Consistency in Distributed Systems

Bettina Kemme, Ganesan Ramalingam, André Schiper, Marc Shapiro, Kapil Vaswani

Report from Dagstuhl Seminar 13081

Consistency in Distributed Systems

Edited by

Bettina Kemme[1], Ganesan Ramalingam[2], André Schiper[3], Marc Shapiro[4], and Kapil Vaswani[5]

1 **McGill University – Montreal, CA, kemme@cs.mcgill.ca**
2 **Microsoft Research India – Bangalore, IN, grama@microsoft.com**
3 **EPFL – Lausanne, CH, Andre.Schiper@epfl.ch**
4 **INRIA & LIP6 – Paris, FR, Marc.Shapiro@acm.org**
5 **Microsoft Research India – Bangalore, IN, kapilv@microsoft.com**

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 13081 "Consistency in Distributed Systems."

Seminar 18.–22. February, 2013 – www.dagstuhl.de/13081
1998 ACM Subject Classification C.1.4 Parallel Architectures – Distributed architectures, C.2.4 Distributed Systems – Distributed databases, D.4.2 Storage Management – Distributed memories, E.1 Data Structures – Distributed data structures, F.1.2 Modes of Computation – Parallelism and concurrency, H.2.4 Database Management Systems – Distributed databases
Keywords and phrases Replication, Consistency, Strong Consistency, Weak Consistency, Distributed Systems, Distributed Algorithms
Digital Object Identifier 10.4230/DagRep.1.1.1

1 Executive Summary

Bettina Kemme
Ganesan Ramalingam
André Schiper
Marc Shapiro

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. This seminar brought together researchers and practitioners in the areas of distributed systems, programming languages, databases and concurrent programming, to make progress towards the abovmentioned 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,

Consistency in Distributed Systems, *Dagstuhl Reports*, Vol. 3, Issue 2, pp. 1–35
Editors: Bettina Kemme, Ganesan Ramalingam, André Schiper, Marc Shapiro, and Kapil Vaswani
Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 three 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.
- Principled Approaches to Consistency: 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].
- Principles for weak consistency: 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.

Goals

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 presented 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

This report is the compilation of notes taken by several note-takers, rotating at each session. The majority of the participants served as scribe for some session.

It will be helpful to refer to the abstracts and slides of the different presentations, which are available at http://www.dagstuhl.de/mat/index.en.phtml?13081.

2 Table of Contents

3 Tutorials

3.1 Tutorial on geo-replication in data-centre applications: Marcos Aguilera

Carla Ferreira and Rodrigo Rodrigues

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.AguileraMarcos2.Slides.pdf

The basic scenario is a 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 focuses on geo-replication mechanims, 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). We suggest the reader to refer to the classification table, included in his slides.

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.

Doug Terry inquired why not mention 2PC at this point. Marcos replied it is more powerful and therefore could be used to implement these simpler abstractions, but he would leave the more powerful constructs to a later point in the talk. He refined the basic construction to obtain the ABD algorithm. Then moved to synchronous state machine replication. The basic abstraction to implement this is consensus and referred to two existing consensus protocols (Paxos and PBFT).

Doug questioned the difference between consensus and 2PC locks. Marcos replied that in consensus you don't have to worry about locks. Consensus is implementing an abstract distributed lock manager. Then moved on to synchronous replication with transaction support. Basic approach is to broadcast all operations to all replicas, locking can be problematic and a separate locking service helps.

A possible technique is deferred updates: execute transactions locally and use total order broadcast to force all replicas to execute all ops in the same order. Another possibility is to use state machine replication to geo-replicate storage servers and 2PC upon commit. (Several people asked for clarifications.)

Moving on to asynch R/W replication, this raises issues of lost updates and of updates arriving at different replicas in different orders. To address the latter, you can have a primary replica that is the only one that accepts updates, you can use merge strategies like last writer wins or an application-specific merge function. He spoke about how to detect concurrent writes using vector timestamps, which raised lots of questions regarding why such detection matters when you are doing blind writes.

Bettina then gave an example where such detection mattered: if one replica adds 2 to x and another replica adds also 2 to x, you want to keep both updates to x. Marcos commented that, because operations are not grouped together (as in transactions), one might get unexpected results.

At this point Marc Shapiro commented that transactions are a proxy for invariants. Marcos stated that these techniques are not good for maintaining invariants, specially for black-box invariants. Then with state machine asynch, there is an advantage in using

commutative operations since different replicas can apply updates in any order.

Regarding asynch transaction replication, commutativity also helps and you arrive to an isolation level called parallel snapshot isolation. He concluded with an overview of examples of several systems.

Alan Fekete pointed out that the talk was about replication but not so much about geo-replication, where replicas are far apart. Marcos concurred and explain a series of challenged with geo-replication. André Schiper asked which techniques are in the table that Marcos presented (see the slides) are useful in the context of geo-replication. Marcos gave an overview of some techniques and said that his advice is to stay in the upper left corner (synch read-write) if possible, and add complexity if performance or semantics require it.

Petr Kuznetsov asked whether new consistency levels are required to meet the needs of geo-replicated applications. Marcos pointed out the need to have a negotiation between performance and invariant preservation. A strategy would be to start from what the application expects in terms of invariants and then select the techniques adequate for those.

3.2 Cloud Storage Consistency Explained Through Baseball: Tutorial by Doug Terry

Kapil Vaswani and Alan Fekete

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.TerryDouglas1.Slides.pptx

Doug Terry spoke about the different consistency choices that a cloud system could offer to applications. He mentioned several systems. *(i)* AWS S3 has eventual consistency. *(ii)* AWS SimpleDB gives a choice between eventual or strong (it started with eventual, and later added the choice by a flag in the API for a given read to be done with strong consistency). *(iii)* Google AppEngine offers the same choice, but it started from a strong read and then added a choice for eventual consistent read. *(iv)* Yahoo! PNUTS also offers a choice of eventual or strong consistency on each read. *(v)* Cassandra has eventual or strong, but the choice must be made system wide by setting the size of R and W parameters. *(vi)* Windows Azure is 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 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. (We refer the interested reader to his slides.) 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).

Peter van Roy asked whether the system could figure things out, and determine which level is needed? Doug's answer was probably not, but annotations or other mechanisms might help.

Jennifer Welch asked is there a performance advantage in the combined guarantees, compared to just asking for strong consistency. Doug said Yes.

Allen Clement asked why not talk about the state of the system (rather than what reads return)? Doug said you can only observe state through reads, so properties talk only about reads.

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

Doug was asked what is the write consistency model? His answer was that only reads matter.

3.3 Database consistency tutorial: Alan Fekete

Carlos Baquero and Nicholas Rutherford

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.FeketeAlan1.Slides.pdf

Alan Fekete 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. The comment was made that many databases in the wild run with read-commit isolation, rather than serialisability, and may not be aware of the consequences.

Marc Shapiro asked about the registering of real world events on the database. Alan replied that the database is not limited to the registering of real world events without influencing it, and complemented with an example: When processing a transaction that

registers the paying for a real world exchange, if the transaction aborts due to lack of credit this is reported back to the real world and cancels the exchange that was taking place.

Rodrigo Rodrigues, Marc Shapiro and David Lomet commented on the issue that often there are consistency constraints that are not made explicit in the database. Some constraints are enforced in the program logic that interacts with the database.

Marcos Aguilera asked if there was a reason why the control flow is excluded from the database management system. Alan commented that to some extent this can be declared in stored procedures.

In response to a question by Bettina Kemme, Alan commented that serializability can be extended to include external consistency, leading to linearizability and encompassing session guaranties.

Marc Shapiro questioned whether considering that all operations are effective is equivalent to only considering transactions that have committed, ignoring those that have aborted. Alan replied that one can project the execution to include only the committed transactions.

4 Technical presentations

4.1 Presentation of the workshop: Marc Shapiro

Carla Ferreira and Rodrigo Rodrigues

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.SWM.Slides.pdf

Marc Shapiro presented the workshop and gave some context regarding Schloss Dagstuhl. He proposed the main topic of the workshop could be the tension between strong versus weak consistency. He also called the attention to the fact that different communities use different terminology, focus on different angles of this problem, and sometimes use the same terminology to mean different things. He concluded with some open questions.

Liuba Shrira commented that there is an interesting question that was left out was the evolution of technology, and how it influences the work of these communities. An example was the availability of the notion of global time.

Alan Fekete commented that in cross-community gatherings it is important to pay attention that different communities use different languages, and to make few assumptions about what the audience knows about terminology or background knowledge.

4.2 Making geo-replication fast as possible, consistent when necessary: Rodrigo Rodrigues

Kapil Vaswani and Alan Fekete

Rodrigo Rodrigues spoke next on “Making geo-replication fast as possible, consistent when needed.”

He pointed that delays make users unhappy, and the system achieves less revenue. So one aims to place replicas around world, thus 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 1st replication level is a central server or master/primary. The 2nd level has remote georeplicated replicas. At level 3: replicas are in CDN infrastructure. The 4th level of replication is P2P or hybrid CDNS (e.g., Akamai NetSession). A 5th 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 [12, 15]. His systems goal is to balance strong consistency (coming 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 proposes 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.

Rodrigo worked through a bank application.

A question asked why not synchronise red with all operations (as done in previous systems)? Answer is 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.

So Rodrigo suggests 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 don't 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.

In question time, Rodrigo made the following points: causal consistency is done with version vectors having one entry per datacentre; the 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).

4.3 Cloud Types and Revision Consistency: Sebastian Burckhardt

Carlos Baquero and Nicholas Rutherford

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.BurckhardtSebastian.Slides.pptx

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.

Maurice Herlihy made a comment on possible relations with persistent data structures [14].

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.

Liuba Shrira asked if they considered direct communication between devices. In reply, Sebastian added that communication with the servers is important but that extra connections, between devices, can be also useful as a complement.

4.4 Semantics of Eventually Consistent Systems: Alexey Gotsman

Petr Kustnetzov and Kaushik Rajan

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.GotsmanAlexey.Slides.pdf
Paper http://www.dagstuhl.de/mat/Files/13/13081/13081.GotsmanAlexey.Paper.pdf

The speaker is primarily interested in 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 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, restrict 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, such as:

- Timestamp last writer,
- High level commutativity,
- Return all conflicting values to users,
- Application-dependent resolution,

as well as invariants that should be observed in the presence of ongoing updates, such as:

- Read your own writes
- Causal or FIFO Ordering of operations

The framework addresses the conflict-resolution techniques via *data type specification* and invariants via *consistency axioms*.

Users interact with the system via requests and response events. An execution is modeled as a tuple which capture relations between events (operations, parameters, and returned values, etc.), session order (similar to program order), visibility relation (delivery of updates), and arbitration relation (for conflicting updates).

Now the data type is specified as a function $F : CONTEXT \mapsto OPERATION\ OUTPUT$, where context of an operation a is a projection of events that have been delivered to the replica performing a (actions visible to a).

Now 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 (available on the seminars' web page) validates the specifications via example abstract implementations.

Marcos Aguilera and Marc Shapiro questioned if the proposed definition of eventual consistency is not too weak actually to be useful.

4.5 Quantifying inconsistency: the transactional way: Bettina Kemme

Petr Kustnetzov and Kaushik Rajan

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.KemmeBettina.Slides.pdf

How likely is it that a real execution gives strong consistency even though the system only guarantees weaker consistency? The talk advocates the idea of *quantifying* inconsistency with respect 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 inconsistency actually occur. This may help to answer questions like : when we move to cloud storage with a different consistency guarantee, how consistent my existing application would 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. The technique here was based on building a dependency graph with RW, WR, WW conflict edges. To make the search for cycles efficient, we can use the properties of isolation levels, which allows us to exclude some paths. Cycles are visualized, and patterns are identified that might help in rewriting the system.

The second application of the method was cloud storage where transactions are not necessarily supported. The notion of a *work unit* is introduced instead.

It is assumed that local read-write ordering is preserved and that write order is consistent with commit order (primary master with FIFO messaging). However, in this case, some edges (WW, RW) are undetectable. The solution would be to build an approximate graph which says things like either one of two ordering holds. The resulting anomaly classification would thus include notions like *maybe cycles*.

Future work will include object-based anomalies, stale reads, violations of monotonicity.

Questions:

- Anomalies are often caused by some system failures: can the framework be used to detect and identify failures – yes it can.
- What is the slowdown it causes: not much, finding cycles in application engine incurs slowdown of only 2-3%
- How the does the effect depend on workloads: need testing framework around.

4.6 The Emperor's new consistency – the case against weak consistency in data centres: Marcos Aguilera

Alexey Gotsman and Jennifer Welch

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.AguileraMarcos3.Slides.pdf

Several data center storage systems adopt weak consistency models, in order to improve performance and availability; plus they are easier to develop. 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.

Someone asked whether stale data is inherent in all weak consistency systems. Marcos answered that staleness is the most common form of lack of strong consistency, but there could be worse things like garbage. Someone pointed out that stale data may still be consistent, e.g., you get multiple data items corresponding to a consistent snapshot. For instance, serialiability can return stale data in reads; the response was to combine serializability with external consistency. Another comment was that PSI [15] returns data that is stale but still pretty good. One can also have eventual consistency + causality + session guarantees, which is pretty good. One needs to qualify the statement that weak consistency returns bad data. A response was that there are degrees of badness.

Marcos argues first that *the drawbacks of weak consistency are expanding* due to technical, legal, and sociological reasons.

Someone raised the point that weak consistency is not well-defined (it is commonly construed as simply meaning not strong consistency), and thus who is to say that there isn't a better definition of weak consistency that avoids the drawbacks?

Users typically tolerate strange behavior in free apps. However, increasingly users pay for their apps, and paying users plus strange behavior equals angry users. This is one example

of increasing drawback of weak consistency.

The question was asked why equate staleness with strange behavior? The answer gave an example that violates causality. Then it was pointed out that causality can be provided cheaply. Someone pointed out that serializability also gives stale data, but the response was that usually systems try to avoid it.

Another factor is the increasing reliance on data center apps (e.g., banking and medical records). As the application provider is subject to liability, anomalous behavior can lead to monetary loss.

Someone pointed out that the pressure of liability does not lead to systems being built better, it just leads to checklists. The response was that then the burden is on the government to set up the checkboxes.

A third factor is that there are increasing number of layers and more integration among apps. Humans can identify and tolerate inconsistency but programs have a harder time.

Marcos' second argument is 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.

Someone commented that strong consistency is not that expensive to solve, say with Paxos in the data center. Someone expressed disagreement with the claim that partitions will go away and latencies will reduce in the relatively near future. Marc showed a graph of throughput vs. latency within a data center for different protocols, showing that Parallel Snapshot Isolation is much better than serializability. However, some people said that the implementation of serializability was not optimized.

4.7 HAT, not CAP: Highly available transactions: Alan Fekete

Alexey Gotsman and Jennifer Welch

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.FeketeAlan.Slides.pdf

Alan advocates 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 is on ways to group operations on multiple items for internet sale 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 Adya [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 an implementation, in which the client buffers its updates. On commit, it propagates them asynchronously as a group, and tracks causal precedence. The point is mainly to show the possibility of an implementation; lots of engineering will be necessary to get decent performance.

— Q: is this is a model where you ship data to the client, run everything at the client, and the storage system is just storage? — A: yes, but this is not to suggest this is how you *should* implement it. It is just to show the possibility that it can be done even in the presence of partitions.

— Q: Aren't you just reinventing group communication in transactions? There the whole problem was merging; is it the same problem here? — A: Alan doesn't know that they will have that problem. Transaction operations are reads and obliterating writes. It may be that lots of applications may also want to provide user-defined merging; that's a separate issue. Certainly, it is very similar to what happened in partition-tolerant group communication systems, but with the addition of transactions.

— Q: Are you delaying commit until the partition is reconciled? — A: No: the transaction commits, but the other side doesn't learn about it until later. So the transaction on the other side will not see these changes.

— Q: Can a client observe an older value after getting a newer value? — A: read-committed implementations also have a monotonicity property. Definitions typically don't have that property. But they do have per-item monotonicity, because they add causality.

— Q: Since the client must maintain causality meta-data, how to avoid an enormous space overhead? — A: It's an issue we need to study.

4.8 Scalable transactional consistency for your cloud data: David Lomet

Alexey Gotsman and Jennifer Welch

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.LometDavid.Slides.pptx

The dream is to have transactions anywhere with data in the cloud, mobile device, 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. The cloud backbone is the data center. For transactions and inter-operation, start with two-phase commit (2PC), but 2PC almost never crosses administrative domains, but is only in cluster-based systems.

Vendors provide limited offerings. 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 2PC participant.

Think of the data center (DC) s 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, yet transaction support is very limited.

The intuition of the new idea is *we do not do 2PC with disk, which is an atomic page store.* Put transactions on top of atomic record stores. We need to worry about performance. Details of Deuteronomy were given, which separates transactions from data.

The question was asked whether the contract between the TC and the DC should include locking, but the answer is no, the locking is all confined to the TC.

A prototype has been built, tested, and experimented on. Performance graph shows that latency has big impact on performance. They can get millions of operations per second!

— Q: Is the system designed for a specific workload, or is it general? — A: It is OLAP-focused, but nothing precludes using it in a general purpose way; however, one would need to think harder about performance.

— Q: You have to propagate LSN to the data store; is it transparent to the data store? — A: You have to enforce idempotence. Also use it to manage the log locally.

— Q: What kind of queries do you send down? Predicates? — A: We only send singleton record operations.

— Q: What is an operation? — A: Insert, delete or update.

— Q: Do you use locking? — A: Yes.

— Q: How do you deal with TC failure? — A: Paxos-style replication. — Q: Isn't that putting the burden of 2PC on Paxos? — A: It does not involve DCs.

— Q: Isn't the TC a bottleneck? — A: Write bandwidth is the main problem for TC, as updates must go through TC. In contrast, a large read do not have to go through TC If you "run out of gas", instantiate another TC.

4.9 Principles for Strong Eventual Consistency: Marc Shapiro

Vivien Quéma and Amr el Abbadi

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.ShapiroMarc.Slides.pdf

Marc Shapiro defined strong eventual consistency as "Eventually, correct replicas eventually reach equivalent states (potentially going through different histories)".

Doug Terry mentioned that this is the case in Bayou provided that replicas re-order updates themselves. Amr El Abbadi noted that Dynamo allows replicas to diverge, but lets users solve conflicts. He asked whether this is a problem or not.

Marc then introduces 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), etc. Several questions were asked during the talk:

- Doug Terry asked whether operations are necessarily idempotent. Marc replied no.
- André Schiper asked why causal delivery is needed. Marc replied that this is to be able to apply a sequential specification. Causal delivery is ensured using vector clocks.
- Marcos Aguilera 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.

4.10 Composing Lattices and CRDTs: Carlos Baquero

Vivien Quéma and Amr el Abbadi

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.BaqueroCarlos.Slides.pdf

Carlos Baquero gave a talk titled "Composing Lattices and CRDTs."

This talk was a continuation of the previous talk. Carlos emphasized the language aspects of CRDTs. The work presented by Carlos relies on the state-based approach and is based on order theory and the notion of lattices. Carlos presented a type hierarchy comprising the following types: set, poset, lattice, lattice with bottom. Carlos developed various examples, including lexographic ordering and the antichain.

Carlos then introduced a notion of "inflation" (ensures monotonically-advancing updates) with examples. He also explained how inflations can be sequentially composed. Several questions were asked during the talk:

- Someone asked whether lattice compositions are universal. Carlos replied that this is an open question.
- Peter van Roy asked whether it is possible to track connections between CRDTs using the theory presented by Carlos. Carlos does not know because he did not try. He said that using this theory, it is possible to know what is updated and that, consequently, it is possible to ship adequate parts of the state. This is a first step towards tracking connections between CRDTs.

- Carla Ferreira asked what exactly they can prove on CRDTs. Carlos replied that it is only possible to know that CRDTs are well defined. It is not possible to prove that CRDTs do something useful. But this is an open question: they did not try to do it.

4.11 Swiftcloud: geo-replication right to the edge: Nuno Preguiça

Doug Terry and Sebastian Burckhardt

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.PreguicaNuno1.Slides.pdf

The context of this talk is the latency experienced by the client when accessing the closest datacenter. 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.

The diagram on the slides shows the transaction execution on the client. The key design features include: CRDTs (conflict-free replicated data types), Asynchronous Replication, Multi-Version Server, Version-Vectors, and Client-Assisted Failover.

The audience raised several questions:

- *Are the transactions replayed at the data center?* Yes, for the effects.
- *Must we keep the updates at the client until confirmed by the data center?* Yes.
- *How do you implement the isolation levels?* Using the dependency vector.
- *How could you commit in disconnected mode?* Only in asynchronous mode.
- *What is the size of the vector clock?* The number of data centers.
- *How is causal consistency achieved?* By executing at data center only if dependencies are satisfied.

4.12 Applications for geo-replicated systems — Where are the limits?: Annette Bieniusa

Doug Terry and Sebastian Burckhardt

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.BieniusaAnnette.Slides.pdf

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.

Example 1

The first example is a social network supporting the operations log in/out, post, send, view wall, manage friends, poll, and like.

There was a question from the audience: How do you choose the granularity of the CRDTs (object vs. field)? The answer was: at the object level; this is mostly a performance question, not a semantic one.

Example 2

The second example is 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.

The slide on execution time shows 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.

The audience raised several questions at this point:

- *How do you handle "double creation" of a file?* We assume the user is trying to create the same file (other options are possible as well).
- *What's the overhead of FUSE in your implementation?* This is shown in the performance results (about 33%).
- *Did you actually implement this?* Yes, everything.
- *How do you implement the FS?* Each subdirectory is its own map.
- *Do you ever renew the cache if the client only reads?* Yes, we have a notification system to inform clients of new updates.

Example 3

The third example is web e-commerce, such as a bookstore. This example contained a number of challenges:

- Limited resource (we should not order books that are not available)
- Top-N (we use approximation)
- We need an index of products (we provide this as service)
- The code mixes small and big transactions (we use SQL to execute big ones on server)
- There needs to be 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.

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

Doug Terry and Sebastian Burckhardt

Sharing memory in networked systems is nicer than message passing; otherwise we wouldn't 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.

4.14 Snapshot Isolation with Eventual Consistency: Douglas B. Terry

Ioannis Nikolakopoulos and Ricardo Jiménez-Peris

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.TerryDouglas2.Slides.pptx

Doug Terry presented a cloud stsorage system, the main goal of which is to provide: multiple consistency levels; read-write transactions on replicated and partitioned data with snapshot isolation; 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, such as strong, Read-Modify-Write, . . . , Prefix.

At this point Alan Fekete asked if prefix consistency is the lowest sensible level. Doug answered that this happens only due to the specific implementation. Alan followed up asking if serializability is an option. The speaker answered that it is and explained the details in the last part of the talk.

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).

At this point there was a question if there is any need for transactions to be tagged as read-only. The answer was negative.

After that Doug got to the description of the key issue which was how to get the Read timestamp. He explained that this depends on the consistency chosen, giving 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.

4.15 ChainReaction: a Causal+ Consistent Datastore based on Chain Replication: Luís Rodrigues

Ioannis Nikolakopoulos and Ricardo Jiménez-Peris

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.RodriguesLuis.Slides.pdf

ChainReaction is a system for data store that provides Causal+ consistency using a variant chain replication. The servers are organized in a One-Hop DHT with different objects spanning on different parts of the chain. The changes are requested to a proxy and propagate from the head of the respective chain to the tail. The reads are distributed along the chain for reducing the tail bottleneck, weakening though the consistency. The last node that the client read from is kept in metadata.

At this point there was a question about the size of the metadata. Luís answered that the worst case scenario is one entry per object. However, they 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 less the writes are. The next question was what can be done for the write operations to be optimized. Luís 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.

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

Ioannis Nikolakopoulos and Ricardo Jiménez-Peris

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.ElAbbadiAmr.Slides.pdf

Amr gave an overview of the evolution and the recent situation of data management in the cloud, starting by 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 achieved in having high level abstractions like "joins" and transactions and that is what we can expect as the new wave of solutions.

Continuing, the speaker presented different approaches on splitting RDBMS systems to multiple nodes, both static (ElasTras, SQL Azure, Megastore) and dynamic. Then he presented solutions for elasticity and fault tolerance via replication and how this solutions can be classified according to their consistency guarantees.

4.17 Consistency without consensus and linearizable resilient data types: Kaushik Rajan

Maurice Herlihy and Allen Clement

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.RajanKaushik1.Slides.pptx
Paper http://www.dagstuhl.de/mat/Files/13/13081/13081.RajanKaushik1.Paper.pdf

Kaushik Rajan gave a talk titled "Consistency without consensus: linearizable resilient data types (LRDT)." He observed straight away that the title itself had evolved over the course of the workshop; many of the previous talks had introduced ideas and themes that were relevant and dramatically simplified the background and motivation for his talk.

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

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.

4.18 ACID and modularity in the cloud: Liuba Shrira

Maurice Herlihy and Allen Clement

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.ShriraLiuba.Slides.ppt

Liuba Shrira discussed ACID and modularity in the cloud. The 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 observd 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 data base 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 proposes 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 echo 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.

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

Alex Shvartsman and Luís Rodrigues

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.MichaelMaged.Slides.pdf

The talk 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). On this topic, Peter van Roy questioned the relevance of the idempotent work stealing for long running tasks (where multiple executions could be detrimental for performance). 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. In this context, David Lomet and Marcos Aguilera made a number of interesting remarks and questions reagrding the different ways the designer could use to convert a SNC-API to an non-SNC API.

4.20 Commutativity, Inversion, and other Stories of Consistency and Betrayal: Maurice Herlihy

Alex Shvartsman and Luís Rodrigues

The talk started 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 the transaction level concurrency control mechanisms. Under this model, transaction recovery needs to be based on an operation log, and ibe 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").

In this context, David Lomet asked about the differences between the proposed approach and multi-level concurrency control schemes that have been designed for databases. Maurice agreed that there are similarities between the approaches.

Alan Fekete made a note about the difficulties in deriving the appropriate inverse operations to support recovery. Maurice clarified that in Scala, closures were used for this purpose.

Marcos Aguilera commented that 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.

Alexey Gotsman asked if some of the code used to implement concurrent objects was not 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.

4.21 Concurrent Data Representation Synthesis: Mooly Sagiv

Mike Dodds and Marcos Aguilera

Mooly Sagiv presented a technique for synthesising fine-grained concurrent data-structures from high-level relational specifications.

Sagiv 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.

Alexey Gotsman asked why the DAG needed to encode multiple paths to a given node. Sagiv responded that 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.

Marcos K. Aguilera and Bettina Kemme both asked why not simply use a database system. Sagiv responded that his approach would be faster by tuning for particular workloads. David Lomet commented that general databases can't control the workload, so they can't specialise.

Sagiv commented early in the talk that he also hoped to help concurrent database implementors, and Bettina Kemme asked how. Sagiv argued that low-level concurrent datastructures from Linux are highly optimised for particular applications. If one could compile an in-memory database into such data-structures, it might be possible to achieve better performance.

Alexey Gotsman asked whether the two-phase locking approach would be sufficient for an application such as Linux. Sagiv said that it isn't clear, but Linux does lots of things that his system couldn't currently handle.

4.22 Distributed unification an a basis for transparently managing consistency and replication in distributed systems: Peter van Roy

Mike Dodds and Marcos Aguilera

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.VanRoyPeter.Slides.pdf

Peter van Roy 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 synchronisation 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. Shapiro said that this property does not hold for CRDTs, and wondered what the connection might be. Peter responded that his aim was to remove sychronisation.

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.

4.23 Time bounds for shared objects in partially synchronous systems: Jennifer Welch

Pierre Sutra and Achour Mostefaoui

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.WelchJennifer1.Slides.pptx

Jennifer Welch 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 points out that those results refine the CAP impossibility result.

4.24 Reduction theorems for proving serialisability with application to RCU-based synchronisation: Hagit Attiya

Pierre Sutra and Achour Mostefaoui

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.AttiyaHagit.Slides.pptx

Hagit Attiya presented a reduction theorem for proving serializability, with application to RCU-based synchronization.

The talk starts 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 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].

4.25 Abstractions for Transactional Memory: Noam Rinetzky

Annette Bieniusa and Peter van Roy

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.RinetzkyNoam.Slides.pdf

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.

Annette Bieniusa wanted to know what proof obligations remain for obtaining a correctness proof for a concrete TM implementation. Noam explained that using their results it suffices to show that the language implements opacity.

Doug Terry asked how aborted transactions can be observable, even though they do not have any effect. Noam clarified that the language definition allows aborted transactions to issue side-effects by modifying local state, similar to "nested top actions" in database systems.

Hagit Attiya then underlined the relevance of this model as upcoming Hardware Transactional Memory systems are implementing these semantics.

4.26 Idempotent transactional workflow: Ganesan Ramalingam

Annette Bieniusa and Peter van Roy

Slides http://www.dagstuhl.de/mat/Files/13/13081/13081.RamalingamGanesan.Slides.pptx

This 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.

Many questions were asked during the talk:

Q: What happens if the first transaction commits and the second doesn't, you need to roll back the first one? **A:** Yes, to handle that case the programmer must provide compensations.

Q: You could use transaction chopping, only the first can abort (all checking is put there), if it commits the rest will too. **A:** Our approach handles cases where this doesn't work.

Q: This corresponds to multilevel transaction model of the 1990s, which has compensations. **A:** Yes.

Q: Do you need a scheduler for guaranteeing the order of the flow? **A:** We only need a scheduler for performance, correctness is guaranteed by the model.

Q: How do you guarantee global uniqueness of ids? **A:** It's up to the programmer.

4.27 BubbleStorm: replication, updates and consistency in rendezvous information systems: Alejandro Buchmann and Robert Rehner

Annette Bieniusa and Peter van Roy

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 questions were asked about this system.

Q: What is the difference with quorum systems? **A:** Placement of replicas.
Q: How much of the system can be used as building blocks? **A:** All the lower layers, event scheduler, network. communication, have been used in many student projects.
Q: What about the visualization interface? **A:** Yes, and in addition we have a statistics interface.
Q: How would you measure inconsistencies in your system? Using logs? **A:** Yes, we could do this with postprocessing, with a testbed application that uses timestamps in a database. Note that a lot of information is gathered in various parts of the system, including the simulator, and we use it to gather different kinds of statistics.
Q: Could your analysis tool for communication patterns be used to analyze other systems? **A:** It has a clean interface, so this should be possible.
Q: What is G-Lab? **A:** A German system similar to PlanetLab. It is closer to (German) users and more stable than PlanetLab. It may be possible to use it outside of Germany, this is a legal issue.

5 Breakout sessions

5.1 Breakout sessions on distributed applications

Pawel Wojciechowski and Noam Rinetzky

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

Ganesan 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.

5.2 Breakout Groups and Discussion of Workshop Followup

Alex Shvartsman, Luís Rodrigues, Pierre Sutra and Achour Mostefaoui

Doug Terry announced the formation of four breakout groups for the after-lunch brainstorm sessions. The topics to be discussed include: *(i)* Consistency models, tools; *(ii)* Automatic analysis of consistency requirements; *(iii)* Performance impact of consistency models; *(iv)* Theory and potential help for developers.

The break-out groups are to answer the question: "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 Schiper 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.

6 Workshop Followup

6.1 Consensus paper

Alex Shvartsman and Luís Rodrigues

Marc Shapiro offered for discussion of what should be the written outcome of the workshop. The options on the table were: (a) do nothing; (b) write a common "consensus" paper that would summarize and integrate the different perspectives discussed during the workshop; (c) produce a collection of (individual, independent) papers; (d) to produce a coherent book with original material, along the lines of the book produced after the Monte Verità 2007 workshop [6]. Hagit Attyia proposed to start with the consensus paper, then aim to produce an original content book. After a lively discussion it was agreed to start on a common consensus paper, then consider producing the book based on the outcome of the first task.

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.* Springer-Verlag, 2010. 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.