research.microsoft.com/pub/tr/tr-99-85.doc.

[14] X. Fan, W.-D. Weber, and L.A. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA '07: Proceedings of the 34th annual International Symposium on Computer Architecture*, pages 13–23, San Diego, CA, 2007. ACM.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of SOSP '03*, pages 29–43, Bolton Landing, NY, 2003. ACM.

[16] S. Girdzijauskas, G. Chockler, R. Melamed, and Y. Tock. Gravity: An interest-aware publish/subscribe system based on structured overlays (Fast Abstract). In *DEBS'08: The 2nd International Conference on Distributed Event-Based Systems*, Rome, Italy, 2008.

[17] J. Hamilton. On designing and deploying Internet-scale services. In *LISA'07: Proceedings of the 21st conference on Large Installation System Administration*, pages 1–12, Dallas, TX, 2007. USENIX Association.

[18] Danga Interactive. memcached: a distributed memory object caching system. 2009. http://www.danga.com/memcached.

[19] J. Kirsch and Y. Amir. Paxos for system builders: An overview. In LADIS'08 [21]. http://doi.acm.org/10.1145/1529974.1529979.

[20] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of SOSP '07*, pages 45–58, Stevenson, WA, 2007. ACM.

[21] LADIS'08: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware. Yorktown Heights, NY, USA, 2008. ACM. http://doi.acm.org/10.1145/1529974.

[22] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[23] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *OSDI'04: Proceedings of the 6th symposium on Operating Systems Design and Implementation*, San Francisco, CA, 2004. USENIX Association.

[24] D Patterson. Recovery Oriented Computing. 2009. http://roc.cs.berkeley.edu.

[25] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In LADIS'08 [21]. http://doi.acm.org/10.1145/1529974.1529978.

[26] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Defining weakly consistent Byzantine fault-tolerant services. In LADIS'08 [21]. http://doi.acm.org/10.1145/1529974.1529990.

[27] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine fault tolerance. In *NSDI'09: Proceedings of USENIX Networked Systems Design and Implementation*, Boston, MA, 2009. USENIX Association.

[28] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of SOSP '95*, pages 172–182, Copper Mountain, CO, 1995. ACM.

[29] R. Van Renesse, K.P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(3), May 2003.

[30] R. Van Renesse, D. Dumitriu, V. Gough, and C. Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In LADIS'08 [21]. http://doi.acm.org/10.1145/1529974.1529983.

[31] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, and Y. Tock. Dr. Multicast: Rx for data-center communication scalability. In *HotNets VII: Seventh ACM Workshop on Hot Topics in Networks*. ACM, 2008.

[32] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *Proceedings of SIGCOMM'04*, Portland, OR, August 2004. ACM.

[33] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of OSDI'08*, San Diego, CA, December 2008. http://research.microsoft.com/en-us/projects/DryadLINQ.

Trusting the Cloud

Christian Cachin
IBM Research
Zurich, Switzerland
cca@zurich.ibm.com

Idit Keidar
Technion
Haifa, Israel
idish@ee.technion.ac.il

Alexander Shraer
Technion
Haifa, Israel
shralex@tx.technion.ac.il

Abstract

More and more users store data in "clouds" that are accessed remotely over the Internet. We survey well-known cryptographic tools for providing integrity and consistency for data stored in clouds and discuss recent research in cryptography and distributed computing addressing these problems.

Storing data in clouds

Many providers now offer a wide variety of flexible online data storage services, ranging from passive ones, such as online archiving, to active ones, such as collaboration and social networking. They have become known as computing and storage "clouds." Such clouds allow users to abandon local storage and use online alternatives, such as Amazon S3, Nirvanix CloudNAS, or Microsoft SkyDrive. Some cloud providers utilize the fact that online storage can be accessed from any location connected to the Internet, and offer additional functionality; for example, Apple MobileMe allows users to synchronize common applications that run on multiples devices. Clouds also offer computation resources, such as Amazon EC2, which can significantly reduce the cost of maintaining such resources locally. Finally, online collaboration tools, such as Google Apps or versioning repositories for source code, make it easy to collaborate with colleagues across organizations and countries, as practiced by the authors of this paper.

What can go wrong?

Although the advantages of using clouds are unarguable, there are many risks involved with releasing control over your data. One concern that many users are aware of is loss of privacy. Nevertheless, the popularity of social networks and online data sharing repositories suggests that many users are willing to forfeit privacy, at least to some extent. Setting privacy aside, in this article we survey what else "can go wrong" when your data is stored in a cloud.

Availability is a major concern with any online service, as such services are bound to have some down-time. This was recently the case with Google Mail[1], Hotmail[2], Amazon S3[3] and MobileMe[4]. Users must also understand their service contract with the storage provider. For example, what happens if your payment for the storage is late? Can the storage provider decide that one of your documents violates its policy and terminate your service, denying you access to the data? Even the worst scenarios sometimes come true — a cloud storage-provider named LinkUp (MediaMax) went out of business last year after losing 45% of stored client data due to an error of a system administrator[5]. This incident also revealed that it is sometimes very costly for storage providers to keep storing old client data, and they look for ways to offload this responsibility to a third party. Can a client make sure that his data is safe and available?

No less important is guaranteeing the integrity of remotely stored data. One risk is that data can be damaged while in transit to or from the storage provider. Additionally, cloud storage, like any remote service, is exposed to malicious attacks from both outside and inside the provider's organization. For example, the servers of the Red Hat Linux distribution were recently attacked and the intruder managed to introduce a vulnerability and even sign some packages of the Linux operating-system distribution[6]. In its Security Advisory about the incident, Red Hat stated:

> *... we remain highly confident that our systems and processes prevented the intrusion from compromising RHN or the content distributed via RHN and accordingly believe that customers who keep their systems updated using Red Hat Network are not at risk.*

Unauthorized access to user data can occur even when no hackers are involved, e.g., resulting from a software malfunction at the provider. Such data breach occurred in Google Docs[7] during March 2009 and led the Electronic Privacy Information Center to petition[8] with the Federal Trade Commission asking to *"open an investigation into Google's Cloud Computing Services, to determine the adequacy of the privacy and security safeguards..."*. Another example, where data integrity was compromised as a result of provider malfunctions, is a recent incident with Amazon S3, where users experienced silent data corruption[9]. Later Amazon stated in response to user complaints[10]:

> *We've isolated this issue to a single load balancer that was brought into service at 10:55pm PDT on Friday, 6/20. It was taken out of service at 11am PDT Sunday, 6/22. While it was in service it handled a small fraction of Amazon S3's total requests in the US. Intermittently, under load, it was corrupting single bytes in the byte stream ... Based on our investigation with both internal and external customers, the small amount of traffic received by this particular load balancer, and the intermittent nature of the above issue on this one load balancer, this appears to have impacted a very small portion of PUTs during this time frame.*

A further complication arises when multiple users collaborate using cloud storage (or simply when one user synchronizes multiple devices). Here, consistency under concurrent access must be guaranteed.

[1] http://googleblog.blogspot.com/2009/02/current-gmail-outage.html
[2] http://www.datacenterknowledge.com/archives/2009/03/12/downtime-for-hotmail
[3] http://status.aws.amazon.com/s3-20080720.html
[4] http://blogs.zdnet.com/projectfailures/?p=908
[5] http://blogs.zdnet.com/projectfailures/?p=999
[6] https://rhn.redhat.com/errata/RHSA-2008-0855.html
[7] http://blogs.wsj.com/digits/2009/03/08/1214/
[8] http://cloudstoragestrategy.com/2009/03/trusting-the-cloud-the-ftc-and-google.html
[9] http://blogs.sun.com/gbrunett/entry/amazon_s3_silent_data_corruption
[10] http://developer.amazonwebservices.com/connect/thread.jspa?threadID=22709

A possible solution that comes to mind is using a Byzantine fault-tolerant replication protocol within the cloud (e.g., [14]); indeed this solution can provide perfect consistency and at the same time prevent data corruption caused by some threshold of faulty components within the cloud. However, since it is reasonable to assume that most of the servers belonging to a particular cloud provider run the same system installation and are most likely to be physically located in the same place (or even run on the same machine), such protocols might be inappropriate. Moreover, cloud-storage providers might have other reasons to avoid Byzantine fault-tolerant consensus protocols, as explained by Birman et al. [3]. Finally, even if this solves the problem from the perspective of the storage provider, here we are more interested in the users' perspective. A user perceives the cloud as a single trust domain and puts trust in it, whatever the precautions taken by the provider internally might be; in this sense, the cloud is not different from a single remote server. Note that when multiple clouds from different providers are used, running Byzantine-fault-tolerant protocols across several clouds might be appropriate (see next section).

What can we do?

Users can locally maintain a small amount of trusted memory and use well-known cryptographic methods in order to significantly reduce the need for trust in the storage cloud. A user can verify the integrity of his remotely stored data by keeping a short hash in local memory and authenticating server responses by re-calculating the hash of the received data and comparing it to the locally stored value. When the volume of data is large, this method is usually implemented using a hash tree [25], where the leaves are hashes of data blocks, and internal nodes are hashes of their children in the tree. A user is then able to verify any data block by storing only the root hash of the tree corresponding to his data [4]. This method requires a logarithmic number of cryptographic operations in the number of blocks, as only one branch of the tree from the root to the hash of an actual data block needs to be checked. Hash trees have been employed in many storage-system prototypes (TDB [22] and SiRiUS [13] are just two examples) and are used commercially in the Solaris ZFS filesystem[11]. Research on efficient cryptographic methods for authenticating data stored on servers is an active area [26, 28].

Although these methods permit a user to verify the integrity of data returned by a server, they do not allow a user to ascertain that the server is able to answer a query correctly without actually *issuing* that particular query. In other words, they do not assure the user that all the data is "still there". As the amount of data stored by the cloud for a client can be enormous, it is impractical (and might also be very costly) to retrieve all the data, if one's purpose is just to make sure that it is stored correctly. In recent work, Juels and Kaliski [18] and Ateniese et al. [2] introduced protocols for assuring a client that his data is retrievable with high probability, under the name of *Proofs of Retrievability* (PORs) and *Proofs of Data Possession* (PDP), respectively. They incur only a small, nearly constant overhead in communication complexity and some computational overhead by the server. The basic idea in such protocols is that additional information is encoded in the data prior to storing it. To make sure that the server really stores the data, a user submits challenges for a small sample of data blocks, and verifies server responses using the additional information encoded in the data. Recently, some improved schemes have been proposed and prototype systems have been implemented [29, 6, 5].

The above tools allow a single user to verify the integrity and availability of his own data. But when multiple users access the same data, they cannot guarantee integrity between a writer and multiple readers. Digital signatures may be used by a client to verify integrity of data created by others. Using this method, each client needs to sign all his data, as well as to store an authenticated public key of the others or the

[11] http://blogs.sun.com/bonwick/entry/zfs_end_to_end_data

root certificate of a public-key infrastructure in trusted memory. This method, however, does not rule out all attacks by a faulty or malicious storage service. Even if all data is signed during write operations, the server might omit the latest update when responding to a reader, and even worse, it might "split its brain," hiding updates of different clients from each other. Some solutions use trusted components in the system [11, 31] which allow clients to audit the server, guaranteeing atomicity even if the server is faulty. Without additional trust assumptions, the atomicity of all operations in the sense of linearizability [16] cannot be guaranteed; in fact, even weaker consistency notions, like sequential consistency [19], are not possible either [9]. Though a user may become suspicious when he does not see any updates from a collaborator, the user can only be certain that the server is not holding back information by communicating with the collaborator directly; such user-to-user communication is indeed employed in some systems for this purpose.

If not atomicity, then what consistency can be guaranteed to clients? The first to address this problem were Mazières and Shasha [24], who defined a so-called *forking* consistency condition. This condition ensures that if certain clients' perception of the execution becomes different, for example if the server hides a recent value of a completed write from a reader, then these two clients will never again see each other's newer operations, or else the server will be exposed as faulty. This prevents a situation where one user sees part of the updates issued by another user, and the server can choose which ones. Moreover, fork-consistency prevents Alice from seeing new updates by Bob and by Carol, while Bob sees only Alice's updates, where Alice and Bob might think they are mutually consistent, though they actually see different states. Essentially, with fork consistency, each client has a linearizable view of a sub-sequence of the execution, and client views can only become disjoint once they diverge from a common prefix; a simple definition can be found in [7]. The first protocol of this kind, realizing fork-consistent storage, was implemented in the SUNDR system [20].

To save cost and to improve performance, several weaker consistency conditions have been proposed. The notion of fork-sequential-consistency, introduced by Oprea and Reiter [27], allows client views to violate real-time order of the execution. The fork-* consistency condition due to Li and Mazières [21] allows the views of clients to include one more operation without detecting an attack after their views have diverged. This condition was used to provide meaningful service in a Byzantine-fault-tolerant replicated system, even when more than a third of the replicas are faulty [21].

Although consistency in the face of failures is crucial, it is no less important that the service is unaffected in the common case by the precautions taken to defend against a faulty server. In recent work [8, 7], we show that for all previously existing forking consistency conditions, and thus in the protocols that implement them with a single remote server, concurrent operations by different clients may block each other even if the provider is correct. More formally, these consistency conditions do not allow for protocols that are wait-free [15] when the storage provider is correct. We have also introduced a new consistency notion, called weak fork-linearizability, that does not suffer from this limitation, and yet provides meaningful semantics to clients [7].

One disadvantage of forking consistency conditions is that they are not so intuitive to understand as atomicity, for example. Aiming to provide simpler guarantees, we have introduced the notion of a Fail-Aware Untrusted Service [7]. Its basic idea is that each user should know which of his operations are seen consistently by each of the other users, and in addition, find out whenever the server violates atomicity. When all goes well, each operation of a user eventually becomes "stable" with respect to every other correct user, in the sense that they have a common view of the execution up to this operation. Thus, in all cases, users get either positive notifications indicating operation stability, or negative notifications when the server violates atomicity. Our Fail-Aware Untrusted Services rely on the well-established notions of eventual consistency [30] and fail-awareness [12], and adapt them to this setting. The FAUST protocol [7] implements

this notion for a storage service, using an underlying weak fork-linearizable storage protocol. Intuitively, FAUST indicates stability as soon as additional information is gathered, either through the storage protocol, or whenever the clients communicate directly. However, all complete operations, even those not yet known to be stable, preserve causality [17]. Moreover, when the storage server is correct, FAUST guarantees strong safety (linearizability) and liveness (wait-freedom).

Obviously, if the cloud provider violates its specification or simply does not respond, not much can be done other than detecting this and taking one's business elsewhere in the future. It is, however, possible to be more prudent, and use multiple cloud providers from the outset, and here one can benefit from the fruitful research on Byzantine-fault-tolerant protocols. One possibility is running Byzantine-fault-tolerant state-machine replication, where each cloud maintains a single replica [10, 14]. This approach, however, requires computing resources within the cloud, as provided, e.g., by Amazon EC2, and not only storage. When only a simple storage interface is available, one can work with Byzantine Quorum Systems [23], e.g., by using Byzantine Disk Paxos [1]. However, in order to guarantee the atomicity of user operations and to tolerate the failure of one cloud, such protocols must employ at least four different clouds.

Summary

Though clouds are becoming increasingly popular, we have seen that some things can "go wrong" when one trusts a cloud provider with his data. Providing defenses for these is an active area of research. We presented a brief survey of solutions being proposed in this context. Nevertheless, these solutions are, at this point in time, academic. There are still questions regarding how well these protections can work in practice, and moreover, how easy-to-use they can be. Finally, we have yet to see how popular storing data in clouds will become, and what protections users will choose to use, if any.

References

[1] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk Paxos: Optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[2] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. ACM CCS*, pages 598–609, 2007.

[3] K. Birman, G. Chockler, and R. van Renesse. Towards a cloud computing research agenda. *SIGACT News*, 40(2), June 2009.

[4] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994.

[5] K. D. Bowers, A. Juels, and A. Oprea. Hail: A high-availability and integrity layer for cloud storage. Cryptology ePrint Archive, Report 2008/489, 2008. http://eprint.iacr.org/.

[6] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. Cryptology ePrint Archive, Report 2008/175, 2008. http://eprint.iacr.org/.

[7] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *Proc. DSN 2009, to appear. Full paper available as Tech. Report CCIT 712, Department of Electrical Engineering, Technion*, Dec. 2008.

[8] C. Cachin, I. Keidar, and A. Shraer. Fork sequential consistency is blocking. *IPL*, 109(7), 2009.

[9] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. PODC*, pages 129–138, 2007.

[10] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. OSDI*, pages 173–186, 1999.

[11] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. SOSP*, pages 189–204, 2007.

[12] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *Proc. PODC*, 1996.

[13] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proc. NDSS*, 2003.

[14] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proc. SOSP*, 2007.

[15] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 11(1), 1991.

[16] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[17] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *Proc. ICDCS*, 1990.

[18] A. Juels and B. S. K. Jr. Pors: proofs of retrievability for large files. In *Proc. ACM CCS*, pages 584–597, 2007.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[20] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI*, 2004.

[21] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. NSDI*, 2007.

[22] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. OSDI*, 2000.

[23] D. Malkhi and M. K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[24] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. PODC*, 2002.

[25] R. C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[26] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2(2):107–138, 2006.

[27] A. Oprea and M. K. Reiter. On consistency of encrypted files. In *Proc. DISC*, 2006.

[28] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proc. ACM CCS*, pages 437–448, 2008.

[29] H. Shacham and B. Waters. Compact proofs of retrievability. In J. Pieprzyk, editor, *Proceedings of Asiacrypt 2008*, volume 5350 of *LNCS*, pages 90–107. Springer-Verlag, Dec. 2008.

[30] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP*, 1995.

[31] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *ACM Transactions on Storage*, 3(3), 2007.

Open-Source Grid Technologies for Web-Scale Computing

Edward Bortnikov
Yahoo! Research
ebortnik@yahoo-inc.com

1 Introduction

Analyzing web-scale datasets has become a key routine for all leading Internet companies. For example, machine-learning of search relevance results from web query logs provides vital feedback for improving the quality of Internet search. Consider, for example, the task of ingesting the events generated by online users. Billions of interesting events (e.g., web queries and ad clicks) happening daily translate to multi-terabyte data collections. Real-time capturing, storage, and analysis of this data are common needs of all high-end online applications.

Grid computing technologies that emerged in recent years (e.g. [5, 16, 17, 20, 21]) address these requirements, thus enabling data-intensive supercomputing at web scale. They allowed establishing data centers with hundreds of thousands of CPU cores, terabytes of RAM, and petabytes of disk space (e.g., [1]), in which multiple data processing applications share a common infrastructure. Typically, these data centers harness commodity hardware – off-the-shelf PCs with directly-attached storage[1]. Grid computing software lets developers easily write, deploy, and run data-intensive applications, which commonly require:

1. Storage management of petabytes.
2. Parallel high-speed access to the stored data.
3. Reliability of (1) and (2) in the face of hardware, software, and networking failures.

While the first generation of grid middleware was mainly proprietary (e.g., [17, 21]), the open-source community is rapidly catching up with its own technology, Apache Hadoop [5], which allows much wider exposure and faster innovation. Hadoop was started by Doug Cutting in 2005, and became a top-level Apache project in 2008. Nowadays, Hadoop is embraced by a variety of academic and industrial users, including (in alphabetical order) Amazon A9, Cornell University, ETH, Facebook, IBM, Microsoft, Yahoo!, and many others [4].

[1] Recently, Google unveiled a custom server design which uses standard components [1].

Hadoop Core, the most mature part of this technology, provides two main abstractions: a distributed file system, HDFS (Section 2), and a MapReduce programming framework for processing large datasets (Section 3). Higher levels in the software stack feature Pig [8] and Hive [7], user-friendly parallel data processing languages, Zookeeper (Section 4), a high-availability directory and configuration service, and HBase [6], a web-scale distributed column-oriented store modeled after its proprietary predecessors [13, 14].

Parts of the Hadoop ecosystem emerged from original research [22, 25]. Researchers and developers in the open-source community are working on a variety of open problems, ranging from new paradigms for large-scale information processing to systems-related issues like fault tolerance, task scheduling, and power management. Section 5 discusses some of these efforts.

2 HDFS

The Hadoop distributed file system is designed for batch processing applications that need streaming access to very large files across multiple machines (nodes). These objectives lead to a few clear design choices:

1. *Moving computation closer to the data*. Experience in developing high-performance computing systems teaches that moving computation is more efficient than moving data. This is why HDFS does not separate data nodes from computation nodes (as many enterprise storage systems do). Instead, it opts for storing data on inexpensive directly-attached disks, and provides API-level visibility into data placement, thus enabling data-driven application migration.

2. *Relaxed file access semantics*. Datasets stored in HDFS are typically accessed in a write-once-read-many pattern, in contrast with a mixed read/write access which in traditional multi-user file systems. For this reason, some hard consistency requirements of the traditional POSIX API can be sacrificed for the sake of improved performance. For example, writes can be lost (and redone prior to the first read), and there is no need in locking for concurrency control.

3. *Large read-only files*. HDFS's performance is tuned to large sequential scans, which affects the disk layout and access optimizations (e.g., local caching).

4. *Handling hardware failures*. Finally, the file system overcomes a constant fraction of computer, disk, and network failures through checksums and data replication.

The HDFS architecture distinguishes between *namenodes*, which host the file system's metadata, and *datanodes*, which store data blocks. An HDFS cluster consists of a single namenode and multiple datanodes. A namenode performs file system management operations (allocating block storage, manipulating files handles, etc.). A datanode manages its disk as per the master namenode's instructions, and serves the clients' datapath (read/write) requests. Thus, the namenode is a single point of control but not an I/O bottleneck. Namenodes keep persistent logs of committed control operations to enable fast recovery. The Bookkeeper project (Section 4) introduces additional fault-tolerance features for namenodes, e.g., reliable remote logging.

HDFS replicates the data blocks for resilience. The API allows independent control of the *replication factor* of each file – i.e., the number of copies of each block within the file. Optimizing replica placement distinguishes HDFS from most other distributed file systems. I/O-efficient replica placement policies must deal with multiple constraints like the data center's topology, LAN speeds versus disk speeds, etc. The currently adopted *rack-aware* policy is a first effort in this direction. It exploits the fact that datanodes are organized into hardware racks interconnected by switches. The policy employs a replication factor of 3.

It places two replicas within the same rack, and one more in a remote rack. Reads are served from the closest replica to reduce latency, whereas writes are pipelined among replicas to increase throughput. Each HDFS namenode monitors the liveness of the managed datanodes, as well as their disk utilization. It can initiate re-distribution of data within the cluster, e.g., re-replication in case some replicas fail, or automatic re-partitioning in case of uneven use of storage. Adaptive allocation policies are subject for future research.

Large datasets tend to be written once by a single user, mostly in streaming mode [17]. A typical block size in HDFS is 64 MB (compared to 4KB for mainstream Linux file systems). The namenode tries to spread multiple blocks of the same file to different datanodes. The write-once-read-many semantics allow relaxing some of the traditional POSIX consistency requirements. For instance, HDFS does not implement advisory locks for concurrent updates, neither does it support random writes. Another example in which HDFS deviates from traditional file systems is its *staging* optimization, which trades client side caching for durability guarantees. Under this policy, a file creation request does not reach the namenode immediately. Instead, the HDFS client creates a temporary local file. Only when enough data has been written to fill a data block, the client contacts the namenode, which inserts the file into the HDFS namespace. The namenode allocates a datanode to store the block (and more datanodes for more data). If the namenode crashes in the interim, the file is lost.

3 MapReduce

Map-reduce frameworks (e.g., [21]) proved to be very natural for processing large datasets in streaming mode, e.g., web index building or training email spam filters. A map-reduce job usually splits the input dataset into independent chunks, which are processed by the *map* tasks in parallel. A map task executes a user function to transform input (key,value) pairs into a new set of (key,value) pairs. The framework sorts the outputs of the maps, and forwards them to the *reduce* tasks. A reduce task combines all (key,value) pairs with the same key into new (key,value) pairs. Finally, the reduced outputs are stored in a file system.

The word-count computer program, which outputs the number of instances of each word within a collection of files, is often used to demonstrate the paradigm. In this context, each map task processes a subset of files. For each file, it emits a sequence of (`term, "1"`) pairs for each word `term` in the file. All pairs with the the same `term` key are mapped to the same reduce task, which summarizes the count of `"1"`s, and outputs it. A slight variation of this example builds term *posting lists*, which contain all locations of each term within a document corpus. This *document inversion* operation is the base of building an efficient web search index.

Hadoop implements a map-reduce Java API, and the supporting runtime system. For non-Java programmers, it offers Hadoop Streaming – a utility that allows users to create and run jobs with arbitrary executables (e.g., shell utilities) as the mapper and the reducer. Programmers looking for higher-level data processing abstractions can resort either to Pig [8, 25], a procedural yet powerful query language, or Hive [7], a SQL-like declarative language.

Hadoop map-reduce tasks store their final and intermediate outputs in HDFS. Data compression is used aggressively to reduce the required I/O bandwidth. The runtime system exploits the visibility of data placement within HDFS to move computation tasks closer to their data. The framework takes care of scheduling tasks, monitoring them, and re-executing the failed ones. Task granularity is configurable, which allows trading fine-grained load balancing and fast recovery for I/O efficiency. Hadoop supports speculative execution – it runs duplicates of slow tasks, and picks those that finish first. This alleviates the need for accurate failure detectors, and suppresses the job latencies' long tail.

Yahoo!'s Webmap application is an example of successful deployment of map-reduce. This system

maintains a gigantic table of information about every web site, page, and link the search engine knows about. Webmap provides infrastructure for various algorithms for e.g., ranking, de-duplication, region classification, and spam detection. Porting Webmap to Hadoop allowed researchers focus on these applications rather than on the platform. The new system achieved 33% improvement in job latency compared to a similar-size cluster built with the previous technology. Its largest jobs perform above 100K maps and 10K reduces, handling 300 TB of data, and producing 200 TB of compressed output [10].

4 ZooKeeper and BookKeeper

ZooKeeper [9] is a coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can use to share data reliably, e.g., implement a distributed configuration repository. ZooKeeper is optimized for read-dominated access to small objects (e.g., application metadata). It leverages in practice many achievements of the distributed algorithms community.

ZooKeeper provides to its clients the abstraction of a set of data objects (*znodes*), organized in a hierarchical namespace resembling a file system structure. The client API includes object manipulation (`create`/`delete`), data access (`read`/`write`), and change notifications (`watch`). For fault-tolerance, all znodes are replicated across multiple servers.

Znodes are essentially distributed shared read/write registers [23], extended with the watch abstraction. ZooKeeper provides sequential consistency [23], i.e., all clients observe the same order of writes, but reads may return stale data. This approach allows for local reads – i.e., a server can reply to a client request without coordinating with the other servers. A client that wishes to receive fresh data can force its server to synchronize (`sync`) with the rest of the cluster.

ZooKeeper servers implement a leader-based atomic broadcast protocol to guarantee agreement on the order of writes. This implementation is not wait-free (i.e., some write requests may theoretically block forever [23]). However, to optimize for read-dominated workload, it has been preferred over a wait-free implementation of a linearizable shared register [11] in which reads cannot be served locally.

The service implements watches to avoid frequent probes for changes on znodes. The order of change notifications received by watch clients is identical to the order of writes. Servers manage their watches locally, i.e., a server notifies its clients upon learning about a change. Therefore, some clients might not receive notifications in realtime. Similarly to reads, a client must explicitly sync in order to receive fresh notifications.

The ZooKeeper API allows building a variety of synchronization primitives on top of the shared object API - e.g., a distributed lock service like Chubby [12]. It offers a neat mechanism of *ephemeral nodes* to track group membership changes, e.g., for systems that wish to implement leader election [23].

ZooKeeper is successfully deployed within production systems, e.g., Web crawlers and publish-subscribe platforms. The project's roadmap includes new optimizations for read and write scaling, dynamic cluster re-configuration [18], and replacing atomic broadcast with quorum systems [23].

BookKeeper BookKeeper is a service for reliable storage of write-ahead logs. Many critical systems, e.g., relational databases and journal file systems, employ write-ahead logging (WAL) to guarantee recoverability [19]. With WAL, a transaction appends a state change record to the persistent log; this change may be applied to the main storage asynchronously after the transaction's completion. In case of system crash, recovery is achieved through replaying changes committed to the log. WAL also reduces transaction latencies, because it replaces random I/O with sequential writes to the log.

The remote BookKeeper service replicates the log across multiple servers, or *bookies*. It can handle arbitrary (Byzantine) failures of less than $\frac{1}{4}$ of all bookies, as well as Byzantine clients. BookKeeper employs a read-write quorum system for accessing bookies [15], and stores its metadata in ZooKeeper. As a proof of concept, BookKeeper has been integrated into the HDFS namenode (Section 2), in which it replaced the non-fault-tolerant logging to a local file. Experiments show that this change boosts the nodename throughput by 30% in typical configurations.

5 What Next?

Hadoop's resource management is still in its infancy. For example, the system can successfully control, through its job tracker component, the execution of multiple tasks within a single job on a dedicated cluster. The Hadoop-on-demand (HOD) technology [2] allows provisioning such isolated virtual clusters within a data center. However, this approach makes resource sharing among multiple jobs problematic. More recent research and development target a variety of issues, like:

1. Better scheduling policies (e.g., job pooling by size, and fair scheduling within the pools [26]).
2. Improving the scheduling policies in heterogeneous environments, e.g., virtualized instances deployed within a remote grid infrastructure like Amazon EC2 [27].
3. Sharing tasks among multiple map-reduce jobs, to run common computations once.

At the data processing side, supporting *interactive* queries over Web-scale data is the next challenge. For example, batch queries over gigantic datasets like Webmap (Section 3) can take hours to evaluate. Recent research [24] suggests splitting the querying process into two stages: first supply a query *template*, and later supply specific instantiations of the template. With this approach, the pre-processing stage, which needs not be realtime, can pre-compute and cache the (partial) results of instantiations that incur high query latencies.

Hadoop's wiki [3] outlines many more interesting research directions. These include enhanced data placement, map-reduce performance modeling, improvement of parallel sort algorithms, HDFS namespace expansion, integration with external resource management services, and more.

6 Conclusion

Innovation in leading Internet companies revolves around analyzing huge datasets. Modern grid technologies offer tools for building scalable and reliable web-scale data centers for this purpose. We surveyed the recent achievements in this multidisciplinary area, focusing on the open-source Hadoop suite. We reviewed the fundamentals of the Hadoop technology, and focused on selected research projects in distributed computing, ZooKeeper and BookKeeper, which emerged around it. Finally, we outlined some open research problems that await resolution to support next-generation web data center infrastructure.

Acknowledgments

This survey is partially based on Ajay Anand's talk at the Usenix '2008 conference [10]. I thank Brian Cooper, Flavio Junqueira, Christopher Olston and Benjamin Reed for providing many useful inputs.

References

[1] Google container data center tour. http://www.youtube.com/watch?v=zRwPSFpLX8I.

[2] Hadoop on Demand. http://hadoop.apache.org/core/docs/current/hod_user_guide.html.

[3] Hadoop Research Project Suggestions. http://wiki.apache.org/hadoop/ProjectSuggestions#research_projects.

[4] Technologies powered by Hadoop. http://wiki.apache.org/hadoop/PoweredBy.

[5] The Apache Hadoop Project. http://hadoop.apache.org.

[6] The HBase Project. http://hadoop.apache.org/hbase.

[7] The Hive Project. http://hadoop.apache.org/hive/.

[8] The Pig Project. http://hadoop.apache.org/pig.

[9] The Zookeeper Project. http://hadoop.apache.org/zookeeper.

[10] A. Anand. Using Hadoop for Webscale Computing. *ACM Usenix*, 2008. http://www.usenix.org/events/usenix08/tech/slides/anand.pdf.

[11] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. Assoc. Comput. Mach.*, 42:124–142, 1995.

[12] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. *OSDI*, 2006.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *OSDI*, 2006.

[14] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *VLDB*, 2008.

[15] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4), 1998.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, S. Sivasubramanian A. Pilchin, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *SOSP*, 2007. http://aws.amazon.com/s3/.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *SOSP*, 2003.

[18] S. Gilbert, N. Lynch, and A. Shvatrsman. RAMBO II: Rapidly Reconfigurable Atomic Memory for Dynamic Networks. *DSN*, 2003.

[19] J. Gray. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. *EuroSys*, 2007.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[22] F. Junqueira and B. Reed. A simple totally ordered broadcast protocol. *Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2008.

[23] Nancy A. Lynch. *Distributed Algorithms (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 2002.

[24] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. Reed. Interactive Analysis of Web-Scale Data. *CIDR*, 2009.

[25] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. *SIGMOD*, 2008.

[26] M. Zaharia. Hadoop Fair Scheduler. http://developer.yahoo.net/blogs/hadoop/FairSharePres.ppt.

[27] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. *OSDI*, 2008.

Cloud Computing Architecture and Application Programming

DISC'09 Tutorial, half day, Sept. 22nd 2009

Roger Barga Jose Bernabeu-Auban Dennis Gannon Christophe Poulain
Microsoft Corporation
Contact: barga@microsoft.com

Background

Over the past decade, scientific and engineering research via computing has emerged as the third pillar of the scientific process, complementing theory and experiment. Several studies have highlighted the importance of computational science as a critical enabler of scientific discovery and competitiveness in the physical and biological sciences, medicine and health care, and design and manufacturing. The ability to create rich, detailed models of natural and artificial phenomena and to process large volumes of experimental data, itself created by a new generation of scientific instruments that are themselves powered by computing, makes computing a universal intellectual amplifier, advancing all of science and engineering and powering the knowledge economy. This revolution has been enabled by the availability of inexpensive, powerful processors; low cost, large capacity storage devices; and flexible software tools, each driven by a vibrant consumer and industry marketplace.

The explosive growth of research computing systems has created major management, technical and fiscal challenges for both funding agencies and research universities. Typically, faculty members acquire research computing systems, usually small to medium (32–256 nodes) clusters, via research grants and contracts and departmental funds. This distributed acquisition of research computing and inadequate plans for long-term sustainability and technology refresh, mean that universities and funding agencies that support university research, are now struggling to create and maintain compute and data centers to house these systems and to operate and maintain them reliably, in energy-efficient, environmentally friendly ways. Moreover, university budget constraints make efficiency ever more necessary. A growing challenge is satisfying the ever rising demand for research computing and data management - the enabler of scientific discovery. Fortuitously, the emergence of cloud computing- software and services hosted by networks of commercial data centers and accessible over the Internet - offers a solution to this conundrum.

Cloud Computing

The explosive growth and rapid development of cloud services are driven by technology and business economics. Consolidating computing and storage in very large data centers creates economies of scale in facility design and construction, equipment acquisition and operations and maintenance that are not possible when the elements are distributed. However, the benefits of cloud services extend far beyond economies of scale.

First, optimized and consolidated facilities reduce total energy consumption, and they can be designed to exploit environmentally friendly and renewable energy sources. Second, cloud computing enables a "pay only for use" strategy where users bear no cost unless they use the cloud services, and then pay only for the number of service units consumed. Third, groups can deploy and expand services rapidly - in minutes, rather than the weeks or months needed to procure and install local infrastructure - to meet rising demand

or to address time-critical needs. Finally, the elasticity of cloud services means that time and computing are interchangeable - the user cost to use 10,000 processors for one hour is the same as using ten processors for 1,000 hours. This is a transformative equivalence; even individuals and small companies can exploit computing resources at a scale heretofore accessible only to large companies and governments.

By outsourcing computing, data management and business intelligence services to cloud software plus services providers, businesses are increasing operational efficiencies and decreasing costs, allowing them to focus on their core competencies. Similar opportunities exist in academic and research computing, but these opportunities are not being exploited.

DISC'09 Tutorial Description

The goal of this tutorial is to demonstrate how clouds can augment traditional supercomputing by expanding access to data and tools to a broader community of users than are currently served by the conventional HPC centers. Supercomputers provide the capability to conduct massive simulation and analysis computations for a few users at a time. They are not designed for on-demand access by hundreds or thousands of simultaneous users. In addition, supercomputers are not configurable by their users. Thus supercomputer applications must be modified and, in some cases, refactored as hardware and systems software is upgraded. Clouds offer the ability for each user to customize the execution environment, and to archive that customization for future use independently of the infrastructure's lifecycle. Currently, a number of publically accessible computational platforms provide instant access to cloud-hosted services such as web search, maps, photo galleries and social networks. There are now hundreds of cloud-based services we use in our everyday life and we are starting to see some of them also touch our scientific lives. For example, Google and Live maps have been used to gain insight from geo-distributed sensor data and the Sloan Digital Sky Survey and the SkyServer have provided scientific data and tools to thousands of astronomy users. We are now at an important inflection point in the capability of cloud computing to serve the research community. Not only has the total capacity of the commercial data centers exceeded that of supercomputing centers, we now have the software infrastructure in place to allow anybody to build scalable scientific services for broad classes of users, without having to deploy, maintain and upgrade dedicated and expensive compute and data servers. This tutorial will introduce the attendees to this new technology. This tutorial will be of value to those interested in exposing data and services to a broader audience of users without incurring the costs of acquiring and maintaining scalable infrastructure. This tutorial will introduce the attendees to the key concepts and technologies used to build and deploy scientific data analysis applications on cloud platforms. The tutorial begins with general concepts of data center architecture including the use of virtualization; the role of low power, multicore and packaging; and web service architectures. We will look at the cloud storage models with a detailed look at the Azure XStore and a brief look at Google's BigTable and GFS. We will then focus on models of application programming. We will describe both commercial and open source tools for "map reduce" computation including Hadoop and Dryad and workflow tools for orchestrating remote data services. Following this we will examine cloud application frameworks by looking at Google's App Engine and Microsoft Azure. Throughout the tutorial we will use scientific examples to illustrate the potential applications. The tutorial concludes with a view of the future for the cloud in science.