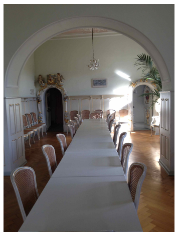
Distributed Computing Column 53

Dagstuhl Seminar Review: Consistency in Distributed Systems

Jennifer L. Welch
Department of Computer Science and Engineering
Texas A&M University, College Station, TX 77843-3112, USA
welch@cse.tamu.edu

This issue's column is devoted to a detailed review of a fascinating Dagstuhl seminar on consistency in distributed systems, held in February 2013, which I was fortunate enough to attend. This seminar brought together researchers and practitioners in distributed systems, programming languages, databases and concurrent programming to make progress in understanding the trade-offs between data consistency, availability and fault-tolerance. In addition to the tutorials, technical presentations, and breakout sessions reported on below, attendees got the opportunity to participate in a chilly but rewarding trip to the city of Trier, containing the Porta Nigra (the Black Gate, the largest Roman city gate north of the Alps) and the Trier Cathedral (home to the Holy Tunic).



Schloss Dagstuhl dining room

Porta Nigra in Trier, Germany

In the square near Porta Nigra

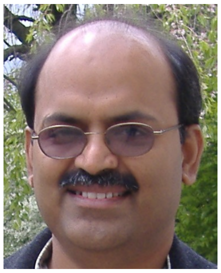
Many thanks to the seminar organizers, Bettina, Rama, André, and Marc as well as all the attendees, for their contribution!

Call for contributions: I welcome suggestions for material to include in this column, including news, reviews, open problems, tutorials and surveys, either exposing the community to new and interesting topics, or providing new insight on well-studied topics by organizing them in new ways.

Dagstuhl Seminar Review: Consistency in Distributed Systems

Bettina Kemme
McGill University
Montreal, Quebec, Canada
kemme@cs.mcgill.ca

G. Ramalingam
Microsoft Research
Bangalore, India
grama@microsoft.com



André Schiper
Ecole polytechnique fédérale de Lausanne
Lausanne, Switzerland
andre.schiper@epfl.ch

Marc Shapiro
INRIA
France
marc.shapiro@acm.org

1 Introduction

In distributed systems, there exists a fundamental trade-off between data consistency, availability, and the ability to tolerate failures. This trade-off has significant implications on the design of the entire distributed computing infrastructure such as storage systems, compilers and runtimes, application development frameworks and programming languages. Unfortunately, it also has significant, and poorly understood, implications for the designers and developers of end applications. As distributed computing become mainstream, we need to enable programmers who are not experts to build and understand distributed applications.

A seminar on "Consistency in Distributed Systems" was held from 18th to 22nd, February, 2013 at Dagstuhl (number 13081). This seminar brought together researchers and practitioners in the areas of distributed systems, programming languages, databases and concurrent programming, to make progress towards the above-mentioned goal. Specifically, the aim was to understand lessons learnt in building scalable and correct distributed systems, the design patterns that have emerged, and explore opportunities for distilling these into programming methodologies, programming tools, and languages to help make distributed computing easier and more accessible.

We may classify current approaches to deal with the challenges of building distributed applications into the following two categories:

- Strong Consistency and Transactions: Strong consistency means that shared state behaves like on a centralised system, and programs (and users) cannot observe any anomalies caused by concurrent execution, distribution, or failures. From a correctness perspective, this is a most desirable property. For instance, a database management system protects the integrity of shared state with transactions, which provide the so-called ACID guarantees: atomicity (all-or-nothing), consistency (no transaction in isolation violates database integrity), isolation (intermediate states of a transaction cannot be observed by another one), and durability (a transaction's effects are visible to all later ones).

- Weak Consistency: Unfortunately strong consistency severely impacts performance and availability [1, 5]. As applications executing in the cloud serve larger workloads, providing the abstraction of a single shared state becomes increasingly difficult. Scaling requires idioms such as replication and partitioning, for which strongly-consistent protocols such as 2-Phase Commit are expensive and hard to scale. Thus, contemporary cloud-based storage systems, such as Amazon's Dynamo or Windows Azure Tables, provide only provide weak forms of consistency (such as eventual consistency) across replicas or partitions. Weakly consistent systems permit *anomalous* reads, which complicates reasoning about correctness. For example, application designers must now ascertain if the application can tolerate stale reads and/or delayed updates. More parallelism allows better performance at lower cost, but at the cost of high complexity for the application programmer.

A number of approaches and tools have been developed for reasoning about concurrently-accessed shared mutable data. The concept of linearizability [11] has become the central correctness notion for concurrent data structures and libraries. This has led to significant advances in verification, testing and debugging methodologies and tools. Transactional memory provides a higher-level, less error-prone programming paradigm [10].

More recently, a number of principles have emerged for dealing with weak consistency. For example, if all operations in a program are monotonic, strong correctness guarantees can be provided without the use of expensive global synchronization. Similarly, certain data structures such as sets and sequences can be replicated in a correct way without synchronisation.

These developments illustrate the benefits of cross-fertilization of ideas between these different communities, focused on the topic of concurrency. We believe that such principled approaches will become increasingly critical to the design of scalable and correct distributed applications. The time is ripe for the development of new ideas by cross-fertilisation between the different research communities.

It is crucial for researchers from different communities working in this same space to meet and share ideas about what they believe are the right approaches to address these issues. The questions posed for the seminar include:

- Application writers are constantly having to make trade-offs between consistency and scalability. What kinds of tools and methodologies can we provide to help this decision? How does one understand the implications of a design choice?
- Weakly consistent systems are hard to design, test and debug. Do existing testing and debugging tools suffice for identifying and isolating bugs due to weak consistency?
- Can we formalize commonly desired (generic) correctness (or performance) properties?
- Can we build verification or testing tools to check that systems have these desired correctness properties?
- How do applications achieve the required properties, while ensuring adequate performance, in practice? What design patterns and idioms work well?
- To what degree can these properties be guaranteed by the platform (programming language, libraries, and runtime system)? What are the performance tradeoffs (when one moves the responsibility for correctness between the platform and application)?

In order to ensure a common understanding between the different research communities that the workshop brings together, the seminar started with a few tutorials from the perspective of each community. Other presentations focused on a specific piece of research or a research question. Participants brain-stormed on a specific issue during each of the two break-out sessions.

This report is a condensed version of the full report, available at `http://drops.dagstuhl.de/opus/volltexte/2013/4014/pdf/dagrep_v003_i002_p092_s13081.pdf`. The full report is the compilation of notes taken by the following participants: Marcos Aguilera, Carlos Baquero, Annette Bieniusa, Sebastian Burckhardt, Allen Clement, Mike Dodds, Amr El Abbadi, Alan Fekete, Carla Ferreira, Alexey Gotsman, Maurice Herlihy, Ricardo Jiménez-Peris, Petr Kuznetsov, Achour Mostefaoui, Ioannis Nikolakopoulos, Vivien Quéma, Kaushik Rajan, Noam Rinetzky, Luís Rodrigues Rodrigo Rodrigues, Nicholas Rutherford, Alex Shvartsman, Pierre Sutra, Doug Terry, Peter van Roy, Kapil Vaswani, Jennifer Welch, and Pawel Wojciechowski. The abstracts and slides of presentations are available at `http://www.dagstuhl.de/mat/index.en.phtml?13081`.

2 Tutorials

2.1 Geo-Replication in Data-Center Applications: Marcos Aguilera

Marcos's tutorial concerned multi-data center infrastructure supporting a web application or a number-crunching computation. Data is usually considered valuable, so geo-replication allows for disaster tolerance. It also provides availability and low latency access. A major challenge is that delays across replicas are significant.

The talk focused on geo-replication mechanisms, classified along two axes: when replicas are updated (synchronous vs. asynch replication) and what type of service is supported (read/write vs. state machine vs. transactions). He gave examples of protocols and systems that implement all combinations of these two features. He started with read/write synchronous replication, and gave a basic quorum construction. He refined the basic construction to obtain the ABD algorithm. Next he moved to synchronous state machine replication; the basic abstraction to implement this is consensus (e.g., Paxos and PBFT). He then moved on to synchronous replication with transaction support, where the basic approach is to broadcast all operations to all replicas. Locking can be problematic and a separate locking service helps. Moving on to asynchronous read/write replication raises issues of lost updates and of updates arriving at different replicas in different orders. Coping mechanisms include primary replicas, merge strategies, and vector timestamps. With asynchronous state machine replication, there is an advantage in using commutative operations since different replicas can apply updates in any order. Regarding asynchronous transaction replication, commutativity helps as does an isolation level called parallel snapshot isolation. He concluded with an overview of examples of several systems.

Discussion revolved around the relationship between transactions and invariants, identifying which techniques in the matrix are particularly good for geo-replication when replicas are far apart, and whether new consistency levels are required.

2.2 Cloud Storage Consistency Explained Through Baseball: Doug Terry

Doug spoke about the different consistency choices that a cloud system could offer to applications. For example, AWS S3 provides eventual consistency; AWS SimpleDB and Google AppEngine give a choice between eventual or strong; Yahoo! PNUTS offers a choice of eventual or strong consistency on each read; Cassandra has eventual or strong, but the choice must be made system wide; and Windows Azure provides only strong, except for a limited period (e.g., 15 minutes) if there is a major system failure. Many other consistency options have been proposed in research papers, but they not being used in practice. The goal of Doug's talk is to explain the usefulness of some intermediate forms. Doug is using a model of operations that are either read or arbitrary writes (these can be append, deposit, etc., not just obliterate the prior value with a new one).

Doug gave six favourite consistency options; he sees these, and combinations of them, as having real use in different situations. In all the consistency options, a read should never see inflight writes, nor should it return a value that was invented out of nowhere. Also conditions apply per item but they could instead be defined on the whole database.

- Strong: the read must see all previous writes.
- Eventual: the read must see the outcome of a subset of previous writes.
- Consistent prefix: the read must see the outcome of an initial sequence of the previous writes.
- Monotonic reads: each read within a session sees a subset of previous writes, and the subset seen increases (or stays equal) from one read to the next.
- Read My Writes ("RMW"): each read sees a subset of previous writes, and the subset must include all the writes issued in the reader's session before the read itself as issued.
- Bounded staleness: each read sees a subset of previous writes, and the subset must include all the writes that are reasonably old (for example this might be defined as those writes issued up to some time interval before the read was issued, though other definitions are based on the number of writes missed rather than the clock-time).

Prefix, bounded, monotonic, RMW properties are incomparable, measuring a property by the set of allowable return results; each of these is stronger than eventual and each is weaker than strong. There is a tradeoff: the properties differ in performance (mostly latency), and availability, as well as in consistency. Generally the stronger consistency comes with worse performance and lower availability.

Doug worked through some examples, showing how different participants in a baseball game would be able to usefully ask for reads with different consistency. For example, the scorekeeper can do RMW to the score variable, since he is only writer of this location. The umpire needs to read score locations and make binding decisions, so strong consistency is needed. All six guarantees appear in the examples, sometimes in combination. They would all be happy with strong consistency, but performance trade-offs drive weaker consistency (but it must be not too weak for the application's need).

Doug's conclusions are that replication schemes involve tradeoffs; consistency choices can benefit applications. We as researchers must provide better definitions, analysis and tools.

A discussion point was whether the system could automatically figure out which consistency level is best; probably not, but annotations or other mechanisms might help.

2.3 Database Consistency: Alan Fekete

Alan presented a tutorial on consistency from a database perspective, with the aim of helping people from other fields spot differences of perspective and terminology. The talk included the relationship of real-world state-changes to transactions, ACID guarantees, isolation levels, and where inconsistency might originate in a faulty system or program.

Comments from the audience touched on the fact that many databases "in the wild" run with read-commit isolation, rather than serializability; and that often there are consistency constraints that are not made explicit in the database, but instead are enforced in the program logic that interacts with the database. Topics of discussion included the registering of real-world events on the database; where control flow should be modeled; and how to deal with external consistency and with aborted transactions.

3 Technical Presentations

3.1 Making Geo-Replication Fast as Possible, Consistent when Necessary: Rodrigo Rodrigues

Rodrigo began his talk by pointing out that, in distributed systems, delays make users unhappy and the system achieves less revenue. Thus one aims to place replicas around the world so that people can find a close one (with low latency) when they want to do something. Rodrigo pointed to system designs with levels of hierarchy (the higher ones are cheaper to synchronise but have slower latency). Replication at different levels can coexist. The first replication level is a central server or master/primary. The second level has remote geo-replicated replicas. At level the, the replicas are in a CDN infrastructure. The fourth level of replication is P2P or hybrid CDNs (e.g., Akamai NetSession). A fifth level comes with replicas on mobile devices.

Rodrigo identified the challenge: to design distributed systems to be aware of this hierarchy. For example, in Facebook, or PNUTS, writes are done at a single master, and there are read-only mirror replicas. He mentioned prior work by Ladin et al. [12] and by Sovran et al. (Walter) [15]. His system's goal is to balance strong consistency (coming up with a total order on operations) with eventual (which is sometimes called causal) consistency, in which there is a partial order on operations. The aim can be summarised as being fast whenever possible, strongly-consistent when necessary.

Rodrigo proposed red-blue consistency, with some operations labelled red and others blue. Blue operations must commute with everything. The partial order is such that all red operations have a total order. The system can be implemented by a token (that circulates from one site to another) and only the token-holder can execute red operations.

In answer to the question, why not synchronise red with all operations (as is done in previous systems), Rodrigo said that one can implement the proposed scheme efficiently, and create tools to decide which operations to make red—any operation that doesn't universally commute must be red, thus slow.

Rodrigo suggested that one can redesign the application to split the operation into a generator (which computes what changes are needed) and a shadow (which applies the computed delta). Then we label as red all resulting operations which do not commute universally or which break invariants. The authors are working on how to automate the split of operations into generator and shadow, and how to label them properly (rather than manually, as so far). They did an evaluation on real applications, and found that many operations can be blue, and the red ones are invoked rarely. The red-blue approach gets huge latency improvements compared to a multisite strong consistent implementation.

During the discussion, Rodrigo made the following points: causal consistency is done with version vectors having one entry per datacentre; they want users to indicate invariants that should be maintained. They don't have a proof whether we can always find a shadow that commutes universally, though in examples so far it has generally been easy. If the application adds a new operation, we need to reanalyse everything that had been analysed previously (as well as analysing the new code).

3.2 Cloud Types and Revision Consistency: Sebastian Burckhardt

Sebastian Burckhardt presented "concurrent revisions" for cloud programming, and the TouchDevelop programming platform where it will be given to users to evaluate its effectiveness as a simpler way to address consistency in distributed concurrent programming. The talk began with a summary of concurrent revisions, a fork-join replicated state model for multicore programming which provides parallelism and preserves determinism, with replica conflicts addressed by type-specific merging. Cloud Types were then presented, an application of concurrent revisions to cloud computing applications, with the example of client-side replicas for mobile applications. TouchDevelop was presented as an existing platform for non-expert programmers to develop mobile applications, to which Cloud Types will be added.

During the discussion, it was brought up that there is a possible relationship of this work to persistent data structures [14]. When asked if they considered direct communication between devices, Sebastian replied that communication with the servers is important but that extra connections, between devices, can be also useful as a complement.

3.3 Semantics of Eventually Consistent Systems: Alexey Gotsman

The topic of Alexey's talk is verification of concurrent programs, which enables reasoning about the correctness of a program only having the specification of the used library in mind, and not the internals of its implementation. Examples of specification techniques includes strong and composable notions like linearizability, as well as weak and non-composable memory models (x86, C/C++). Given that processors and programming leanguages do not provide sequential consistency, a multiprocessor machine should in fact be treated as a distributed system.

In distributed systems, however, strong consistency criteria are often replaced with weak ones (to reach A and P in the CAP triad), such as eventual consistency. A popular definition of eventual consistency, given by Werner Vogels "If no new updates are made to the object, eventually all accesses will return the last updated value" [16] does not allow one to reason about scenarios in which updates never stop. There is a number of fixes of this for various setting and types of implementations (ruling out anomalies, preserving causality, restricting to conflict-free replicated data types or transactions). But there does not exist a declarative definition of the semantics of eventually consistent systems.

This work proposes a framework for declarative specification of consistency model. The system model considers a replicated service with full replication (every replica stores complete data), generic operations not limited to read/write, asynchronous replication scheme, link failures, but no replica crashes. The framework encompasses various existing conflict-resolution techniques, including timestamp last writer, high level commutativity, returning all conflicting values to users, and application-dependent resolution, as well as invariants that should be observed in the presence of ongoing updates, such as read your own writes and causal or FIFO ordering of operations. The framework addresses the conflict-resolution techniques via *data type specification* and invariants via *consistency axioms*. Consistency axioms capture the desired semantics of the system, such as "eventuality" (an operation cannot be invisible to infinitely many actions on the same object), or "causality" (all actions that happen before on the same object are visible). Altogether, this defines semantics of eventually consistent systems, and the paper validates the specifications via example abstract implementations.

There was some discussion as to whether the proposed definition of eventual consistency is not too weak actually to be useful.

3.4 Quantifying Inconsistency: The Transactional Way: Bettina Kemme

How likely is it that a real execution gives strong consistency even though the system only guarantees weaker consistency? Bettina's talk advocates the idea of *quantifying* inconsistency with respect to strong consistency (e.g., serializability). One example of such a quantification is to count the number of serialization *cycles* that are observed in executions of the system under a given workload. The approach looks promising because even if potential inconsistencies are known, it is not clear how often inconsistencies actually occur. This may help to answer questions like: when we move to cloud storage with a different consistency guarantee, how consistent will my existing application be?

The quantitative approach boils down to: (1) Deploy an application on a multitier platform. (2) Choose a level of isolation provided. (3) Run the application and count serialization anomalies, their types, etc. (4) Be efficient, do not slow down execution.

The approach was first applied to a traditional database system. A dependency graph is constructed in which cycles are sought; using properties of isolation levels allows some paths to be excluded. The second application of the method was cloud storage where transactions are not necessarily supported. In this case, an approximate graph is constructed which indicates, for instance, that either one of two orderings holds. Future work will include object-based anomalies, stale reads, and violations of monotonicity.

During the discussion, it emerged that the framework can be used to detect and identify failures and that the slowdown caused by finding cycles in the application engine was only 2-3%.

3.5 The Emperor's New Consistency – The Case Against Weak Consistency in Data Centers: Marcos Aguilera

Marcos began his talk by highlighting several data center storage systems that have adopted weak consistency models, in order to improve performance, availability, and ease of development. Examples include Yahoo! PNUTS, Amazon S2, Amazon SimpleDP, Microsoft Azure, and Dynamo. Users see stale data, but only sometimes, and the conventional wisdom is that users are tolerant.

Marcos argued first that the drawbacks of weak consistency are expanding due to sociological, legal, and technical reasons. As users increasingly pay for their apps, their tolerance for strange behavior is diminishing. With the increasing reliance on data center apps in areas such as banking and medical records, the issue of legal liability means that anomalous behavior can lead to monetary loss. Technically, the increasing number of layers and more integration among apps makes even more difficulties for programs than for humans.

Marcos' second argument was that the benefits of weak consistency are shrinking for technological reasons. First, network partitions will disappear, as their causes are being addressed both within and across data centers. Second, wide-area latency is shrinking. Third, the number of applications is growing relative to the number of storage systems, and so the argument that it is easier to build a weakly consistent storage system loses force because it makes it harder to develop apps. (On the other hand, some factors that would undermine his argument were given, including the possibility that people will just become inured to the problems raised by inconsistency.)

As a result of this double movement, in the future, drawbacks will outweigh benefits. Weak consistency creates challenges that will become harder to overcome, and solves problems that will be solved differently in the future.

There was a lively discussion throughout Marcos' talk, centered on the following points. Is stale data the defining characteristic of weak consistency (cf. PSI [15])? In fact, what is the right definition of weak consistency? The pressure of liability might not lead to better systems, just checklists. There was a debate as to whether partitions really will go away and latency reduce, and also whether strong consistency is really more expensive to provide than other conditions, such as Parallel Snapshot Isolation.

3.6 HAT, not CAP: Highly Available Transactions: Alan Fekete

Alan advocated a model that might be a sweet spot for storage systems for Internet-scale systems. He is assuming that partitions will *not* disappear. Many early systems (eg., BigTable, PNUTS, S3, Dynamo, MongoDB, Cassandra, Simple DB, Riak) offered scalability and availability but missed functionality expected in traditional DBMS platforms. Wouldn't it be nice to add more consistency, richer operations, and grouping of operations on multiple items? The focus of this talk was on ways to group operations on multiple items for Internet-scale storage systems.

It has been known at least since 1985 [7] that a system cannot provide always-available serializable transactions if the system can partition. What about providing ACID transactions with weaker Isolation? Serializability may be the ideal, but most single-site DBMS already don't ensure serializability; instead, read-committed is the default.

HAT is a useful model for programmers: a transaction is an arbitrary collection of accesses to arbitrary sets of read-write objects; its semantics is as strong as feasible while ensuring availability even when partitioned. Availability is clearly not possible if the client is partitioned away from its data. However, even in the presence of a partition, if a transaction can reach a replica of each item it needs, then the transaction should be able to commit.

The question was asked if this claim is with high probability. The answer is maybe, if you set your timeouts wrong. Alan and his coauthors haven't proved any theorems yet. They think this is what to aim for.

HAT transactions are all-or-nothing (atomic), causally consistent (including read-your-writes, monotonic reads, write-follows-reads), and provide an isolation level similar to read-committed and repeatable-reads. However, a read does not necessarily see the most recently committed change.

The question was asked whether read-committed forces updates to become visible. The answer is no: in read-committed, if a transaction observes an update, then the updating transaction has committed; there is no recency in the definition.

Alan and coauthors define the semantics with an approach inspired by [2], i.e., a graph of operations with different kinds of edges (e.g., write-read, write-write, read-write, happens-before), and restrictions on the sorts of cycles that can occur. Alan sketched a proof-of-concept implementation.

The discussion brought up the following points: There is similarity to partition-tolerant group communication systems, but with the addition fo transactions. A transaction can commit before a partition is reconciled but transactions on the other side of the partition will not see the changes. More work is needed to deal with the large space overhead of maintaining causality meta-data.

3.7 Scalable Transactional Consistency for Your Cloud Data: David Lomet

David began his talk by describing the dream of the user having transactions anywhere with data in the cloud, and not having to think about the state of the data. An example was given and discussion ensued with the point being that the world is easier when transactions are available. The economics of going to the cloud are compelling: cheap power, hardware bulk purchase, low land costs, etc. But transactions for data are guaranteed to exist on the same node (e.g., Microsoft, Amazon, Google) but only eventual consistency is guaranteed for multiple-node data. There is very limited support for transactions across the cloud. The problem is that every data source needs to be a two-phase commit (2PC) participant.

The new idea is to think of the data center as if it were an SMP/cluster, with high bandwidth and a highly reliable network interconnect. This is not the web or a federated system, so the CAP theorem does not necessarily apply. The intuition is we do not do 2PC with disk, which is an atomic page store, we put transactions on top of atomic record stores. Details of this new system, called Deuteronomy, were given, which separates transactions from data. A prototype has been built, tested, and experimented on. They can get millions of operations per second!

The discussion clarified a few points. The system is OLAP-focused but it could be used in a general-purpose way; however, one would need to think harder about performance. One has to enforce idempotence and manage the log locally. Locking is used, but only in the transaction component, not the data component. Paxos-style replication is used to deal with transaction component failure. Write bandwidth is the main problem for the transaction component.

3.8 Principles for Strong Eventual Consistency: Marc Shapiro

Marc Shapiro defined strong eventual consistency as “Correct replicas that have received the same set of updates (possibly in different orders) have equivalent states.” He then introduced conflict-free replicated data types (CRDT), monotonic semi-lattices, and discussed state-based CRDTs vs. operation-based CRDT. The main idea in CRDTs is to avoid relying on synchronization. Marc then explained what should be done to handle non-commutative operations. The idea is to extend the semantics to give a deterministic solution that guarantees convergence. CRDTs are used in several key-value stores, in geo-replicated systems (e.g., Walter [15]), etc.

Audience members mentioned that Bayou provides strong eventual consistency if replicas re-order updates themselves, and that Dynamo allows replicas to diverge while letting users resolve conflicts. During the discussion, it was clarified that operations are not necessarily idempotent and that causal delivery—ensured using vector clocks—is needed to apply a sequential specification. When asked whether everything could be done with CRDTs, Marc replied that, as Rodrigo Rodrigues pointed out in his talk, not everything can be done without synchronization.

3.9 Composing Lattices and CRDTs: Carlos Baquero

Carlos’ talk was a continuation of the preceding one. Carlos emphasized the language aspects of CRDTs using the state-based approach, order theory, and the notion of lattices. He presented a type hierarchy comprising the following types: set, poset, lattice, and lattice with bottom. He developed various examples, including lexographic ordering and the antichain. Carlos then introduced the notion of “inflation”, which ensures monotonically-advancing updates and gave some examples. He also explained how inflations can be sequentially composed.

The questions and answers during the talk established several interesting points, listed next. (1) It is an open question whether lattice compositions are universal. (2) Using the theory presented, it may be possible to track connections bewteen CRDTs. (3) It is still open as to what exactly can be proved on CRDTs, other than their being well-defined.

3.10 Swiftcloud: Geo-Replication Right to the Edge: Nuno Preguiça

The context of this talk is the latency experienced by the client when accessing the closest data-center. The trend is to increasingly run code inside clients to (1) improve latency, and (2) improve fault tolerance via disconnected operation. The basic problem is how to support data sharing in web application. To this end, a useful semantics offers (1) Writes that are atomic and mergeable, (2) Reads that support isolation levels, and (3) Transactions. When transactions are executed on the client, key design features include: CRDTs (conflict-free replicated data types), Asynchronous Replication, Multi-Version Server, Version-Vectors, and Client-Assisted Failover.

Questions asked by audience members led to several clarifications. Transactions are replayed at the data center, for the effects. Updates must be kept at the client until confirmed by the data center. Isolation levels are implemented using the dependency vector. Committing in disconnected mode can only be done in asynchronous mode. The size of the vector clock is the number of data centers. Causal consistency is achieved by executing at the data center only if dependencies are satisfied.

3.11 Applications for Geo-Replicated Systems — Where Are the Limits?: Annette Bieniusa

Annette began her talk by pointing out that as cloud storage has evolved, two emerging principles are the use of (1) object-oriented data stores, and (2) eventually consistent key-value stores. The stated goal is to make distributed programming as "simple" as concurrent programming (audience laughs). To this end, three application examples are studied in detail that use CRDTs.

The first example was a social network supporting the operations log in/out, post, send, view wall, manage friends, poll, and like. In response to a question, Annette said that the granularity of the CRDTs was chosen to be at the object level; this is a performance question, not a semantic one.

The second example was a file system. It uses a sequence CRDT for text files (supporting editing) and a simple register CRDT for non-text files. Also, it introduces a special recursive CRDT for directories. Experiments showed that the system performs very well if there is a scout on the client, but up to 20x slower when running with remote scouts in a nontrivial configuration. Discussions clarified several points: Everything was actually implemented, each subidrectory is its own map, there is a notification system to inform clients of new updates, and the overhead is about 33%.

The third example is web e-commerce, such as a bookstore. This example contained a number of challenges: limited resources, use of approximation to find the top-N, mixing small and big transactions, and the need for offline conflict resolution.

The conclusion mentioned several insights: (1) object abstractions are useful, (2) it is important to balance options: too many, and the user is overwhelmed; too few, and some things are impossible to implement, (3) many users had trouble with object creation, object deletion, and with operations involving a large number of objects.

There was an audience question relating to updates: what happens if you subscribe only to a subset of objects for updates, and those objects are updated along with others in a transaction? The answer is that clients still receive all notifications (even for unsubscribed objects), to preserve causal consistency. This was a discussion point as it raised concerns about the scalability of subscription mechanism for receiving updates.

3.12 Specifying, Reasoning About, Optimizing, and Implementing Atomic Data Services for Distributed Systems: Alexander A. Shvartsman

Sharing memory in networked systems is nicer than message passing; otherwise we wouldnt be here talking about consistency! This requires replication for availability and fault-tolerance; with replication comes the challenge of consistency. The easiest notion for users is the one-copy view (e.g., linearizability), but this is expensive. A cheaper guarantee is: a read sees some subset of the previous writes; but this is too hard to use. Between the "slow but correct" approach, e.g., linearizability, vs. "fast but wrong, i.e., weak consistency, the desirable goal is "fast enough and correct."

In dynamic systems new challenges arise due to replicas coming, going, and failing. Such systems must be reconfigurable, and various approaches explored the use of state machine replication, consensus, and group communication. The most efficient techniques appear to be ones based on the ABD approach [4], with extensions for dynamic settings. ABD and quorums provide consistency for small, transient changes; and for larger or more permanent changes the quorums are reconfigured.

The DynaStore [3] algorithm deals with dynamicity by building DAGs of reconfiguration possibilities. An ABD-style exploration looks for a DAG sink. If the number of updates is finite, a sink will be found. This is a promising approach that does not use consensus, although it places constraints on dynamicity. Rambo [9] revises ABD-style operations to make them dynamic, and provides a reconfiguration service that emits a consistent sequence of configurations with the help of consensus; obsolete configurations are garbage-collected in the background. Interestingly, (non-)termination of consensus does not impact the operations in progress. The abstract Rambo algorithm is rigorously proved correct. Several optimizations are proved by "simulation" (a proof technique that shows trace inclusion). Proof-of-concept implementations were methodically derived from specifications. A retargeting of Rambo for mobile settings was explored in GeoQuorums [8], where the reconfiguration does not resort to consensus due to the finite number of configurations.

3.13 Snapshot Isolation with Eventual Consistency: Douglas B. Terry

Doug presented a cloud storage system, the main goal of which is to provide multiple consistency levels, read-write transactions on replicated and partitioned data with snapshot isolation, and consistency-based SLAs. The system features a geo-replication scheme where write operations take place in a datacenter that includes primary and secondary replicas, while read operations can take place in remote secondary replicas. The client API includes standard get and put operations, as well as begin/end transactions and sessions. The latter two have different consistency guarantees as a parameter. The data are partitioned and there are primary and secondary servers per partition. The transactions implement snapshot isolation even across partitions. Version history is stored for every object as well as timestamps for the latest received write transaction (high-time) and the most recent discarded snapshot (low-time). The key issue is how to get the read timestamp, which depends on which consistency is chosen; Doug gave specific examples.

In the case of Bounded Staleness there was a question regarding the need for synchronized clocks. The lecturer answered that in practice it does not matter as values for bounded staleness are much bigger than the clock error. Furthermore, he analyzed the way to choose between available servers and how read-write transactions take place. The next question was what is the read set. Doug stated that transactions can specify and that by default all tablets will be read. Another comment was also that reads must block while a transaction is validating. The last question regarded the name of the system, which is Pilius.

3.14 ChainReaction: a Causal+ Consistent Datastore Based on Chain Replication: Luís Rodrigues

ChainReaction is a system for data store that provides Causal+ consistency using a variant of chain replication. The servers are organized in a one-hop distributed hash table with different objects spanning different parts of the chain. The changes are requested of a proxy and propagate from the head of the respective chain to the tail. The reads are distributed along the chain to reduce the tail bottleneck, weakening though the consistency. The last node that the client read from is kept in metadata. In answer to a question, Luís explaned that the worst-case size of the metadata is one entry per object; however, entries are removed by a garbage collection process, and still their size is much less than competitive systems like COPS.

The experimental results showed that the performance is similar to the competition in balanced read and write operations and it gets much better the fewer the writes are. The next question was what can be done for the write operations to be optimized. Lus explained that this is done by returning the result before the change propagates to the tail. The last question was if the value of the metadata associated with a chain replica can be of any value. The answer was anything between 0 and the end.

3.15 Consistently Spanning the Globe for Fault-tolerance: Amr el-Abbadi

Amr gave an overview of the evolution and the recent situation of data management in the cloud, starting with the fact that databases became a scalability bottleneck around 2000. The first question to the speaker at that point was if he agrees that databases do not scale. He agreed and particularly mentioned that they only scale up as RDBMS achieve ACID and transactions in a single node. However they do not scale out as the key-value stores do.

NoSQL stores was the first wave of solutions for the scalability problem. However, Amr motivated that it would be nice if we could have high-level abstractions like "joins" and transactions and that is what we can expect as the new wave of solutions. The speaker presented different approaches to splitting RDBMS systems onto multiple nodes, both static (ElasTras, SQL Azure, Megastore) and dynamic. Then he presented solutions for elasticity and fault tolerance via replication and how these solutions can be classified according to their consistency guarantees.

3.16 Consistency Without Consensus and Linearizable Resilient Data Types: Kaushik Rajan

Kaushik began the talk with a simple shared shopping cart example and observed that it is important for the multiple versions of the cart are replicas of each other but it is frequently infeasible to run consensus to ensure consistency across replicas. Given these competing desires, he addressed the challenge of identifying when it is (im)possible to construct linearizable replicated data types (LRDT).

Kaushik then identified two key properties of operations on a specific data type: commutativity (intuitively $S+a+b = S+b+a$) and nullification (intuitively $S+a+b = S+b$). Kaushik explained that if at least one of these properties holds for every pair of update operations, then a LRDT is possible and impossible if there exists a pair of update operations for which neither property holds.

The positive construction relies on generalized lattice agreement. In lattice agreement, each replica may propose a value and when it does so waits for a majority of replicas to respond. Once a majority of replicas has responded, the value is accepted. The key to successfully building LRDTs using lattice agreement is to run an infinite sequence of instances of the protocol and ensure that subsequent executions of the protocol always return a superset of the operations returned by the previous instance.

Kaushik concluded the talk by presenting a graph data type implemented with lattice agreement. An interesting point of this datatype is that if the operations *AddEdge* and *RemoveVertex* are both included in the supported operations, then an LRDT graph is not possible; if either operation is removed then linearizability is achievable. This final point generated a fair bit of discussion.

3.17 ACID and Modularity in the Cloud: Liuba Shrira

Liuba's talk highlighted the key tension surrounding transactions in cloud services: while they make life easy for application developers to reason about their system, efficiently supporting ACID transactions in multi-writer cloud services is non-trivial. At the core of this difficulty is allowing for disconnected operation and transactions to be issued/executed at different machines.

Liuba identified four basic strategies for implementing transactions: forcing serial execution, pessimistic concurrency control, optimistic concurrency control, and type-specific concurrency control. She observed that while type-specific techniques were popular topics of conversation in the 80s and early 90s, the techniques were not generally adopted for one simple reason: conventional wisdom says that concurrency control mechanisms must live in the concurrency control engine of the database. Given the extreme efforts that database engineers go to to ensure that the database performs well, inserting application-specific code into the finely optimized database engine was a non-starter.

In response to this tension, Liuba proposed a tiered architecture where type-specific transaction coordination is handled at client devices while all low-level (pessimistic) concurrency control is handled only at the servers. The two keys to Exo are client caches and reservations.

Clients in Exo are treated as caches, locally executing transactions which are pushed to the server at the next opportunity. The key step that allows these transactions to commit locally at the clients is reservations—essentially escrow portions of the object stored at the server that clients are able to acquire in advance. As long as a client has a reservation, it can perform and commit local transactions against the (portion of) the object managed by the reservation. The locally committed transactions are then guaranteed to be non-conflicting when they are reported back to the server (provided that the reservation is returned before it expires).

The Exo system demonstrates that type-specific concurrency control can be implemented outside of the optimized concurrency control mechanisms. Of specific importance in the context of modern systems is that Exo-leasing converts server-side concurrency control to client-side concurrency control, allowing for efficient eventually consistent systems.

3.18 Conditions for Strong Synchronization in Concurrent Data Types: Maged Michael

Maged started by addressing the problem of idempotent work stealing. In this problem one needs to maintain a data structure where available tasks are inserted and then removed by workers. Workers may steal tasks from another worker and, in the idempotent version of the problem, it is actually fine if two workers end up performing the same task. The work addressed the specific question of whether there are algorithms that do not require the task owner to execute expensive store-load fences or atomic operations. The question is answered in the affirmatives, and the talk briefly addressed algorithms that only require the use of CAS in the steal method (for different policies, such as LIFO, FIFO, and double-ended). In response to a question, Maged noted that idempotent work stealing may only be applied in some settings.

The speaker then proceeded to address the more general problem of characterizing what problems cannot be solved without strong synchronization. In this context, the notion of Strong Non-Commutativity (SNC) was defined and it has been shown that problems with such characteristics require the use of strong synchronization. The talk discussed some ways to circumvent this limitation, such as changing the API of the data structure, or using semantics to build idempotent types, such that the operations do not exhibit SNC. Audience members discussed different ways the designer could convert an SNC-API to a non-SNC API.

3.19 Commutativity, Inversion, and Other Stories of Consistency and Betrayal: Maurice Herlihy

Maurice began by making a brief historical overview of how some of the consistency, performance, and fault-tolerant concerns that appear in concurrent applications have been addressed by the distributed and multicore programming communities. Then the talk highlighted the performance limitations that can result from using software transactions to build concurrent programs that only consider read-write objects. This was illustrated by a simple object that returns unique ids (not necessarily consecutive). If this object is implemented using a read-write shared variable, transactions will encounter an unique id conflict. On the other hand, if this is exposed as an object that offers a commutative "getId" operation, the same transactions may not conflict. This reasoning leads to the conclusion that the entanglement between thread-level synchronization and transaction-level synchronization kills concurrency.

Using this motivation, the talk advocated the use of an hierarchical approach, where thread-level concurrency could be implemented by fine-grain, optimized, low-level mechanisms and exposed as a black box to the transaction-level concurrency control mechanisms. Under this model, transaction recovery needs to be based on an operation log, and be implemented by applying the operations' inverses. That talk also addressed the problem of supporting partial rollback and the most suitable abstractions for that purpose, conjecturing that checkpoints and rollback to a given checkpoint could be a suitable alternative to nested transactions (an abstraction that was presented as "widely admired but not widely implemented").

The discussion brought up several topics. (1) It was pointed out that there are similarities between the proposed approach and multi-level concurrency control schemes designed for databases. (2) It can be difficult to derive appropriate inverse operations to support recovery, in response to whcih Maurice clarified that in Scala, closures were used for this purpose. (3) Without additional structure, doing partial recovery based on arbitrary checkpoints could result in code as hard to understand as code using goto's. Maurice agreed that this was still a largely unexplored territory, and that structured approaches would need to be designed to take advantage of this apporach. (4) Some of the code used to implement concurrent objects might be redundant when considering that the object was going to be accessed in the context of transactions. Maurice answered that it was a reasonable price to pay for treating these library objects as black boxes.

3.20 Concurrent Data Representation Synthesis: Mooly Sagiv

Mooly presented a technique for synthesising fine-grained concurrent data-structures from high-level relational specifications. He began by observing that concurrent data structures are often used incorrectly in composite structures. He cited his paper from OOPSLA, which found that 38% of presumed linearizable algorithms in real code were in fact not linearizible. He suggested that a common failure was to use sequences of atomic operations and expect the resulting structure to be linearizable. The aim of Sagiv's approach is to derive composite data structures automatically out of linearizable container structures and a relational specification language. His target is low-level concurrent structures in Linux and similar. For a running example, he examined a Linux filesystem.

Sagiv's language, called RelScala, is translated down into Scala code. Data in Sagiv's approach is represented by a relation, combined with a DAG. The DAG is a high-level shape descriptor, used to represent the structure of the data in memory. Edges in the DAG correspond to sub-relations. For example, in the file-system, one edge accesses the `fs` portion of the relation, while another accesses the inuse portion. Sagiv's tool uses the DAG to automatically synthesize a data structure built out of primitive containers, together with methods for accessing the structure, where the methods rely on two-phase locking for concurrency control. In answer to a question, it was explained that the DAG encodes multiple paths to a given node because applications such as linux often feature multiple traversals of the data. Sagiv described two modes of his approach: the user can define the DAG by hand, or the Autotuner tool can test many possible DAGs, primitive containers, lock placements, and lock implementations, and determine the best combination empirically given a particular workload.

Sagiv presented some performance results. One counter-intuitive result was that a representation with sharing performed better on a sequential processor, whereas copying worked better on a multiprocessor. Sagiv speculated that this resulted from interaction with the cache.

Discussion centered on why this approach is better than using a database system: the reason is that this approach would be faster by tuning for particular workloads, as general databases cannot control the workload, and thus cannot specialize. Also, it is not clear if two-phase locking would be sufficient for an application such as Linux, but Linux does lots of things that this system cannot currently handle.

3.21 Distributed Unification as a Basis for Transparently Managing Consistency and Replication in Distributed Systems: Peter van Roy

Peter presented a model called deterministic data-flow programming, discussed the unification algorithm which drives this model, and drew connections to CRDT data-types for replication and consistency. The deterministic model is a form of concurrent functional programming. In it, variables can only be assigned once, and synchronization is achieved by forcing threads to wait for variables to be assigned. Peter presented this model in the context of the Oz multi-paradigm language.

Much of Peter's talk discussed the unification algorithm underlying the deterministic data-flow programming model. The unification algorithm is a constraint solver for certain kinds of equality constraints. Peter first discussed a sequential algorithm based on rational trees (trees with back edges). He presented a set of operational semantics rules defining this algorithm, then observed that the algorithm could be distributed by changing only one rule: Bind. Doing this results in a distributed unification algorithm. At the end of the talk, Peter discussed adapting this algorithm to CRDT data-types.

Marc Shapiro asked whether concurrent binding would need to search all the replicas. Peter said there exists a master node for each variable, which controls synchronisation. Marc said that this property does not hold for CRDTs, and wondered what the connection might be. Peter responded that his aim was to remove synchronization. Another participant asked whether the unification algorithm could be seen as movement up a lattice. Peter agreed with this, and observed that the algorithm could work with any monotonic process, for example adding edges to a graph.

3.22 Time Bounds for Shared Objects in Partially Synchronous Systems: Jennifer Welch

Jennifer presented several lower bound results on the cost of building atomic shared data structures in a partially-synchronous system. The focus was on objects of arbitrary type, with axiomatic specifications of the operations. This work extends previous work on the difference between sequential consistency and linearizability. Jennifer started with classical results on time complexity (lower and upper bounds) to execute operations on a logically shared atomic memory. Then, she presented new results, which improve the lower bounds on elapsed time for executing operations on a linearizable object (such as a queue or a stack). The proof technique is the classical shifting technique (indistinguishably argument) and an extension of the classical technique to allow larger lower bounds. The proposed bounds are tight or almost tight in many cases. The new algorithms split operations into accessors (read), mutators (write), or both (read-modify-write). This talk ends with several open problems: How to tighten gaps between lower and upper bounds? Is it possible to consider clock drift, failures, churn, etc., in the results? How to extend those results to cover other consistency criteria?

Hagit Attiya pointed out that those results refine the CAP impossibility result.

3.23 Reduction Theorems for Proving Serializability with Application to RCU-Based Synchronization: Hagit Attiya

Hagit presented a reduction theorem for proving serializability, with application to RCU-based synchronization. The talk started with the core idea of sequential reduction: under certain assumptions, one can show that if property P holds in sequential executions, then P holds in all executions. Hagit showed that this reasoning is correct for local locking policies (e.g., tree locking or two-phase locking), i.e., policies that do not employ a centralized concurrency control mechanism. Consequently, for any program M respecting a local locking policy, if M maintains its invariants during all sequential executions, then M maintains its invariants during all interleaved executions. The core of the talk was the reduction theorem. The proof of this theorem makes use of a classical indistinguishably argument.

RCU (for Read-Copy-Update) is a mechanism that allows read-only transactions to read data, even while they are locked for update. Linux developers intensively use RCU-based synchronization. Hagit pointed that this mechanism is not well understood; for instance scan operations in the presence of concurrent updates. She presented work in progress that aims at applying the above reduction theorem to RCU-based synchronization. The idea is to apply the theorem to sub-executions which contain only updates, then to superimpose individual steps of the read-only operations.

At the end of the talk, Bernadette Charron-Bost asked how the reduction theorem relates to Lipton's theorem [13].

3.24 Abstractions for Transactional Memory: Noam Rinetzky

Noam presented a technique called observational refinement for decomposing correctness proofs for Transactional Memory (TM) algorithms implementing opacity. With observational refinement, all possible views of an implementation are contained in the set of possible views admitted by the specification. In this setting, a view is the restriction of an execution history to a thread. Specifically, under opacity every history has an equivalent sequential history, including aborted transactions, such that real-time order is preserved. Noam then introduced a programming language where global variables are only accessed outside atomic blocks, atomic variables only inside atomic blocks. For this language, he sketched a proof of soundness and completeness for observational refinement with respect to opacity based on well-formed traces.

Discussion clarified that, using this result, in order to prove correctness of a concrete TM implementation, it suffices to show that the language implements opacity. Also, the language definition allows aborted transactions to issue side-effects by modifying local state, similar to “nested top actions” in database systems. The fact that Hardware Transactional Memory systems are implementing these semantics underlines the relevance of this model.

3.25 Idempotent Transactional Workflow: G. Ramalingam

Rama’s talk focused on a decentralised technique for realizing idempotent transactional workflows over partitioned data. Data partitioning is commonly used to achieve horizontal scaleout. Applications can potentially leave persistent data in an inconsistent state if the applications fail in the middle. This problem is particularly acute when the persistent data is partitioned. ACID transactions are one solution to the problem of ensuring consistency of persistent data in the presence of application (or transaction) failures. However, when data is partitioned, this requires the use of distributed transactions, which can be a performance concern.

The talk observed that transactional workflows are a common and useful idiom in applications that work with partitioned data. In its simplest form, a workflow is a sequential fault-tolerant composition of (ACID) transactions. These workflows are often required to be idempotent. An idempotent transactional workflow was presented as a useful language construct. The talk described a decentralized implementation technique for implementing such workflows without using consensus or any equivalent coordination across the different partitions. This approach can be extended with compensating actions, automatic retry, and checkpointing.

The discussion points included the following. If the first transaction commits and the second des not, then the first one needs to be rolled back and thus the programmer must provide compensations. The proposed approach handles more cases than that of transaction chopping. This corresponds to the multi-level transaction model of the 1990s, which had compensations. A scheduler is only needed for performance, while correctness is guaranteed by the model. The programmer is responsible for guaranteeing global uniqueness of ids.

3.26 BubbleStorm: Replication, Updates and Consistency in Rendezvous Information Systems: Alejandro Buchmann and Robert Rehner

BubbleStorm is a peer-to-peer system that organizes its peers probabilistically in order to provide different forms of document management, including publish/subscribe and document query, in an environment with high churn. BubbleStorm uses bubbles, i.e., range-limited flooding, in its routing algorithm. Publication bubbles must intersect with query bubbles, which is guaranteed with high probability through the topology management. To manage churn, the system relaxes consistency of its replicated nodes.

BubbleCast is the algorithm used to build the search trees. It stores in non-persistent fashion queries, events, notifications, position updates, and caches. Maintainer-based replication is organized by a manager in a storage pool. When a manager leaves, data is flushed or destroyed. Collective replication is durable and performed by a random set of nodes. Information is flooded using Lamport clocks for consistency. There is a flexible evaluation framework for simulation and deployment. They implemented and demoed a first-person space shooter game based on a “vision range.”

Many topics arose during the discussion. The difference from quorum systems is in the placement of replicas. All the lower layers—event scheduler, network, communication—can be used as building blocks and been used in many student projects. There are visualization and statistics interfaces. Inconsistencies could be measured with post-processing, with a testbed application that uses timestamps ina database; a lot of information is gathered in various parts of the system, inluding the simulator, which used for statistics. The analysis tool for communication patterns has a clean interface so it should be possible to use it to analyze other systems. Finally, G-Lab is a German system similar to PlanetLab, but more stable.

4 Breakout Sessions

4.1 Distributed Applications

Participants were divided into four breakout groups. The common topic was the consistency levels required in a distributed application, either a multiplayer online game or an e-commerce application. All groups selected the games, since they typically show a great deal of variety of interaction and synchronization patterns.

Group 1 Group members Annette Bieniusa and Yiannis Nikolakopoulos discussed a massive multiplayer online game, in which groups of users travel a virtual world performing various tasks, such as looking for weapons, killing monsters, etc. A player holds replicas of immutable objects, describing the static world (trees, buildings, etc.), and mutable objects such as players, monsters, weapons, gifts, etc. Object attributes include position, access order, and ownership.

A player holds the master copy of its own coordinates, and needs to see only those players that are located in the immediate vicinity. An object or field is assigned a specific consistency level; it may change as the game evolves. For example, a player may observe another one that is sufficiently close (in space or time).

These requirements lead to the monotonic reads for writers and bounded staleness for read-only replicas. Consider the special case of fighters against monsters: since it is generally not important who hit the monster first, this commutativity the use of CRDTs.

A player owns items like (virtual) money, weapons, etc. These should be persistent, and transfer of ownership must be transactional, e.g., using two-phase-commitment and conflict detection. An interesting issue is picking up items: if two players try to pick up the same item at the same time, then normally the closer player wins (bounded staleness).

Group 2 André Schiper summarized the discussion in his group. The discussion was focused on the design of a game prototype developed by Alejandro Buchmann and his students (see their talk later in the workshop). It is a P2P multiplayer shooter game. Each player has a sphere of visibility. Another player inside my visibility sphere can interact directly with me, requiring strong consistency or bounded staleness. Eventual consistency suffices for players outside my sphere since we do not interact. As players move, the contents of a sphere changes dynamically. Players from different teams may interact by exchanging messages.

The solution to picking-up items is similar to Bettina's group, but, sometimes, strong consistency may be needed, not just bounded staleness.

Several design problems were discussed: How to combine different levels of consistency? How to simplify programming? Ideally, application programmers should not be burdened by consistency issues. They should be able to assume ideal (strong) consistency, but give criteria for relaxing consistency. The system should be able to switch between the different kinds of consistency.

Another issue is persistence. What should persist across sessions? For example, individual shots are not persistent, but the *results* of shooting definitely must be.

André pointed out that game state and what users observe are separate. Alejandro proposed to have certain criteria for changing consistency levels. The criteria could benefit from setting thresholds that would tell the system when to switch dynamically between different levels of consistency.

Group 3 G. Ramalingam reports for many games, weak consistency is sufficient. His group started with a simple game called Wordament, in which players try (in parallel) to write words using a set of letters. There is no interaction between the players. Each player sends messages with suggested words. The game establishes an ordering between messages from different users, and give feedback. The first player who identifies a word gets a score.

It would be nice to observe real-time order on operations. A conflict, such as multiple players finishing concurrently, can be resolved by giving them the same score. Note that strong consistency is easy to achieve, because everything is done on the server.

The second scenario is a game similar to “Angry Birds,” but with multiple players. If multiple players shoot the same bird, the first one to hit it gets the score. This can be resolved by a central server, as there is enough time to compute a non-ambiguous result (strong consistency).

The third game consists of users collaborating to solve a puzzle. Here, we need a state merge function, as changes done by different users might conflict. The application includes constraints, e.g., in sudoku, no digit is allowed to be used more then once. The game informs the user if its move was overridden; thus, there is no need for strong consistency.

Other techniques can be useful. For instance, a player may use Escrow to make reservations for future operations; for instance, a player may mark an area of the puzzle as his. If the reservation is successful, he can proceed under bounded staleness.

Group 4 Marc Shapiro reports that his group considered both games and e-commerce applications, which share some elements (think of virtual money, etc.). Games were considered more exciting, because state is more complicated and there is more interaction. Games may be easier, since correctness constraints are set by the designer, failures are acceptable, and anonymity is accepted.

The game design discussed was similar to existing commercial systems. The virtual world is divided into disjoint rooms, where all players in a same room are on the same server. Putting multiple users on the same server allows to achieve strong consistency cheaply; there is weak consistency between rooms. They also discussed fairness (e.g., players with shorter network round-trip-time can be slowed down) and functional features (e.g., what anomalies does the game tolerate?).

Alejandro concluded that it was interesting to see that there are so many different levels of consistency which are *not* necessarily the same in every game — different games require different levels of consistency.

4.2 Helping Application Developers Choose Consistency

The break-out groups were to answer the question (posed by Doug Terry): "What can the research community do to help application developers understand the consequences of choosing a particular consistency?"

Group 1 Members of Group 1 identified the basic characteristics of an application for which consistency is important. They listed the nature of operations (idempotence, commutativity), the atomicity of groups of operations, the ordering between operations, and the staleness of reads.

Marc Shapiro pointed out that checking some of these properties cannot be done locally, since they are inherently global.

Group 1 then listed several questions related to these properties. In particular, how to extract the above characteristics from the application, and how to capture design patterns developers use to build concurrent programs. Solutions include static analysis and the use of synthetic workloads. The results returned by these tests can be both quantitative (e.g., performance of some workload) and qualitative (e.g., executing a workload throws an exception).

The report closed with an observation by Alan Fekete: In the context of databases, several studies of the performance difference between consistency levels conclude that the difference is small, because the bottleneck is disk access.

Group 2 André Schiper listed several ideas studied by Group 2. The first one is that model checking may help understanding the consequences of concurrency. Another to look at design patterns promoting good programming practices for weakly-consistent applications.

Somebody pointed out that this is a non-issue: a developer should always first start with atomicity; then, if performance is not sufficient, think about choosing another consistency criterion.

Group 2 proposes that, to help with the development of concurrent program, programmers have to be trained with a non-strongly consistent API (e.g., Cassandra).

This leads to another question: When should a programmer make the decision of what consistency is needed, and how to introduce it into the program?

At the end of the presentation, André underlined that partitioning was out of scope, since it is well understood and extensively covered in other studies.

Group 3 This group observes that developers do not understand what a consistency level means. They always assume that APIs expose strongly consistent operations. Furthermore, a programmer should consider what is executable under weak consistency at the level of the specification, and *not* at the level of the implementation. An ideal programmer, and by extension an ideal system, would choose among different consistency criteria based on the specification. For instance, one could consider a program whose integrity properties would vary according to the consistency criterion employed in the storage system.

A request is to make storage systems more testable, by forcing rare consistency violations to occur.

Group 4 Group 4 started with the following analogy: "A developer may turn a knob to change the consistency level of her application. What should we tell to the developer ?" They listed several refinements of this discussion:

- For each position of the knob, is it possible to give useful information to the developer ?
- Can we build a tool that will tell the developer what will go wrong with her application when turning the knob ? Several persons in the audience pointed out that in most cases this is non-tractable.
- Are assertions enough to understand correctness of a concurrent program ?
- Can we tell if a program can be safely restarted ?

To address the above questions, the group discussed the properties of such a "magic" tool. It would vary three properties: consistency, availability and performance (throughput or response time). This tool would use semantic analysis of the application, with annotations from the developer, and typical workloads. Annotations are necessary to make the analysis tractable. It might work as a model checker, testing application invariants while varying consistency of the APIs used by the application.

References

[1] Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design. *Computer*, 45(2):37–42, February 2012.

[2] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Mass. Institute of Technology, Cambridge, MA, USA, March 1999. Appears also as MIT Technical Report MIT/LCS/TR-786.

[3] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *Journal of the ACM*, 58:7:1–7:32, April 2011.

[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[5] Eric Brewer. CAP twelve years later: How the "rules" have changed. *IEEE Computer*, 45(2): 23–29, February 2012.

[6] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Comp. Sc.*, 2010. Springer-Verlag. A 30-Year Perspective on Replication, Monte Verità, Ascona, Switzerland, November 2007.

[7] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[8] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Alexander A. Shvartsman, and Jennifer L. Welch. GeoQuorums: implementing atomic memory in mobile *ad hoc* networks. *Distributed Computing*, 18(2):125–155, 2005.

[9] S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, December 2010.

[10] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Int. Conf. on Comp. Arch. (ISCA)*, pages 289–300, San Diego CA, USA, May 1993.

[11] Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[12] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *Trans. on Computer Systems*, 10(4):360–391, November 1992.

[13] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[14] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

[15] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385–400, Cascais, Portugal, October 2011. Assoc. for Computing Machinery.

[16] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008.