Modeling, Analyzing, and Extending Megastore using Real-Time Maude

Jon Grov[1] and Peter Ölveczky[1,2]

[1]University of Oslo
[2]University of Illinois at Urbana-Champaign

Thanks to Indranil Gupta (UIUC) and UIUC's Center for Assured Cloud Computing

Goal

Cloud computing also when data consistency critical

- *e-commerce, banking, medical systems*

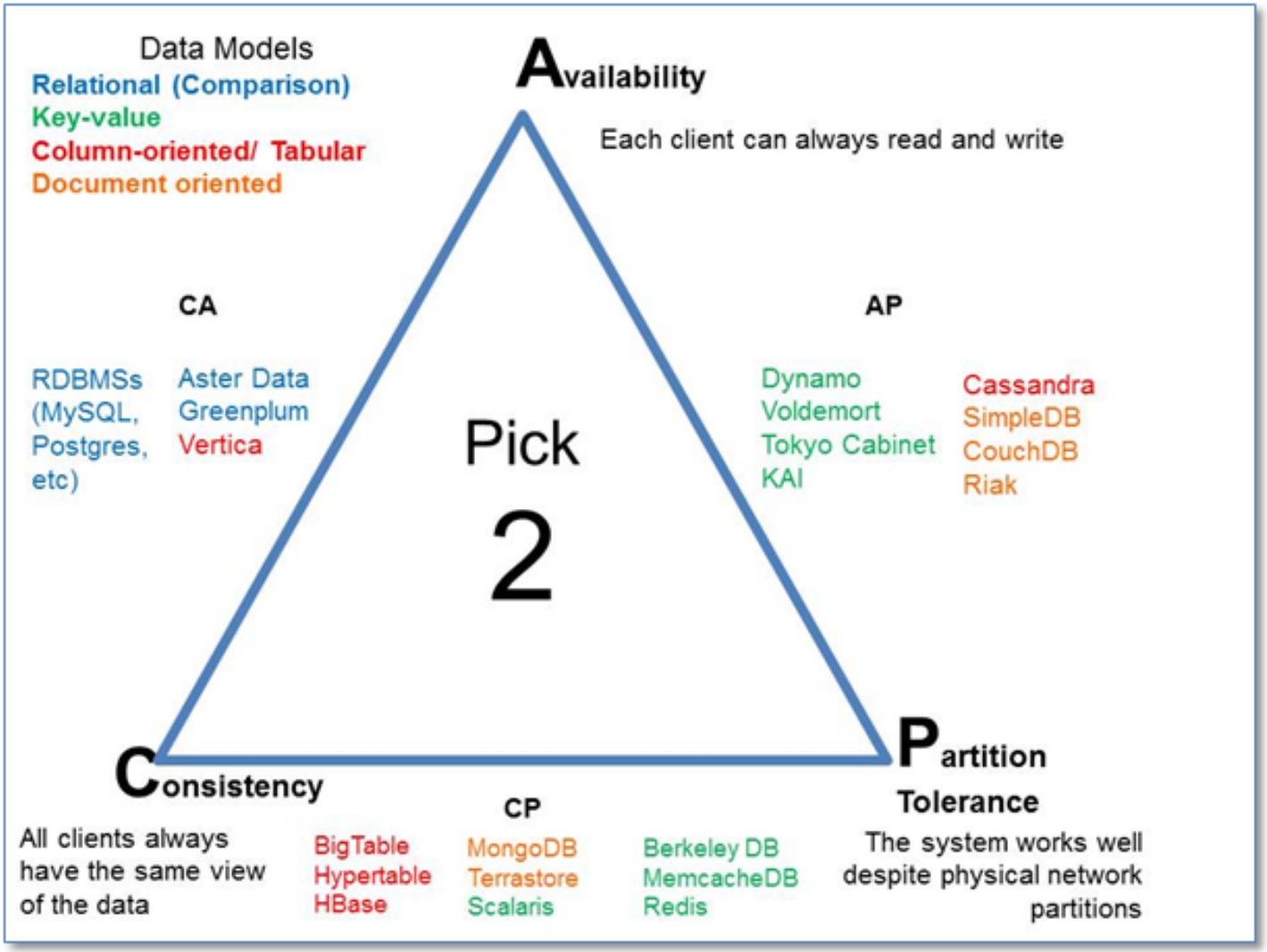
Part I: Formalizing Megastore

Data in the Cloud

- Availability and scalability: data must be replicated

Data in the Cloud

- Availability and scalability: data must be replicated



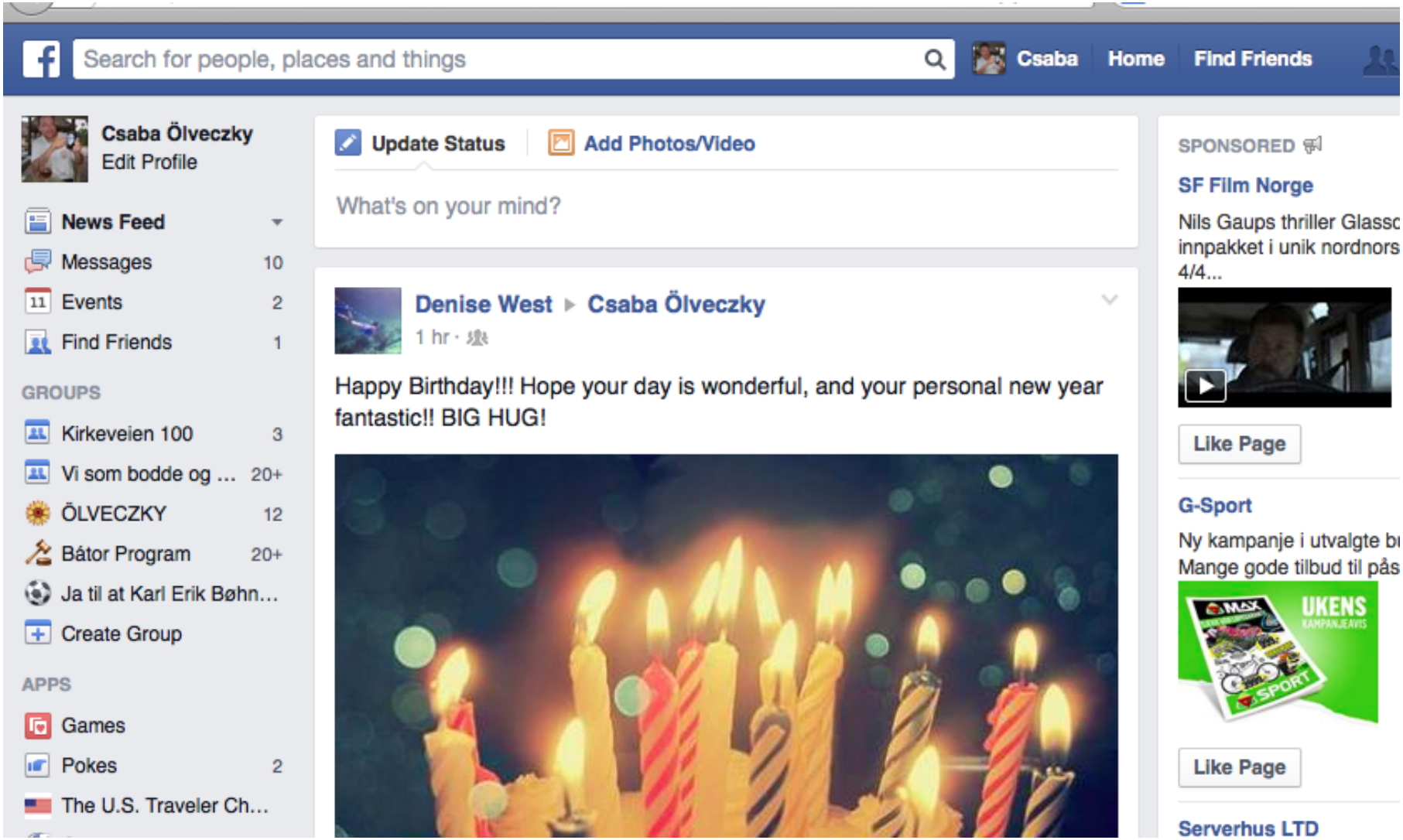
(Figure from http://flux7.com/blogs/nosql/cap-theorem-why-does-it-matter/)

Eventual Consistency

"Eventual consistency" OK for some applications

Eventual Consistency

"Eventual consistency" OK for some applications



Eventual Consistency

"Eventual consistency" OK for some applications

Eventual Consistency

"Eventual consistency" OK for some applications

... but not others:

- banking
- stock exchange
- electronic commerce
  - online auctions (eBay)
  - plane tickets ...
- medical (information) systems

Transactions

Databases traditionally provide ACID transactions

- atomicity
- consistency
- isolation ("serializability")
- durability

Transactions

Databases traditionally provide ACID transactions

- atomicity
- consistency
- isolation ("serializability")
- durability

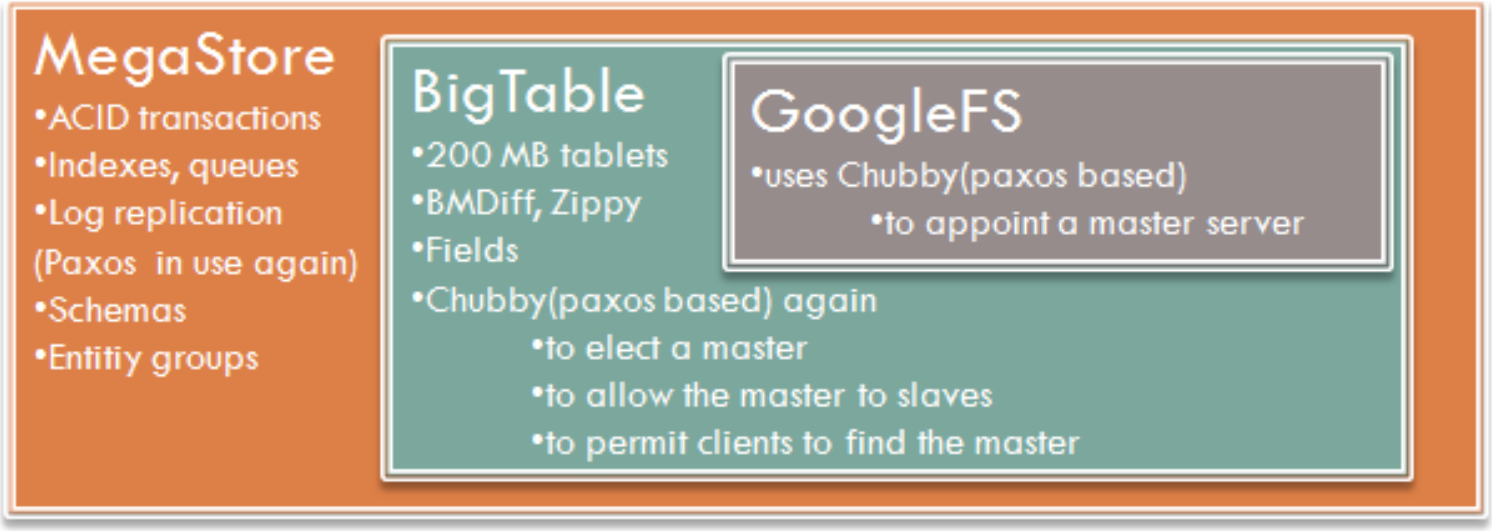
Little support for ACID transactions in cloud computing

Megastore

Megastore:

- Google's wide-area replicated data store
- Key part of Google's cloud infrastructure
- 3 billion write and 20 billion read transactions daily (2011)
- Adds (limited) transactions to wide-area replicated data stores

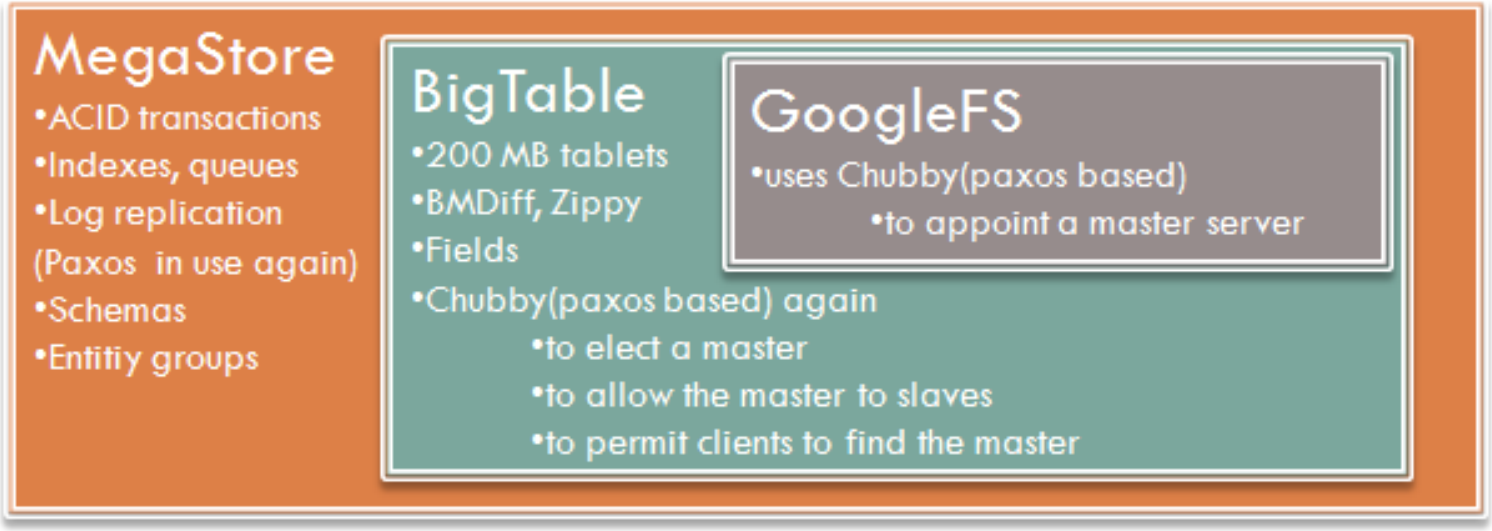
Megastore: Key Ideas (I)



(Figure from http://cse708.blogspot.jp/2011/03/megastore-providing-scalable-highly.html)

- Data divided into entity groups
  - Peter's email
  - Books on rewriting logic
  - Jon's documents

Megastore: Key Ideas (I)

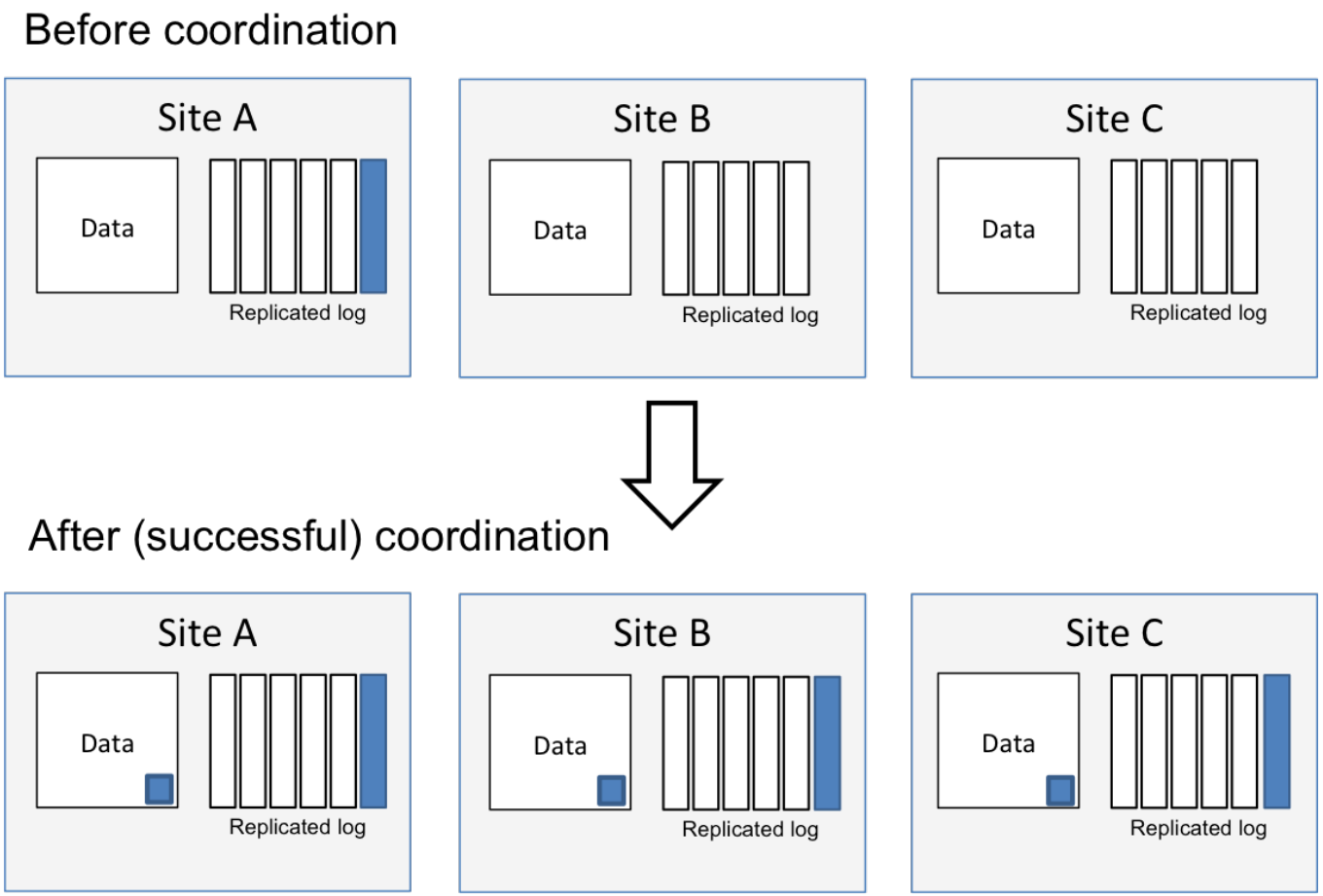


(Figure from http://cse708.blogspot.jp/2011/03/megastore-providing-scalable-highly.html)

- Data divided into entity groups
  - Peter's email
  - Books on rewriting logic
  - Jon's documents
- Replicated transaction log for each entity group

Megastore: Key Ideas (II)

- Node suggests next log entry for an entity group
- Paxos: agree on next log entry if concurrent updates



Megastore: Key Ideas (III)

- Consistency for transactions accessing a single entity group
  - no guarantee if transaction reads multiple entity groups

Megastore: Key Ideas (III)

- Consistency for transactions accessing a single entity group
  - no guarantee if transaction reads multiple entity groups
- ElasTraS, Spinnaker, Calvin, and Microsoft's Azure: consistency within each data partition

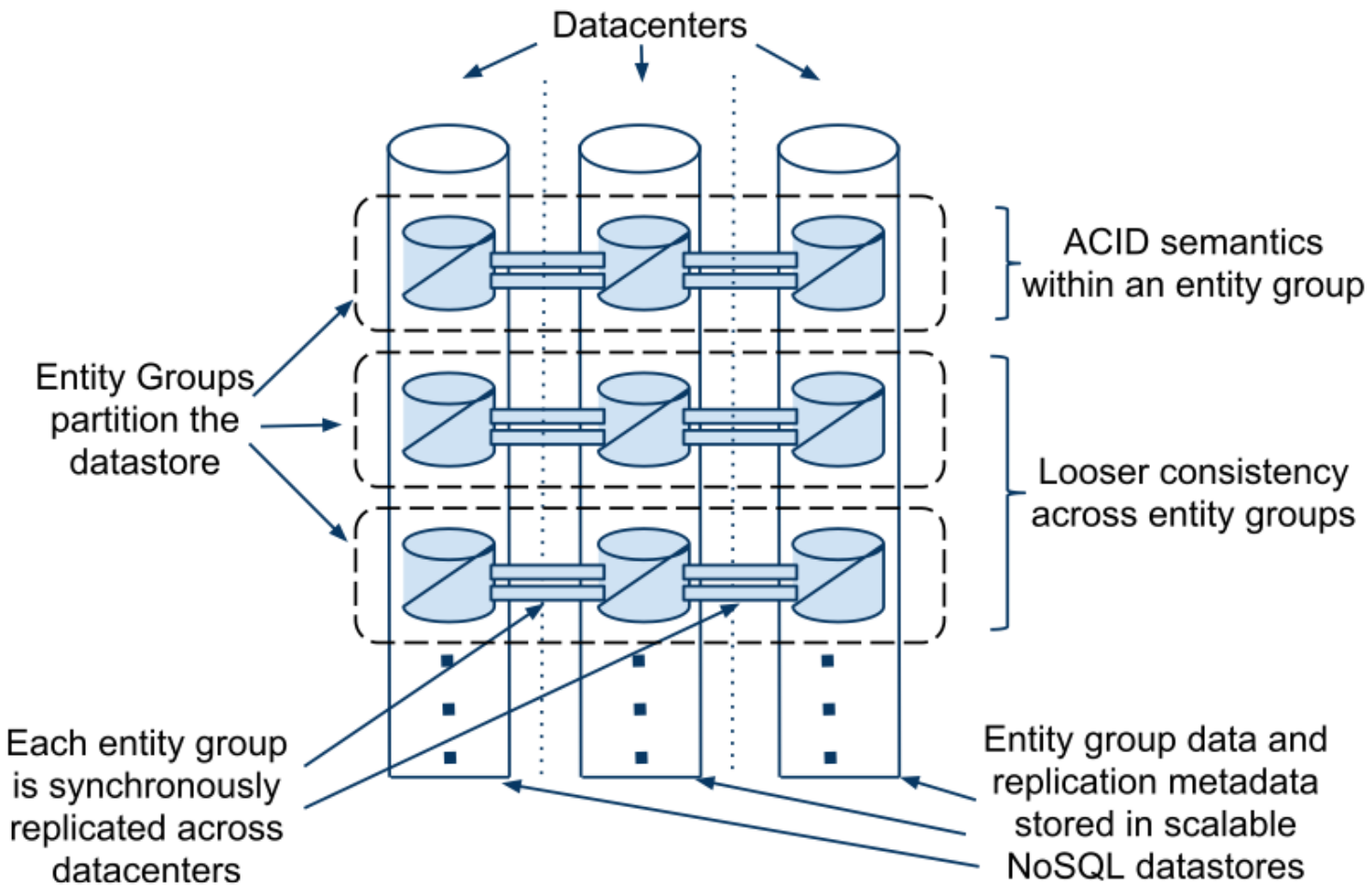


Figure 1: Scalable Replication

(figure from J. Baker et al., "Megastore: Providing Scalable, Highly Available Storage for Interactive Services")

Our Work: Motivation

- 12 page informal overview paper of really complex system
- Understand system and guarantees
- Basis for further research and extensions

Our Work

- [Developed and] formalized [our version of the] Megastore [approach] in Real-Time Maude

Our Work

- [Developed and] formalized [our version of the] Megastore [approach] in Real-Time Maude
  - first (public) formalization/detailed description of Megastore

Our Work

- [Developed and] formalized [our version of the] Megastore [approach] in Real-Time Maude
  - first (public) formalization/detailed description of Megastore
- 56 rewrite rules (37 for fault tolerance features)

Our Work

- [Developed and] formalized [our version of the] Megastore [approach] in Real-Time Maude
  - first (public) formalization/detailed description of Megastore
- 56 rewrite rules (37 for fault tolerance features)
- Real-Time Maude simulation and model checking used extensively throughout development of (our) Megastore

Background: Real-Time Maude

Real-Time Maude: rewriting-logic-based language and analysis tool

- Expressive formalism
  - algebraic specification specify data types
  - rewrite rules specify local transitions
- Object-oriented specification of distributed real-time systems

Background: Real-Time Maude

Real-Time Maude: rewriting-logic-based language and analysis tool

- Expressive formalism
  - algebraic specification specify data types
  - rewrite rules specify local transitions
- Object-oriented specification of distributed real-time systems
- Formal analysis:
  - simulation
  - reachability analysis
  - LTL model checking
  - Timed CTL model checking

A Rewrite Rule (in Failure-Free Setting)

When all operations in the `operations` list are completed (reads) or buffered (writes), the transaction is ready to commit. All buffered updates are merged into a candidate log entry. If the transaction updates entities from several entity groups, one log entry is created for each group. For each such entity group, the first step is to send the candidate log entry to the leader for the *next* log position, which was selected during the previous coordination round. The rule for initiating Paxos is modeled as follows:

```
crl [initiateCommit] :
    < SID' : Site |
         entityGroups EGROUPS,
         localTransactions : LOCALTRANS
            < TID : Transaction | operations : emptyOpList,
                                  writes : WRITEOPS, status : idle
                                  readState : RSTATE, paxosState : PSTATE > >
   =>
    < SID : Site |
        localTransactions : LOCALTRANS
           < TID : Transaction | paxosState : NEW-PAXOS-STATE,
                                 status : in-paxos > >
    ACC-LEADER-REQ-MSGS
   if EIDSET := getEntityGroupIds(WRITEOPS) /\
      NEW-PAXOS-STATE := initiatePaxosState(EIDSET, TID, WRITEOPS,
                                            SID, RSTATE, EGROUPS)
  /\ (createAcceptLeaderMessages(SID, NEW-PAXOS-STATE)) => ACC-LEADER-REQ-MSGS
```

A Rewrite Rule (cont.)

`getEntityGroupIds(WRITEOPS)` contains entity groups accessed by operations in `WRITEOPS`, and `NEW-PAXOS-STATE` contains one record for each entity group. These records contain the log position that `TID` requests to update and the candidate log entry *le*. The operator `createAcceptLeaderMessages` generates an `acceptLeaderReq` message to the leader of each entity group containing the transaction id `TID` and candidate log entry.

Another Rewrite Rule (no Failures)

The following rule [...] where a replicating site receives an `acceptAllReq` message. The site verifies that it has not already granted an accept for this log position (since messages could be delayed for a long time, it checks both the transaction log and received proposals). If there are no such conflicts, the site responds with an accept message, and stores its accept in `proposals` for this entity group. The record `(TID' LP SID OL)` represents the candidate log entry, containing the transaction identifier `TID'`, the log position `LP`, the proposed leader site `SID`, and the list of update operations `OL`.

```
crl [rcvAcceptAllReq] :
    (msg acceptAllReq(TID, EG, (TID' LP SID OL), PROPNUM) from SENDER to THIS)
    < THIS : Site |
      entityGroups : EGROUPS
        < EG : EntityGroup | proposals : PROPSET, transactionLog : LEL > >
   =>
    < THIS : Site |
      entityGroups :  EGROUPS
        < EG : EntityGroup |
          proposals : accepted(SENDER, (TID' LP SID OL), PROPNUM) ;
                      removeProposal(LP, PROPSET) > >
    dly(acceptAllRsp(TID, EG, LP, PROPNUM) from THIS to SENDER), T)
   if not (containsLPos(LP, LEL) or hasAcceptedForPosition(LP, PROPSET))
      /\ T ; TS := possibleMessageDelay(THIS, SENDER) .
```

Formal Analysis

- "Monte-Carlo" simulations for performance estimation
- LTL model checking
  - highly nondeterministic setting (many conflicts)
  - limited scenarios
  - super useful to discover many bugs not found during simulation

Performance Estimation

- Key performance measures:
  - average transaction latency
  - number of committed/aborted transactions
- 2 entity groups
- Randomly generated transactions (rate 2.5 TPS)
- Network delays:

	30%	30%	30%	10%
London $\leftrightarrow$ Paris	10	15	20	50
London $\leftrightarrow$ New York	30	35	40	100
Paris $\leftrightarrow$ New York	30	35	40	100

Performance Estimation (cont.)

- Simulating for 200 seconds (no failures):

	Avg. latency (ms)	Commits	Aborts
London	122	149	15
New York	155	132	33
Paris	119	148	18

Performance Estimation (cont.)

- Simulating for 200 seconds (no failures):

	Avg. latency (ms)	Commits	Aborts
London	122	149	15
New York	155	132	33
Paris	119	148	18

- Site failures:
  - mean-time-to-failure 10 seconds per site
  - mean-time-to repair 2 seconds

	Avg. latency (ms)	Commits	Aborts
London	218	109	38
New York	336	129	16
Paris	331	116	21

Model Checking (I)

- Desired properties (for finite number of transactions):
  - all transactions finish execution
  - all replicas of an entity must eventually have same value
  - all logs for entity group must eventually have same entries
  - execution was serializable

Model Checking (I)

- Desired properties (for finite number of transactions):
  - all transactions finish execution
  - all replicas of an entity must eventually have same value
  - all logs for entity group must eventually have same entries
  - execution was serializable
- Serializability tricky property
  - construct serialization graph during execution
  - check that graph has no cycles

Model Checking (II)

- All replicas same unless coordinator invalidated:

```
op entityGroupsEqualOrInvalid : -> Prop [ctor] .

ceq  {< S1 : Site | coordinator : eglp(EG1, LP) ; EGLP,
                    entityGroups :
                      < EG1 : EntityGroup | entitiesState : ES1 > EGS1 >
      < S2 : Site | coordinator : eglp(EG1, LP) ; EGLP,
                    entityGroups :
                      < EG1 : EntityGroup | entitiesState : ES2 > EGS2 >
      REST} |= entityGroupsEqual = false if ES1 =/= ES2 .

eq {SYSTEM} |= entityGroupsEqualOrInvalid = true [owise] .
```

Model Checking (II)

- All replicas same unless coordinator invalidated:

```
op entityGroupsEqualOrInvalid : -> Prop [ctor] .

ceq  {< S1 : Site | coordinator : eglp(EG1, LP) ; EGLP,
                    entityGroups :
                      < EG1 : EntityGroup | entitiesState : ES1 > EGS1 >
      < S2 : Site | coordinator : eglp(EG1, LP) ; EGLP,
                    entityGroups :
                      < EG1 : EntityGroup | entitiesState : ES2 > EGS2 >
      REST} |= entityGroupsEqual = false if ES1 =/= ES2 .

eq {SYSTEM} |= entityGroupsEqualOrInvalid = true [owise] .
```

- Execution serializable:

```
op isSerializable : -> Prop [ctor] .
eq {< th : TransactionHistory | graph : GRAPH >  REST}
          |= isSerializable = not hasCycle(GRAPH) .
```

Model Checking (II)

- All replicas same unless coordinator invalidated:

```
op entityGroupsEqualOrInvalid : -> Prop [ctor] .

ceq  {< S1 : Site | coordinator : eglp(EG1, LP) ; EGLP,
                    entityGroups :
                      < EG1 : EntityGroup | entitiesState : ES1 > EGS1 >
      < S2 : Site | coordinator : eglp(EG1, LP) ; EGLP,
                    entityGroups :
                      < EG1 : EntityGroup | entitiesState : ES2 > EGS2 >
      REST} |= entityGroupsEqual = false if ES1 =/= ES2 .

eq {SYSTEM} |= entityGroupsEqualOrInvalid = true [owise] .
```

- Execution serializable:

```
op isSerializable : -> Prop [ctor] .
eq {< th : TransactionHistory | graph : GRAPH >  REST}
          |= isSerializable = not hasCycle(GRAPH) .
```

- Desired property:

```
<> [] (allTransFinished /\ entityGroupsEqualOrInvalid
       /\ transLogsEqualOrInvalid /\ isSerializable)
```

Model Checking (III)

- Without fault tolerance:
  - model checked untimed Maude model
  - covers all possible message delays, transaction start times, etc.
  - 3 sites and 3 transactions

Model Checking (III)

- Without fault tolerance:
  - model checked untimed Maude model
  - covers all possible message delays, transaction start times, etc.
  - 3 sites and 3 transactions
- With fault tolerance: model checked timed model

Msg. delay	#Trans	Trans. start time	#Fail.	Fail. time	Run (s)
$\{20, 100\}$	4	$\{19, 80\}$ and $\{50, 200\}$	0	-	1367
$\{20, 100\}$	3	$\{10, 50, 200\}$	1	60	1164
$\{20, 40\}$	3	20, 30, and $\{10, 50\}$	2	$\{40, 80\}$	872
$\{20, 40\}$	4	20, 20, 60, and 110	2	70 and $\{10, 130\}$	241
$\{20, 40\}$	4	20, 20, 60, and 110	2	$\{30, 80\}$	DNF
$\{10, 30, 80\}$,and $\{30, 60, 120\}$	3	20, 30, 40	1	$\{30, 80\}$	DNF
$\{10, 30, 80\}$,and $\{30, 60, 120\}$	3	20, 30, 40	1	60	DNF

Part II: Megastore-CGC: extending Megastore

Our Work: Motivation

Goal

Cloud computing also when data consistency critical

- Sometimes transactions must access multiple entity groups

Our Work: Motivation

Goal

Cloud computing also when data consistency critical

- Sometimes transactions must access multiple entity groups
- Megastore: consistency only if transactions access single entity group

Our Work: Motivation

Goal

Cloud computing also when data consistency critical

- Sometimes transactions must access multiple entity groups
- Megastore: consistency only if transactions access single entity group
- Our approach: extend Megastore with consistency for transactions accessing multiple entity groups

Our Work: Motivation

Goal

Cloud computing also when data consistency critical

- Sometimes transactions must access multiple entity groups
- Megastore: consistency only if transactions access single entity group
- Our approach: extend Megastore with consistency for transactions accessing multiple entity groups
  - must maintain Megastore's
    - performance
    - strong fault tolerance

Our Idea

- Key observations:
  1. A Megastore site replicating different entity groups participates in all updates on those entity groups
  2. $\longrightarrow$ site has implicit local ordering on those updates

Our Idea

- Key observations:
  1. A Megastore site replicating different entity groups participates in all updates on those entity groups
  2. $\longrightarrow$ site has implicit local ordering on those updates

- Our idea:
  1. Select one such site and make its order explicit
  2. This site validates transactions on multiple entity groups before commit
     - ordering class = set of entity groups
     - ordering site for each ordering class

Megastore-CGC

Megastore-CGC piggybacks ordering and validation onto Megastore's coordination protocol

- no additional messages for validation/commit!
- maintains Megastore's performance and fault tolerance

Megastore-CGC

Megastore-CGC piggybacks ordering and validation onto Megastore's coordination protocol

- no additional messages for validation/commit!
- maintains Megastore's performance and fault tolerance
- failover protocol when ordering site fails

Developing Megastore-CGC

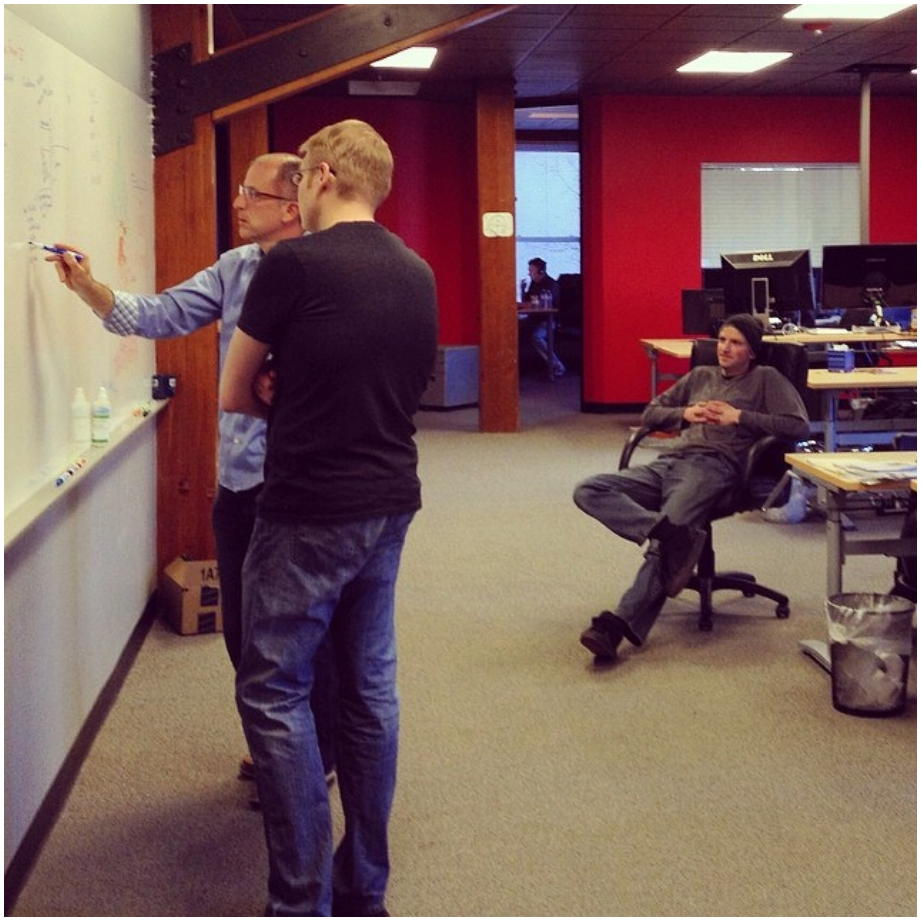
- Developed Megastore-CGC using Real-Time Maude
- 72 (fairly complex) rewrite rules
- Real-Time Maude simulation and model checking used extensively throughout development of Megastore-CGC

Software Engineering Perspective (I)

- Developing a fault-tolerant protocol is difficult
- Typically: pseudo-code description $\longrightarrow$ Java prototype

Software Engineering Perspective (I)

- Developing a fault-tolerant protocol is difficult
- Typically: pseudo-code description $\longrightarrow$ Java prototype
  - many hours of "whiteboard analysis"
  - still incorrect



Software Engineering Perspective (II)

- Our approach: Formal test-driven development:
  1. Express requirements as LTL formulas
  2. Develop Real-Time Maude model
  3. Test model using Real-Time Maude simulation and model checking
  4. For failing tests, analyse the problem and go to step (2)

Software Engineering Perspective (II)

- Our approach: Formal test-driven development:
  1. Express requirements as LTL formulas
  2. Develop Real-Time Maude model
  3. Test model using Real-Time Maude simulation and model checking
  4. For failing tests, analyse the problem and go to step (2)
- Allowed one person to develop and validate complex protocol in short time

Software Engineering Perspective (III)

- With much less effort, the fault-tolerance features of Megastore-CGC are significantly more mature than similar protocols the first author has been working on
- Model checking replaces most of the whiteboard analysis

Performance Comparison using Real-Time Maude

- Simulating for 1000 seconds (no failures)
- Megastore:

	Commits	Aborts	Avg. latency (ms)
Site 1	652	152	126
Site 2	704	100	118
RSite	640	172	151

- Megastore-CGC:

	Commits	Aborts	Val. aborts	Avg.latency (ms)
Site 1	660	144	0	123
Site 2	674	115	15	118
RSite	631	171	10	150

Model Checking (III)

Model checking scenarios

- 5 transactions (3 fixed, 2 with 2 start times), no failures, message delay 30 ms or 80 ms
  $\longrightarrow$ 108,279 reachable states, 124 seconds
- 3 transactions (all with 2 start times), one site failure and fixed message delay
  $\longrightarrow$ 1,874,946 reachable states, 6,311 seconds
- 3 transactions (all with 2 start times), fixed message delay and one message failure
  $\longrightarrow$ 265,410 reachable states, 858 seconds

Conclusion

- Developed model of Megastore from limited information
  - first formal analysis of transactional data stores

Conclusion

- Developed model of Megastore from limited information
  - first formal analysis of transactional data stores
- Extended the Megastore approach to Megastore-CGC
  - consistency for transactions accessing multiple entity groups
  - performance and fault tolerance on par with Megastore

Conclusion

- Developed model of Megastore from limited information
  - first formal analysis of transactional data stores
- Extended the Megastore approach to Megastore-CGC
  - consistency for transactions accessing multiple entity groups
  - performance and fault tolerance on par with Megastore
- Detailed formal specification in Real-Time Maude
  - shorter development time
  - increased quality of specification
  - analyze performance and correctness

Conclusion

- Developed model of Megastore from limited information
  - first formal analysis of transactional data stores
- Extended the Megastore approach to Megastore-CGC
  - consistency for transactions accessing multiple entity groups
  - performance and fault tolerance on par with Megastore
- Detailed formal specification in Real-Time Maude
  - shorter development time
  - increased quality of specification
  - analyze performance and correctness
- First generic (?) cloud data management protocol with consistency for transactions across partitions
  - Google's Spanner: similar features with advanced infrastructure

Conclusion

- Developed model of Megastore from limited information
  - first formal analysis of transactional data stores
- Extended the Megastore approach to Megastore-CGC
  - consistency for transactions accessing multiple entity groups
  - performance and fault tolerance on par with Megastore
- Detailed formal specification in Real-Time Maude
  - shorter development time
  - increased quality of specification
  - analyze performance and correctness
- First generic (?) cloud data management protocol with consistency for transactions across partitions
  - Google's Spanner: similar features with advanced infrastructure
- Real-Time Maude expressiveness key
  - model complex system
  - express complex requirements