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

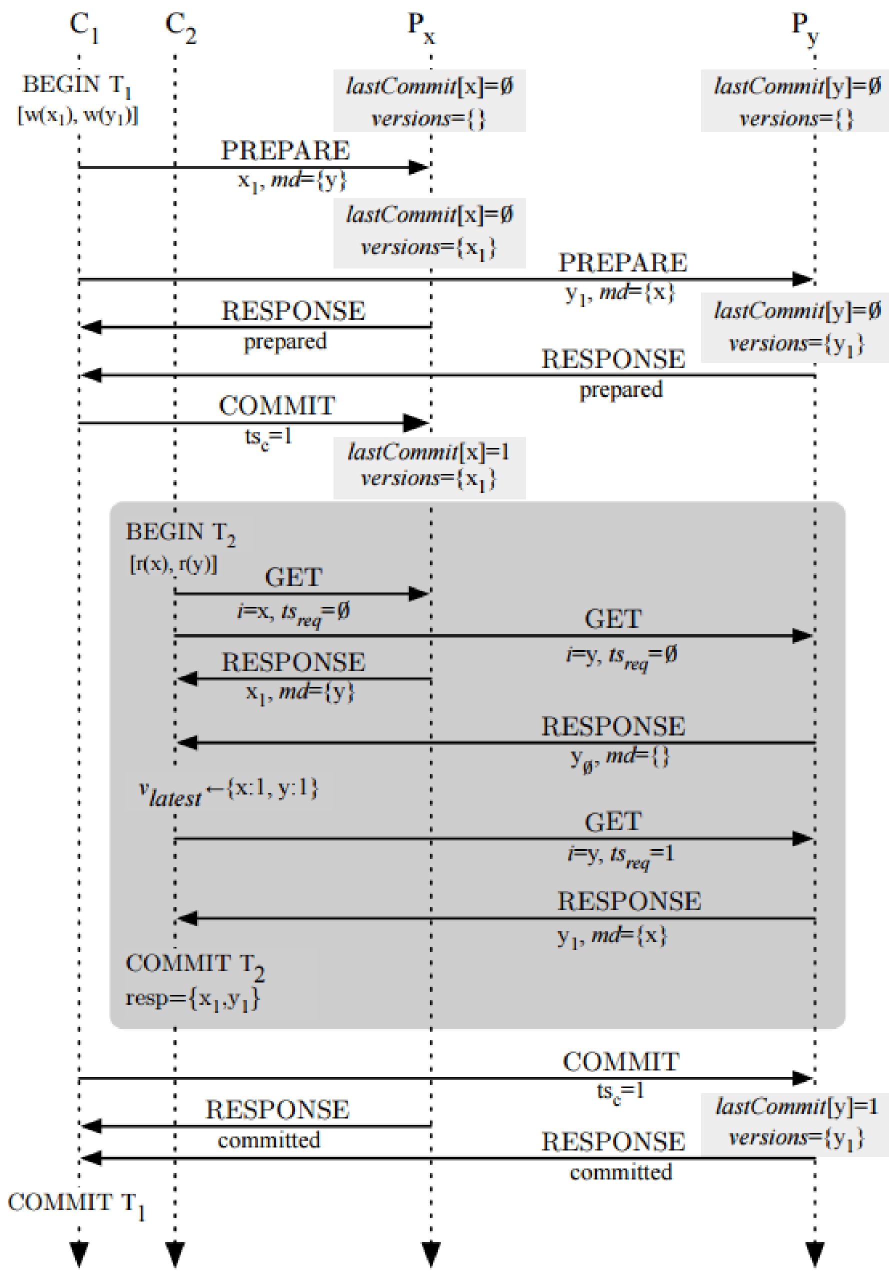
How we go about it

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

RAMP Execution & Algorithm



Algorithm 1 RAMP-Fast

Server-side Data Structures

$versions$: set of versions $\langle item, value$, timestamp ts_v, metadata $md\rangle$
$latestCommit[i]$: last committed timestamp for item i

Server-side Methods

procedure PREPARE(v : version)
    $versions$.add(v)
    return

procedure COMMIT(ts_c : timestamp)
    $I_{ts} \leftarrow \{w.item \mid w \in versions \wedge w.ts_v = ts_c\}$
    $\forall i \in I_{ts}, latestCommit[i] \leftarrow \max(latestCommit[i], ts_c)$

procedure GET(i : item, ts_{req} : timestamp)
    if $ts_{req} = \emptyset$ **then**
        return $v \in versions : v.item = i \wedge v.ts_v = latestCommit[item]$
    else
        return $v \in versions : v.item = i \wedge v.ts_v = ts_{req}$

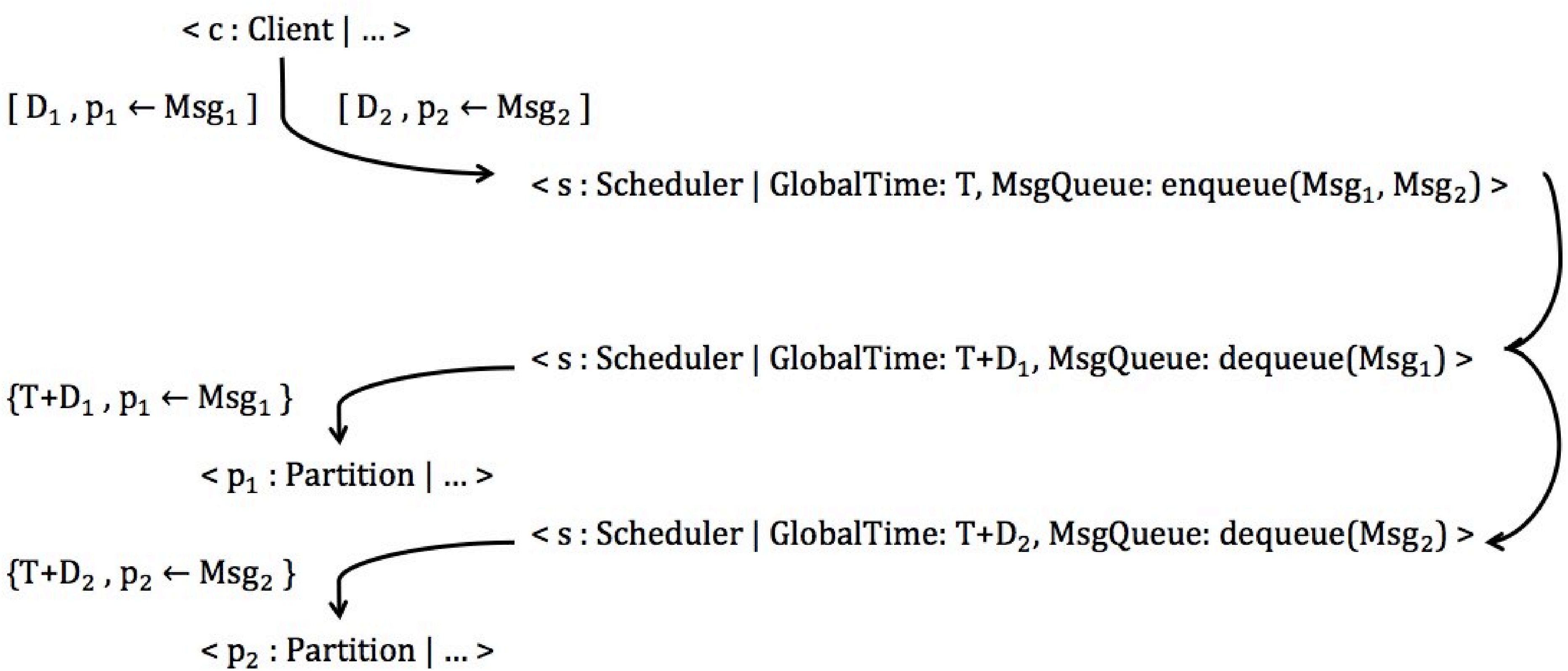
Client-side Methods

procedure PUT_ALL(W : set of $\langle item, value\rangle$)
    $ts_{tx} \leftarrow$ generate new timestamp
    $I_{tx} \leftarrow$ set of items in W
    parallel-for $\langle i, v\rangle \in W$
        $v \leftarrow \langle item = i, value = v, ts_v = ts_{tx}, md = (I_{tx} - \{i\})\rangle$
    invoke PREPARE(v) on respective server (i.e., partition)
    parallel-for server s : s contains an item in W
        invoke COMMIT(ts_{tx}) on s

procedure GET_ALL(I : set of items)
    $ret \leftarrow \{\}$
    parallel-for $i \in I$
        $ret[i] \leftarrow$ GET($i, \emptyset$)
    $v_{latest} \leftarrow \{\}$ (default value: -1)
    for response $r \in ret$ **do**
        for $i_{tx} \in r.md$ **do**
            $v_{latest}[i_{tx}] \leftarrow \max(v_{latest}[i_{tx}], r.ts_v)$
    parallel-for item $i \in I$
        if $v_{latest}[i] > ret[i].ts_v$ **then**
            $ret[i] \leftarrow$ GET($i, v_{latest}[i]$)
    return ret

RAMP Model in Maude

- The distributed state of RAMP model is a “multiset” of Partitions, Clients, Scheduler and Messages



RAMP Model in Maude

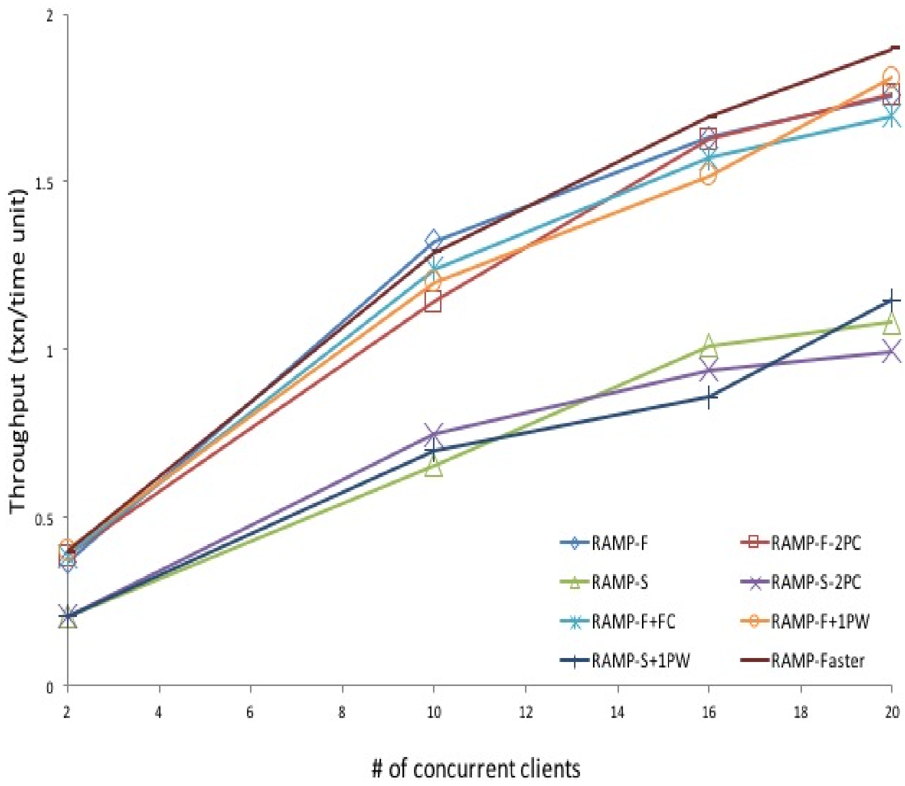
- Example (RAMP-Faster): on receiving PREPARE message

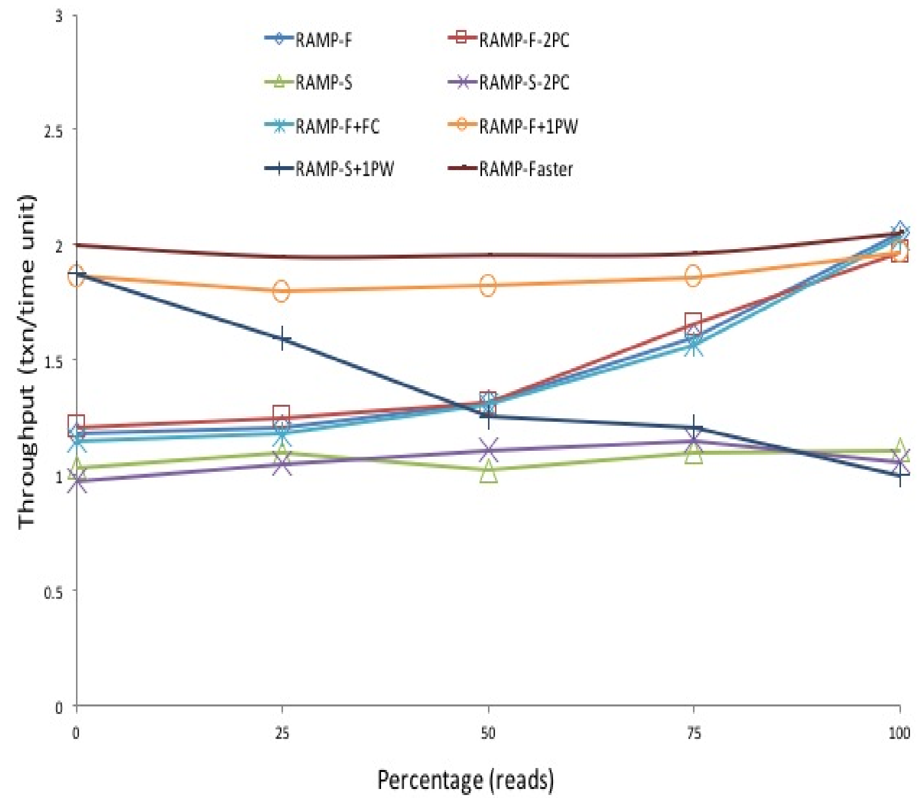
```
crl [receive-prepare-faster] :
    {T, O <- prepare(TID,ID,X,V,ts(O’,SQN,MD,O’))}
    < O : Partition | versions: VS, latestCommit: LC >
=>
    < O : Client | versions: VS’, latestCommit: cmt(LC,VS’,ts(O’,SQN)) >
    [D, O’ <– committed(TID,ID,O)]
if VS’ := (v(X,V,ts(O’,SQN),MD),VS) with probability D := distr(…) .
```

How realistic and informative are the results?

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

Performance: Throughput



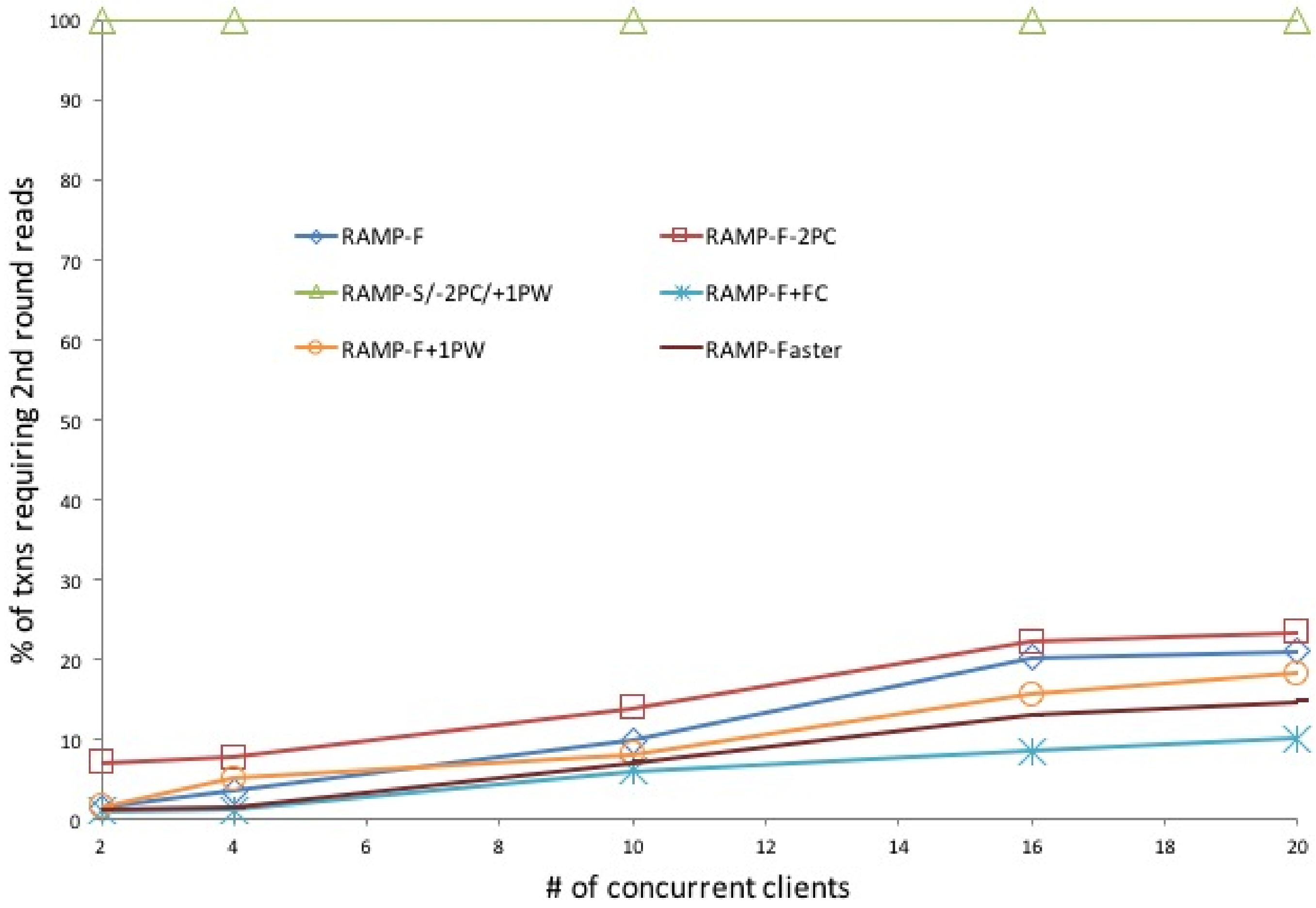


- As client (or read) workload increases, RAMP-F's throughput increases
- …

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

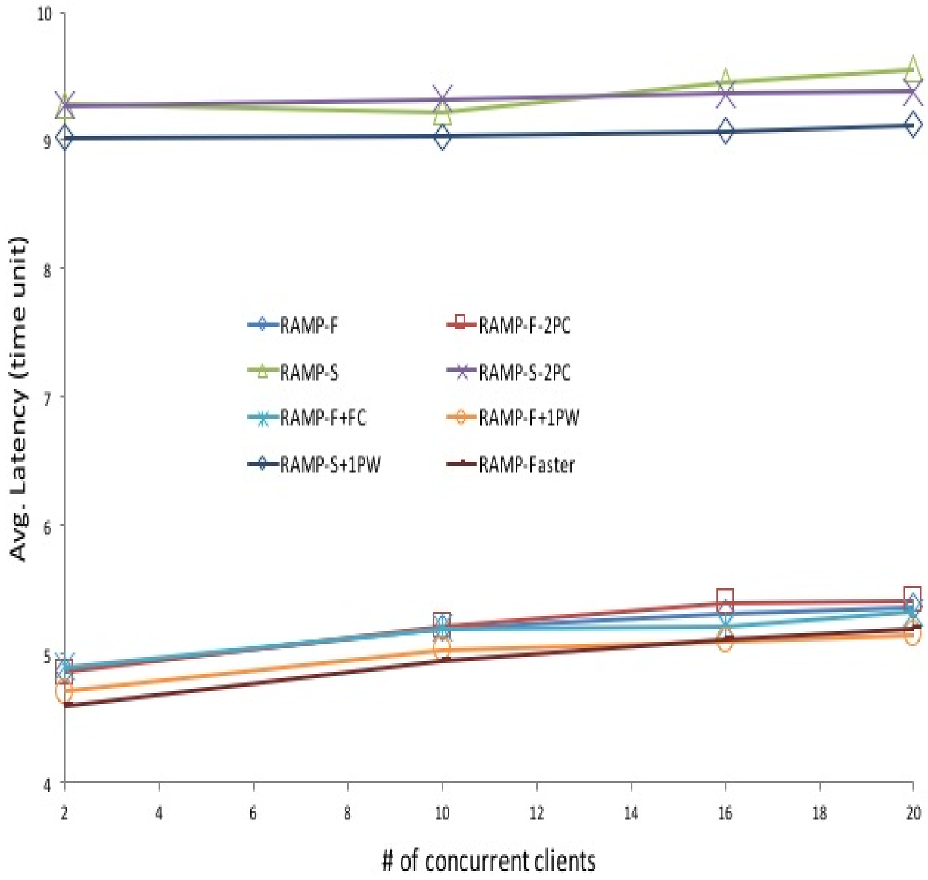


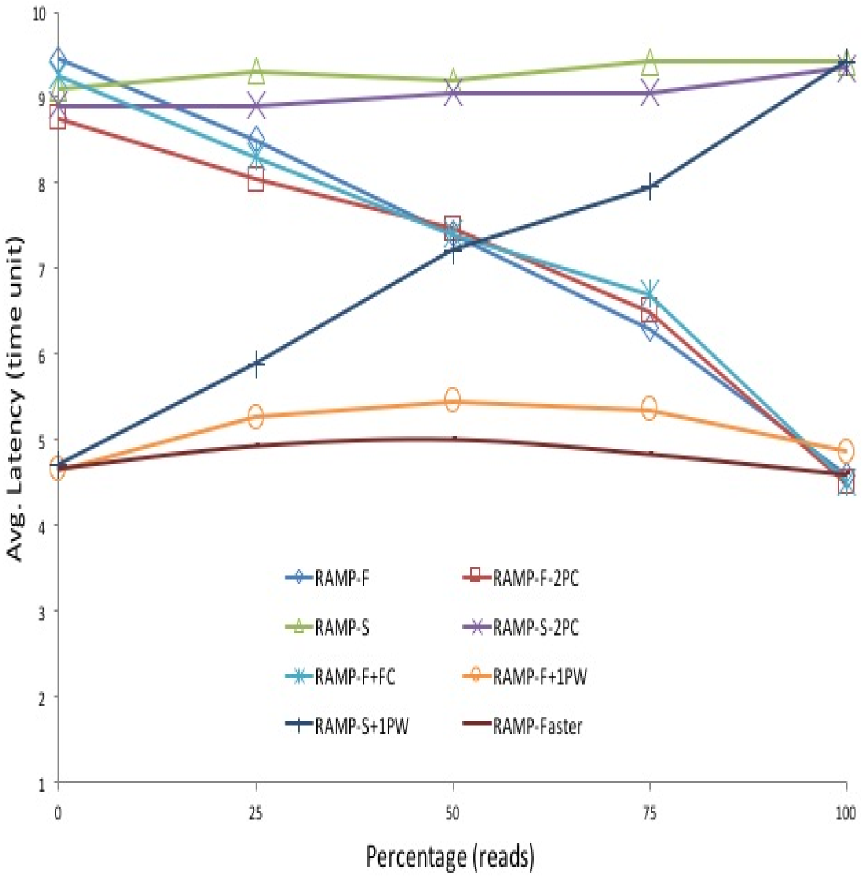
- RAMP-F+FC requires less second-round reads than RAMP-F
- …

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

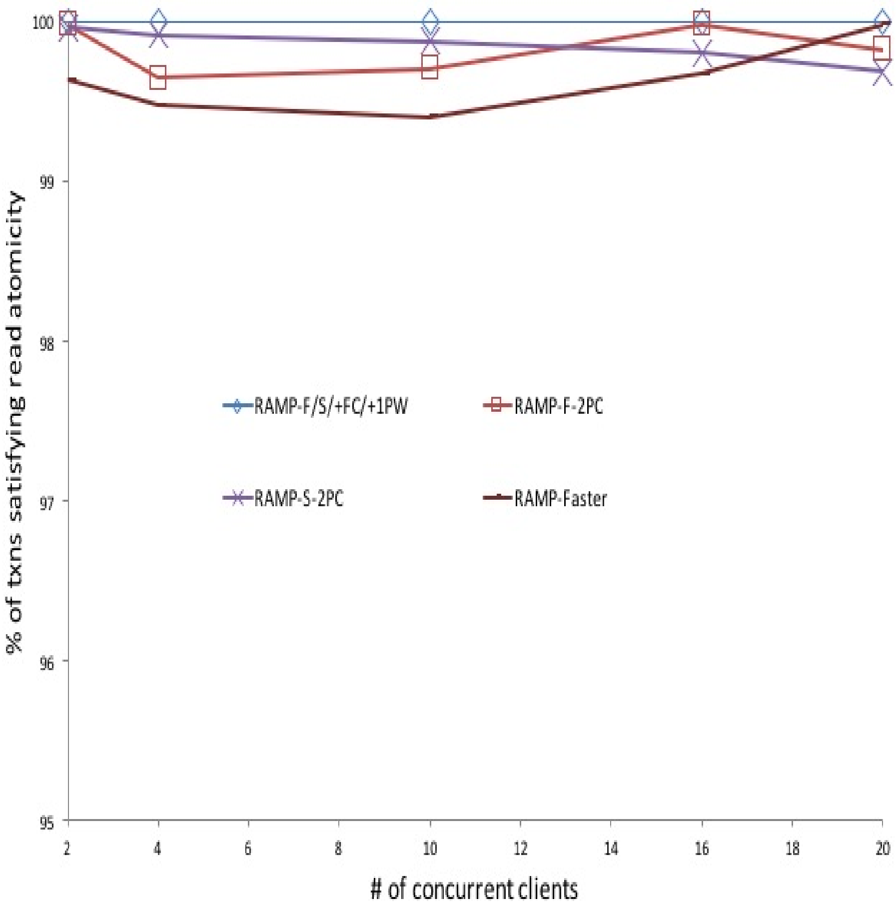
Performance: Avg. Latency

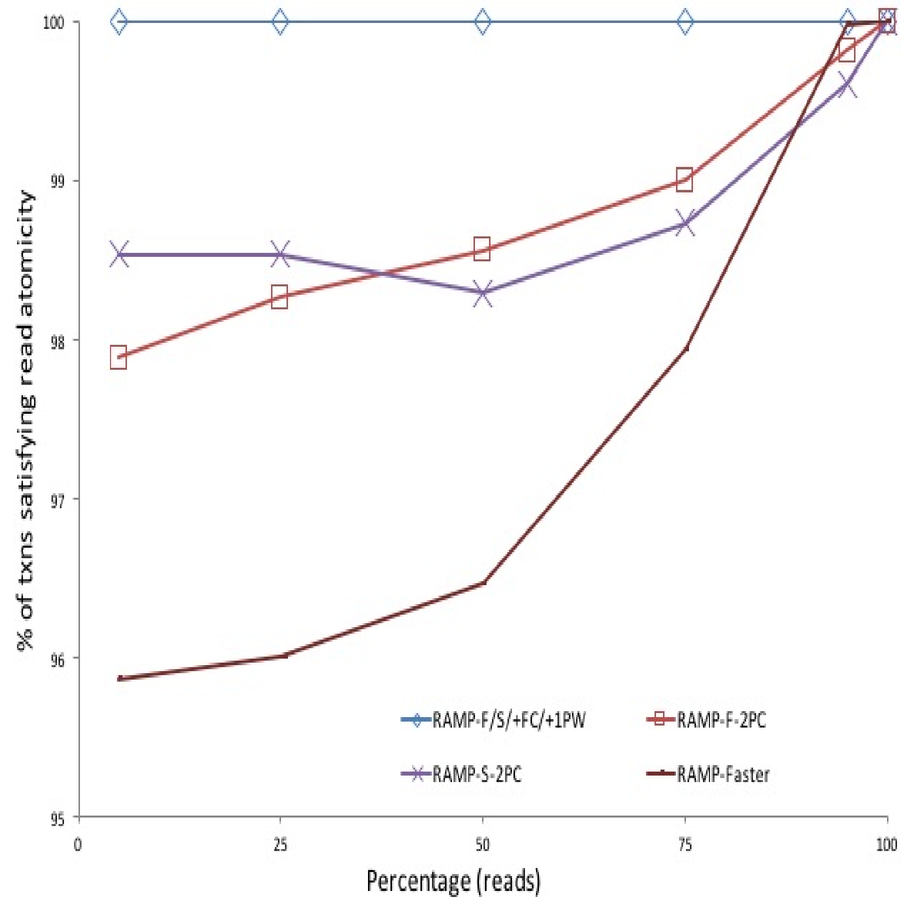




- RAMP-Faster incurs the lowest avg. latency among all RAMP versions
- …

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