Exploring Design Alternatives for the RAMP Transaction System through Statistical Model Checking

Si Liu, Jatin Ganhotra, Peter Csaba Ölveczky, Indranil Gupta, and José Meseguer

ACC – March 2017
Background

• Huge data in cloud systems
  - databases must be partitioned
  - Facebook: 25 terabytes of data per day
• Full consistency of distributed typically impossible
  - trade-offs: consistency level vs. efficiency
• Promising approach: Read Atomic Multi-Partition transactions (RAMP) by Bailis et al (SIGMOD 2014, TODS 2016)
  - atomic visibility: all or none of transactions updates are observable
  - no fractured reads: X friend of Y, but Y not friend of X
• Read Atomicity
  - baseline consistency model by Cerone et al (CONCUR 2015)
  - needed for most sensible consistency levels: Causality, Parallel Snapshot Isolation, Snapshot Isolation, Serialisability
Motivation

• How can the design space of a distributed transaction system such as RAMP be explored with modest effort, so that substantial knowledge about design alternatives can be gained before designs are implemented?

• How realistic and informative are the results of such design explorations?

• Ultimate long-term goal
  - library of formal executable building blocks
  - mix and match to build data stores with desired consistency and availability trade-offs
Our Approach

1. Formalized 8 RAMP-like designs in Maude as probabilistic rewrite theories
   - 5 by the RAMP developers and 3 of our own

2. Used statistical model checking of those models to analyze key performance metrics
   - throughput
   - average latency
   - second-round reads
   - strong consistency
   - read atomicity
How we go about it

We use:

1. Maude
   - Modeling framework for distributed systems
   - Supports rewriting logic specification and programming
   - Efficiently executable

2. PVeStA
   - Statistical model checking tool
   - Runs Monte-Carlo simulations of model
   - Verifies a property up to a user-specified level of confidence
Algorithm 1 RAMP-Fast

Server-side Data Structures
1: versions: set of versions \((\text{item}, \text{value}, \text{timestamp} t_{sv}, \text{metadata} md)\)
2: latestCommit\([i]\): last committed timestamp for item \(i\)

Server-side Methods
3: procedure PREPARE\((v: \text{version})\)
4: \hspace{1cm} versions.add\((v)\)
5: \hspace{1cm} return
6: procedure COMMIT\((ts_c: \text{timestamp})\)
7: \hspace{1cm} \(I_{ts_c} \leftarrow \{w.\text{item} \mid w \in \text{versions} \land w.t_{sv} = ts_c\}\)
8: \hspace{1cm} \forall i \in I_{ts_c}, \text{latestCommit}[i] \leftarrow \max(\text{latestCommit}[i], ts_c)\)
9: procedure GET\((i: \text{item}, ts_{req}: \text{timestamp})\)
10: \hspace{1cm} if \(ts_{req} = \emptyset\) then
11: \hspace{1cm} return \(v \in \text{versions}: v.\text{item} = i \land v.t_{sv} = \text{latestCommit}[i]\)
12: \hspace{1cm} else
13: \hspace{1cm} return \(v \in \text{versions}: v.\text{item} = i \land v.t_{sv} = ts_{req}\)

Client-side Methods
14: procedure PUT\_ALL\((W: \text{set of } (\text{item}, \text{value}))\)
15: \hspace{1cm} \(I_{ts_x} \leftarrow \text{generate new timestamp}\)
16: \hspace{1cm} \(I_{ts_x} \leftarrow \text{set of items in } W\)
17: \hspace{1cm} parallel-for \((i, v) \in W\)
18: \hspace{1cm} \(v \leftarrow (i.\text{item}, v.\text{value} = v.t_{sv} = ts_{ts_x}, md = (I_{ts_x} - \{i\})]\)
19: \hspace{1cm} invoke PREPARE\((v)\) on respective server (i.e., partition)
20: \hspace{1cm} parallel-for \(\text{server } s: s\text{ contains an item in } W\)
21: \hspace{1cm} invoke COMMIT\((ts_{ts_x})\) on \(s\)
22: procedure GET\_ALL\((I: \text{set of items})\)
23: \hspace{1cm} \(ret \leftarrow \{\}\)
24: \hspace{1cm} parallel-for \(i \in I\)
25: \hspace{1cm} \(ret[i] \leftarrow \text{GET}(i, \emptyset)\)
26: \hspace{1cm} \(v_{\text{latest}} \leftarrow \{\}\) (default value: \(-1\))
27: \hspace{1cm} for \(r \in ret\) do
28: \hspace{1cm} for \(i_{ts} \in r.md\) do
29: \hspace{1cm} \(v_{\text{latest}}[i_{ts}] \leftarrow \max(v_{\text{latest}}[i_{ts}], r.t_{sv})\)
30: \hspace{1cm} parallel-for \(\text{item } i \in I\)
31: \hspace{1cm} if \(v_{\text{latest}}[i] > r[i].t_{sv}\) then
32: \hspace{1cm} \(ret[i] \leftarrow \text{GET}(i, v_{\text{latest}}[i])\)
33: \hspace{1cm} return \(ret\)
The distributed state of RAMP model is a “multiset” of Partitions, Clients, Scheduler and Messages.

- `<c : Client | ... >`
- `[ D₁, p₁ ← Msg₁ ] [ D₂, p₂ ← Msg₂ ]` → `<s : Scheduler | GlobalTime: T, MsgQueue: enqueue(Msg₁, Msg₂) >`
- `{T+D₁, p₁ ← Msg₁ }` → `<s : Scheduler | GlobalTime: T+D₁, MsgQueue: dequeue(Msg₁) >`
- `<p₁ : Partition | ... >`
- `{T+D₂, p₂ ← Msg₂ }` → `<s : Scheduler | GlobalTime: T+D₂, MsgQueue: dequeue(Msg₂) >`
- `<p₂ : Partition | ... >`
Example (RAMP-Faster): on receiving PREPARE message

crl [receive-prepare-faster] :
{\text{T, O} \leftarrow \text{prepare}(\text{TID, ID, X, V, ts(O', SQN, MD, O'))}}
< O : \text{Partition} | \text{versions: VS, latestCommit: LC} >
=>
< O : \text{Client} | \text{versions: VS', latestCommit: cmt(LC, VS', ts(O', SQN))} >
[D, O' \leftarrow \text{committed}(\text{TID, ID, O})]$
if \text{VS'} := (v(X, V, ts(O', SQN), MD), VS) \text{ with probability } D := \text{distr}(...).
How realistic and informative are the results?

• Consistent with the experimental results obtained by the RAMP developers for their implemented designs
As client (or read) workload increases, RAMP-F's throughput increases...
How realistic and informative are the results?

• Consistent with the experimental results obtained by the RAMP developers for their implemented designs

• Confirm the conjectures made by the RAMP developers for their 3 unimplemented designs
Performance: 2nd-round reads

- RAMP-F+FC requires less second-round reads than RAMP-F
- ...
How realistic and informative are the results?

- Consistent with the experimental results obtained by the RAMP developers for their implemented designs

- Confirm the conjectures made by the RAMP developers for their 3 unimplemented designs

- Help uncover a new design that outperforms all other designs for latency, throughput and consistency, while providing read atomicity for 99% of the transactions
• RAMP-Faster incurs the lowest avg. latency among all RAMP versions
Performance: Read Atomicity

- RAMP-Faster slightly reduced read atomicity
Summary

• First formal executable probabilistic model of RAMP designs

• Statistical model checking of performance metrics
  • Consistent with the experimental results obtained by the RAMP developers
  • Confirm the conjectures made by the RAMP developers for their unimplemented designs
  • Help uncover new designs
Ongoing and Future Work

• Other consistency models
• Other performance metrics
• Model other distributed transaction systems
• Build a library of building blocks
  – Mix and match to generate any distributed transaction system with desired consistency and availability trade-offs
Read Atomic Multi-Partition Transactions

- Algorithms for ensuring atomic visibility in partitioned databases – either all of a transaction’s updates are observed, or none are.

- Useful for e.g., multi-key updates, maintaining foreign key constraints, secondary indexes and materialized views.