Quantifying state staleness in a distributed resource management control plane

Harshit Gupta, Salini Mishra, Lucia Verdejo Estevez

CS 4420/6422 Database System Implementation

Prof. Joy Arulraj

Problem addressed

- scheme for coordinated management of multiple clusters via state synchronizations
- 2 settings : single-controller and multi-controller settings
- high-level metrics
    - The kind of staleness guarantees does the placement algorithm expect ?
    - Probability of providing staleness guarantees.
    - Overhead of state synchronization.
    - Impact on high level request processing latency.

Assumptions

Network Latency

Network latency between the controller instance and servers are drawn from independent and identical Gaussian distributions L with mean L_{mean} and standard deviation L_{stddev}

Requests and ITT

Requests arrive to the controller instance based on a Poisson process with mean inter-arrival time equal to $t_{IT\ T}$ (inter-transaction time)

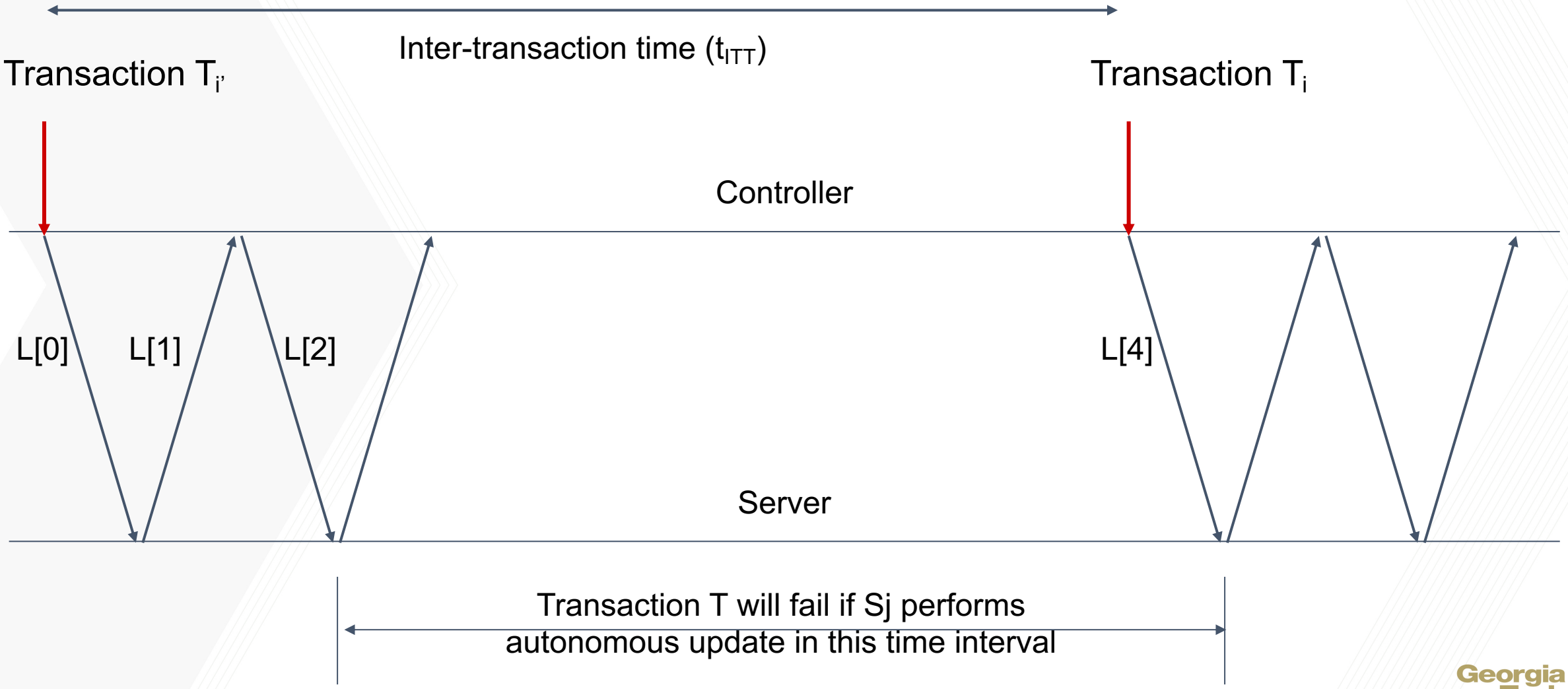
Autonomy of the servers

1. make autonomous decisions without the intervention of the controller
2. λ_{auto}, which indicates the ratio of autonomous decisions made by servers

Window of vulnerability

To quantify state staleness in the controller, we formulated the time-frame in which the controller could become stale, which will be referred as **window of vulnerability.**

Quantifying the window of vulnerability



Georgia Tech
CREATING THE NEXT

Autonomous resource allocation update model

- Per-server autonomous updates follow a continuous-time Markov chain

- Prob (autonomous update in time interval t)
  $= 1 - e^{-\lambda.t}$ [λ = rate of autonomous update]

Probability of transaction success

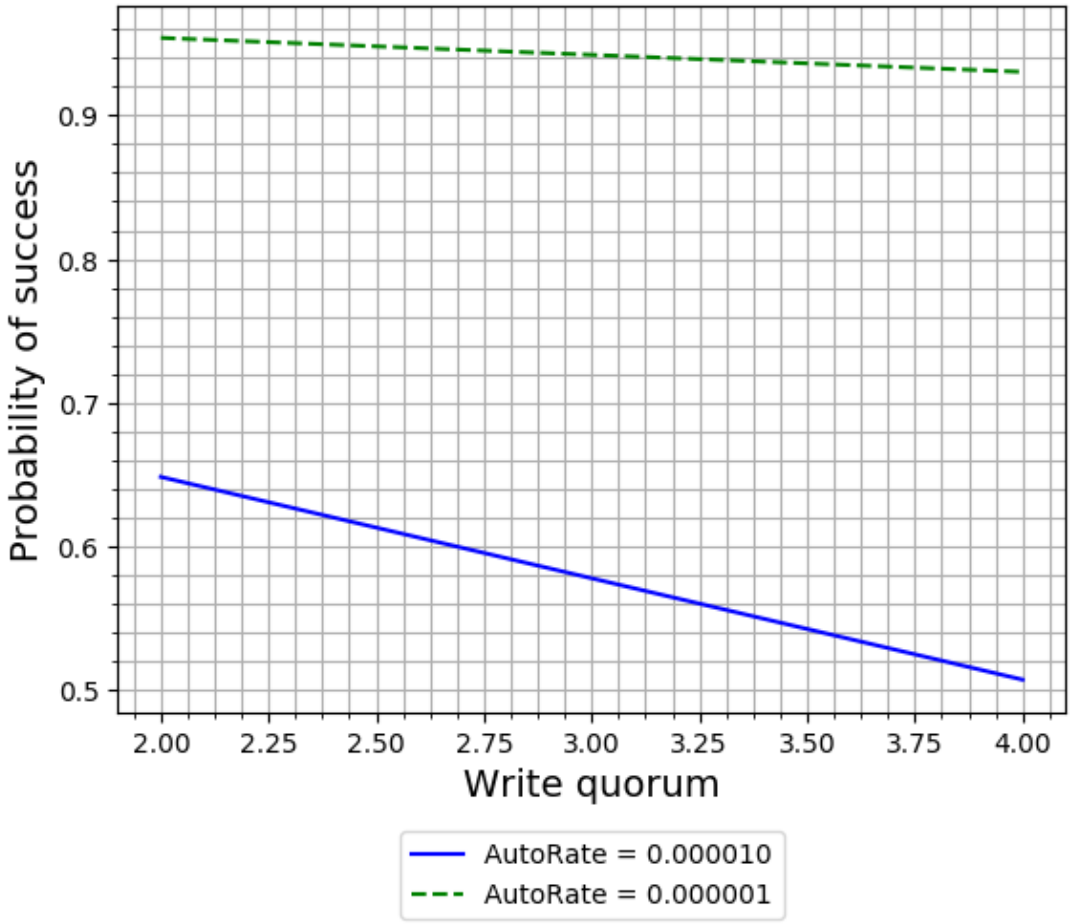
Prob (T_i succeeds) = Prob (S1 is unchanged AND
S2 is unchanged AND .....
Sw is unchanged)

Prob (Sj is unchanged) = 1 - Prob (Sj has changed)
$= \Sigma_k$ { Prob (Sj changed | T_{i-k} was last to update Sj) x
Prob (T_{i-k} was last to update Sj) }

Prob (Sj changed | T_{i-k} was last to update Sj) = $1 - e^{-\lambda . T}$
where $T = k.ITT - L[0] - L[1] - L[2] + L[4]$

Prob (T_{i-k} was last to update Sj) = $({}^{n-1}C_w / {}^{n}C_w)^{k-1} . (1 - {}^{n-1}C_w / {}^{n}C_w)$

Variation of transaction success probability



System Configuration

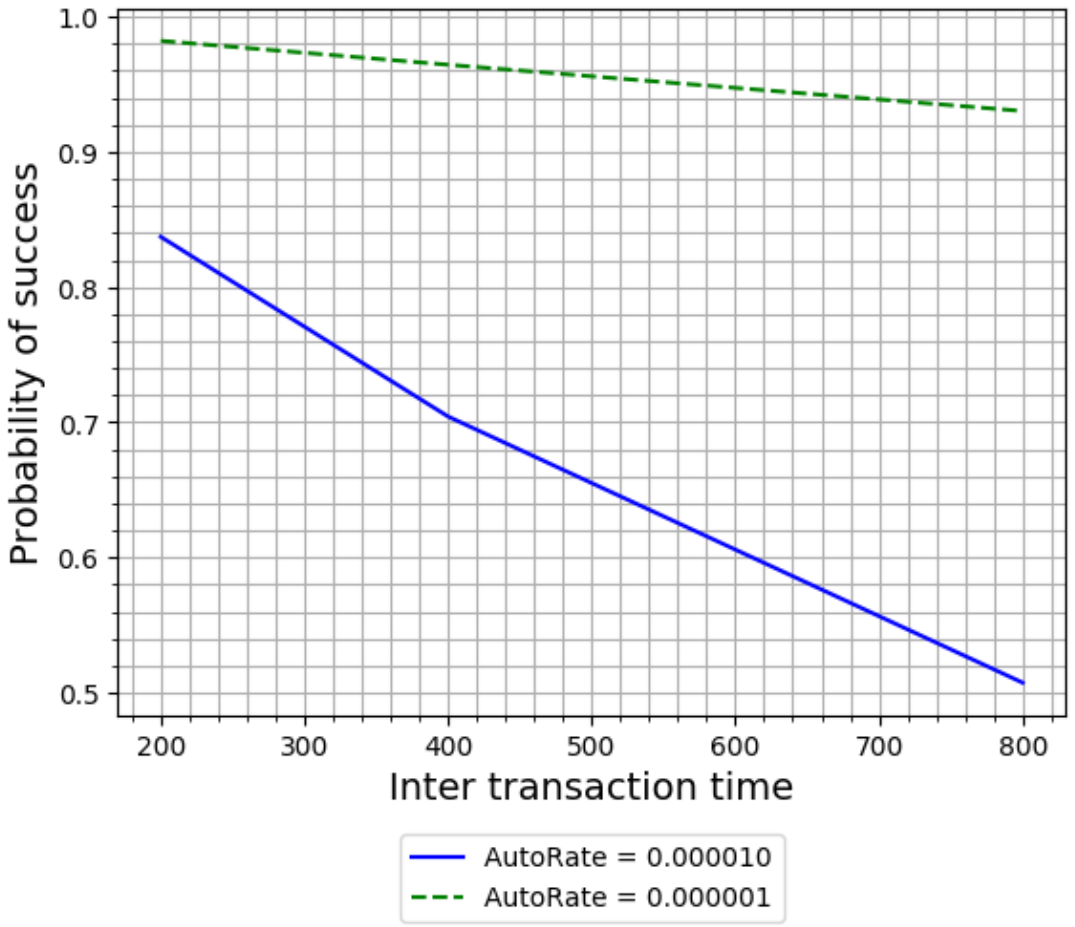
- 100 Servers
- Inter-transaction arrival time = 800ms
- One-way controller-server latency = 20ms

Conclusion : Increasing write set size (w) decreases likelihood of success

⇒ As there are more sources of autonomous allocation updates

Georgia Tech
CREATING THE NEXT

Variation of transaction success probability



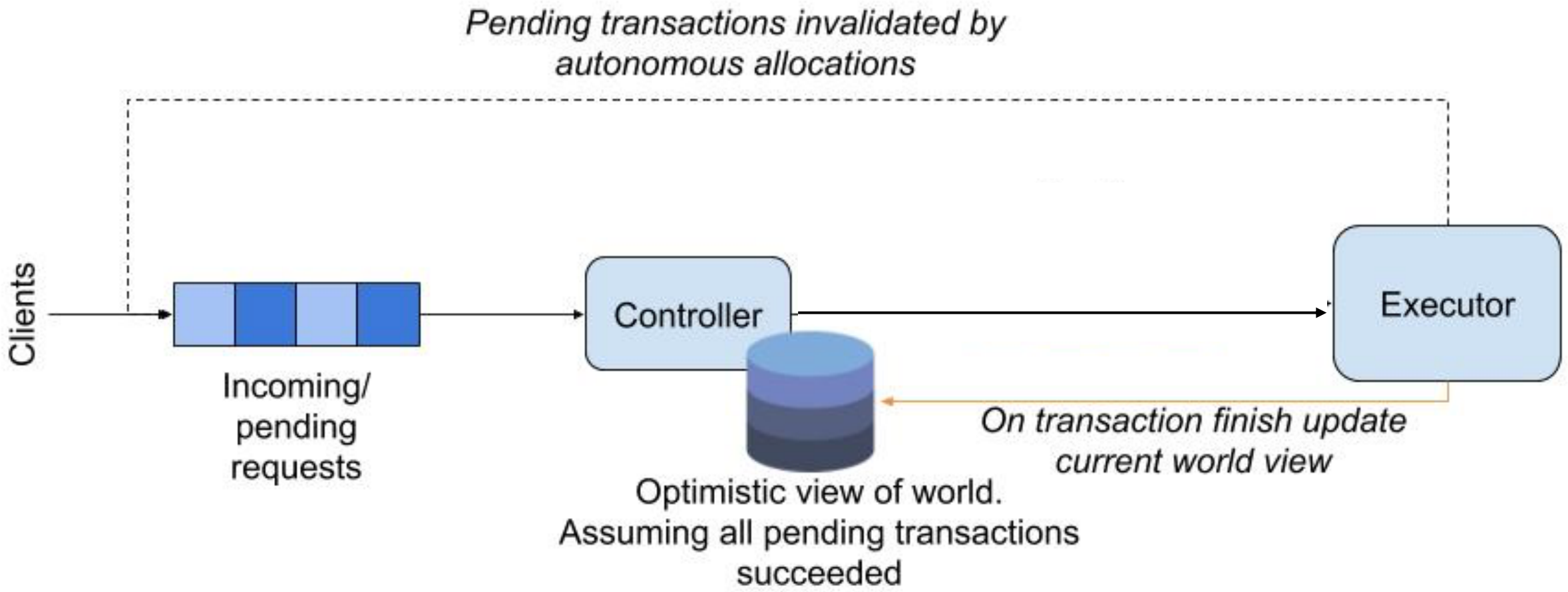
System Configuration

- 100 Servers
- Write-set size = 4
- One-way controller-server latency = 20ms

Conclusion : Increasing inter-transaction time decreases success likelihood

⇒ As window of conflict is higher

Single controller implementation : architecture



Issues with current model

1. Transaction inter-arrival time is constant
    a. Not the case in reality
2. Does not take into account failed transactions
    a. As it does not support retries of transactions
3. Cannot quantify the end-to-end transaction execution time

End-to-end system simulator

Assumptions :

1. Transaction arrival follows a Poisson process (with mean ITT)
2. Autonomous resource allocation updates follow Poisson process
3. There is a single transaction executing at a time
    a. Transaction finishes only after the two phases of a successful 2PC execution
    b. Subsequent transactions have to queue and wait
4. If the 2PC execution fails, the transaction has to retry
5. Explicitly keeps track of each server's window of vulnerability
    a. Hence, calculation of P(S_j changed) is straightforward

Comparing the model to system simulator

	Model	Simulator
N=100 W=4 itt=200 auto_rate=0.00001 lat_mean=40	0.825	0.841
N=100 W=4 itt=400 auto_rate=0.00001 lat_mean=40	0.685	0.709
N=100 W=4 itt=800 auto_rate=0.00001 lat_mean=40	0.483	0.496
N=100 W=4 itt=1600 auto_rate=0.00001 lat_mean=40	0.259	0.248
N=100 W=4 itt=3200 auto_rate=0.00001 lat_mean=40	0.093	0.058
N=100 W=8 itt=200 auto_rate=0.00001 lat_mean=40	0.826	0.832
N=100 W=8 itt=400 auto_rate=0.00001 lat_mean=40	0.681	0.690
N=100 W=8 itt=800 auto_rate=0.00001 lat_mean=40	0.468	0.475
N=100 W=8 itt=1600 auto_rate=0.00001 lat_mean=40	0.232	0.215
N=100 W=8 itt=3200 auto_rate=0.00001 lat_mean=40	0.066	0.051

Implementation nuances

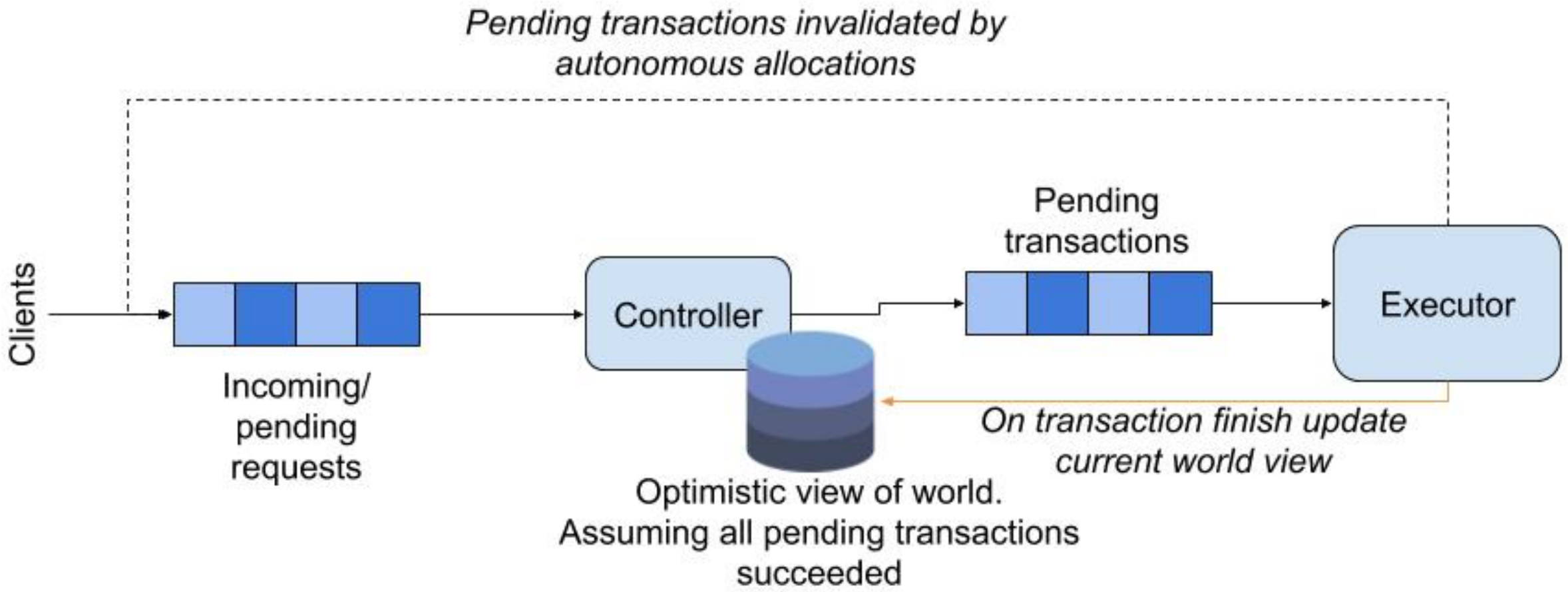
- Python Library used: ZeroMQ
  - ØMQ pattern used : Request-reply mechanism
  - connects a set of clients to a set of services; remote procedure call and task distribution pattern.
  - Socket used in request-reply mechanism : DEALER -> DEALER
  - full asynchronous (non blocking), sending any number of replies back

Implementation nuances

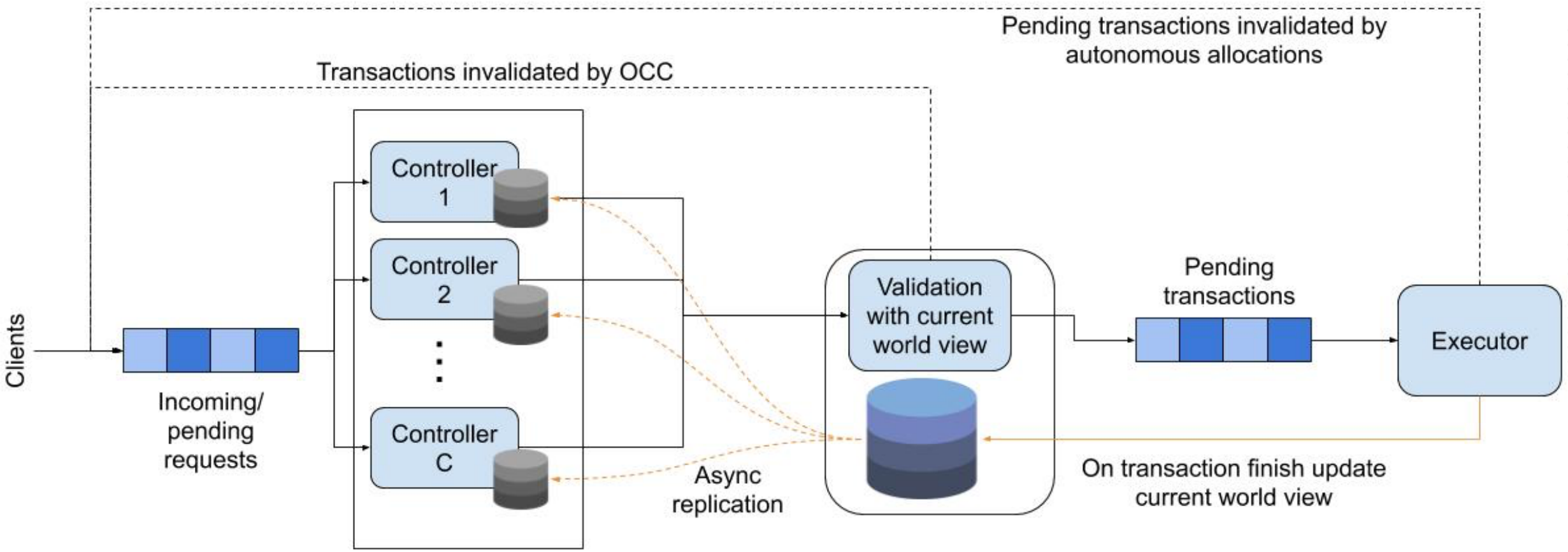
- Google protocol buffers
  - Information to be serializing structured by defining protocol buffer message types in .proto files.
  - Each protocol buffer message: logical record of information, containing a series of name-value pairs

```
syntax = "proto2";
option optimize_for = LITE_RUNTIME;
import "proto/commit-protocol.proto";

message InitServer {
    optional string server_url = 1;
    optional uint32 cpu = 2;
    optional uint32 memory = 3;
}

message AppRequest {
    optional string app_id = 1;
}

message Message {
    enum Type {
        INIT_SERVER = 1;
        APP_REQ = 2;
        COMMIT_PROTOCOL = 3;
    }

    required Type type = 1;
    optional string src = 2;
    optional string dst = 3;
    optional uint64 time_to_send = 4;
    optional InitServer init_server = 5;
    optional AppRequest app_request = 6;
    optional CommitProtocolMessage cp_msg = 7;
}
```

Single controller implementation : evolved architecture



Multi-controller system



Reasons for transaction failure

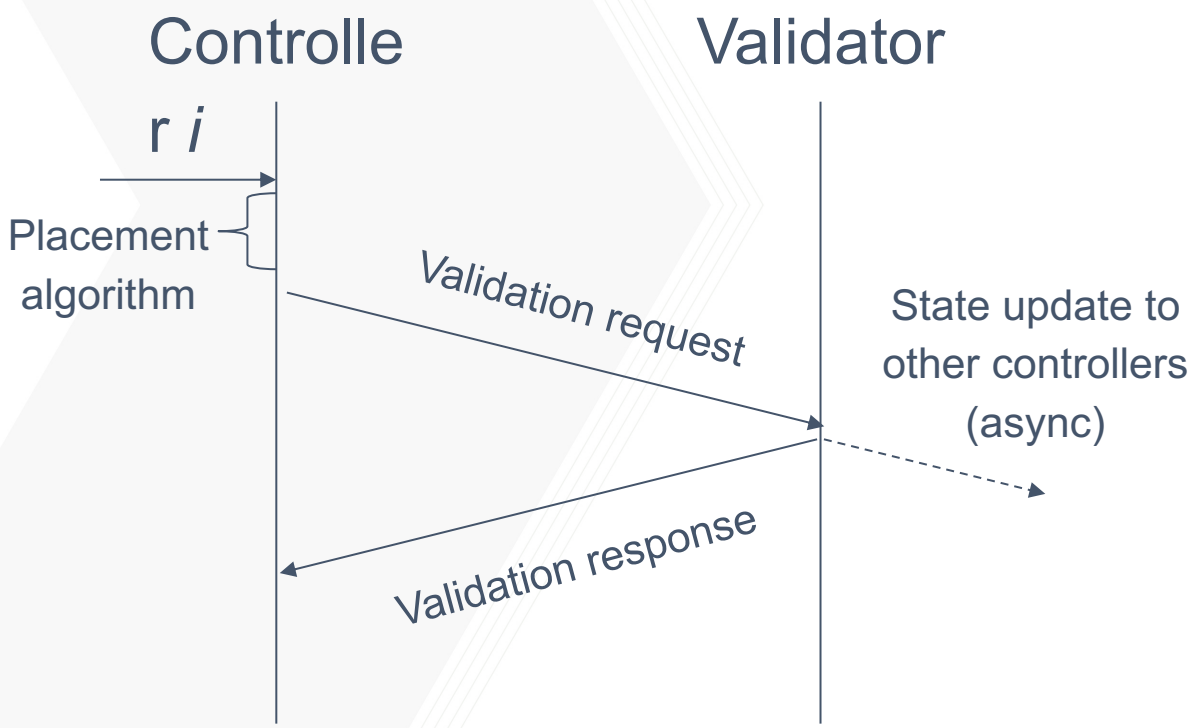
- Conflict with other controllers
  - Other controllers operate on same shared, but loosely-sync state
  - Pitfalls of optimistic concurrency control
- Controller’s state is stale due to autonomous allocations by servers
  - Similar to single-controller model

Optimistic Concurrency Control

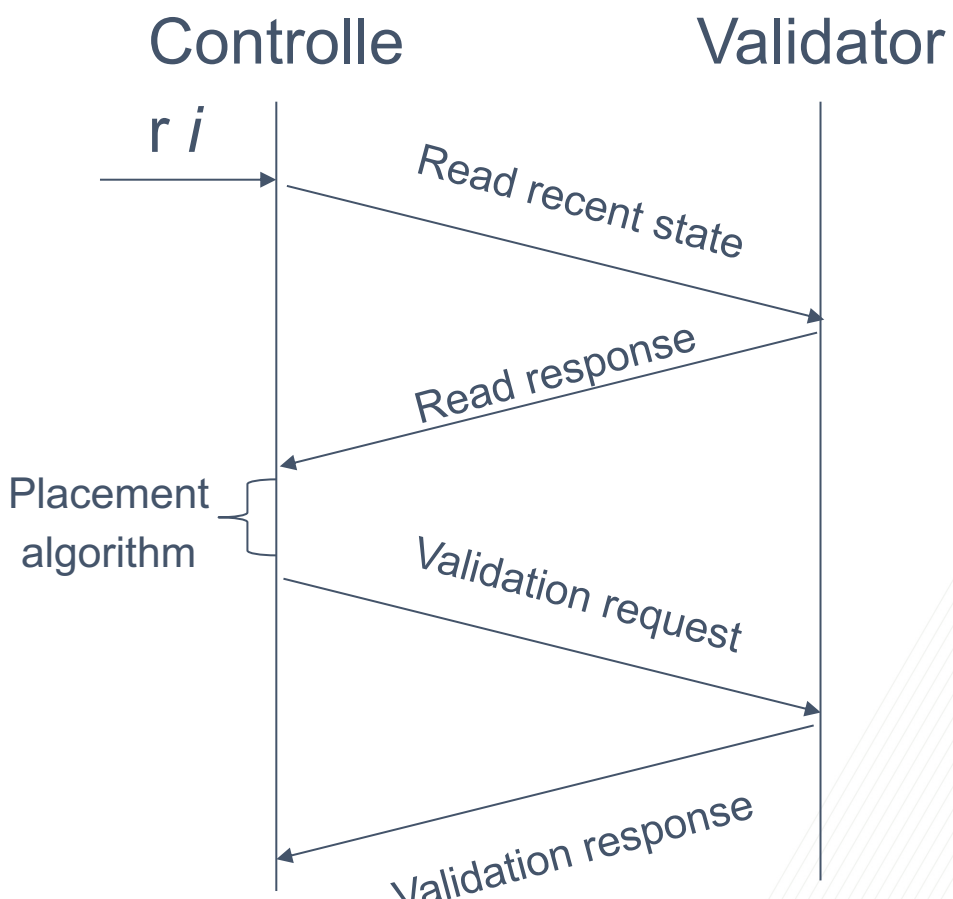
- addresses conflicts with the simultaneous accessing or altering of data that can occur with a multi-user system: multiple transactions can frequently complete without interfering with each other
- While running, transactions use data resources **without acquiring locks** on those resources.
- Before committing, each transaction verifies that no other transaction has modified the data it has read.

Two flavours of optimistic concurrency control

Variant 1
Each controller maintains personal copy of state



Variant 2
Single shared-copy of state



Georgia Tech
CREATING THE NEXT

Two flavours of optimistic concurrency control

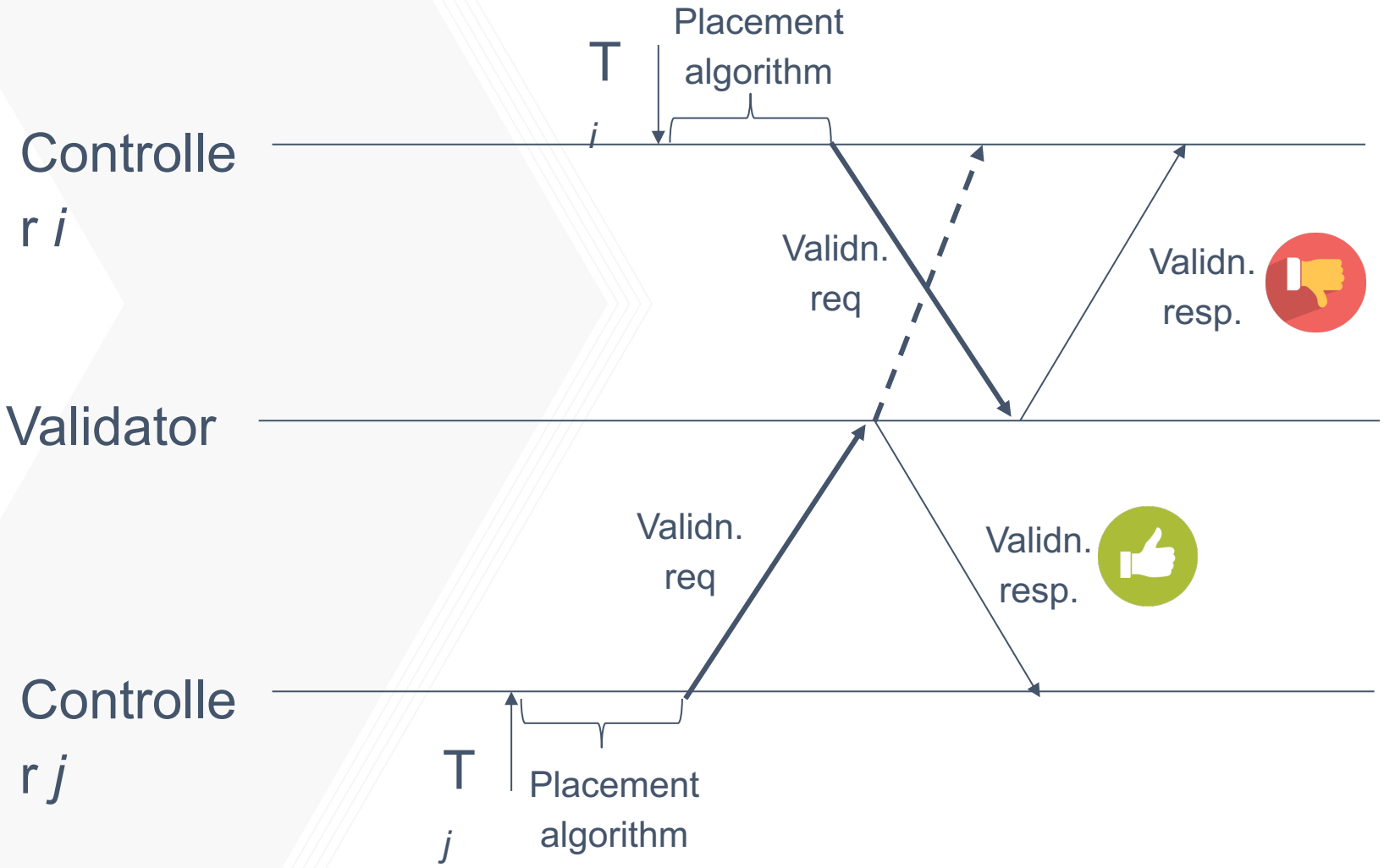
Variant 1
Each controller maintains personal copy of state

- Smaller runtime of execution + validation process
  - Single round-trip to validator
- *Higher conflict probability ???*

Variant 2
Single shared-copy of state

- Higher runtime of execution + validation process
  - Two round trips
- *Window of conflict is smaller ??*

Variant 1 : Each controller maintains copy of state



Conditions for T_i to fail

There exists a Tj such that

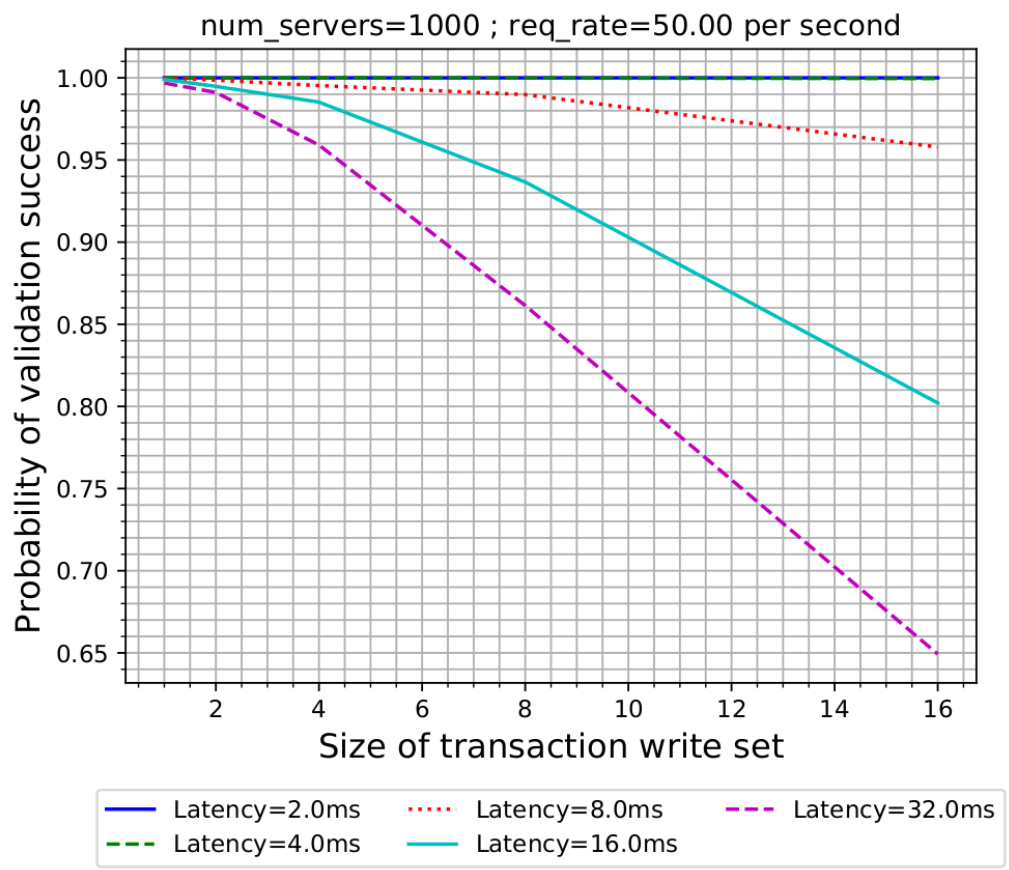
1. T_j is validated before T_i
2. T_i does not see T_j's state update
3. Tj was successfully validated
4. Tj and Ti conflict (update common servers)

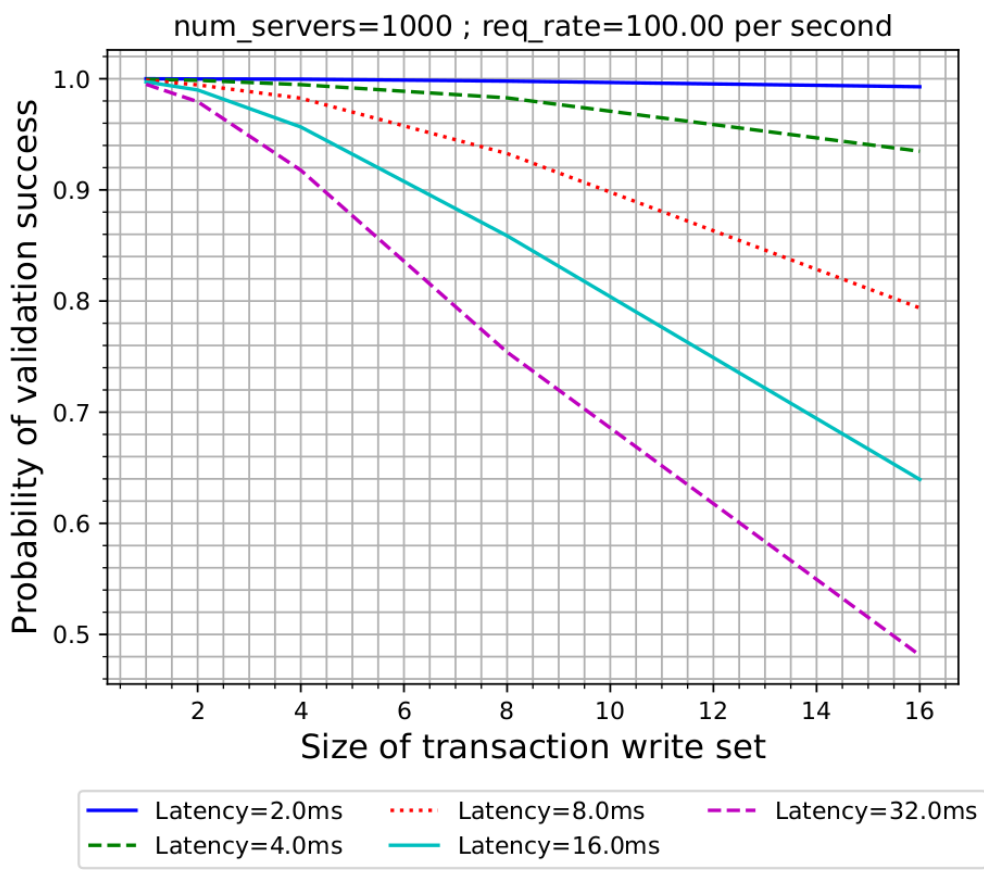
Georgia Tech
CREATING THE NEXT

Variant 1 : Each controller maintains copy of state

- Use Monte-Carlo simulation method
- Sample transaction arrival times from Poisson process
- Sample controller-validator latencies from Gaussian distribution
- For each transaction Ti that attempts to validate itself
  - We check if there exists a Tj that satisfied the aforementioned conditions

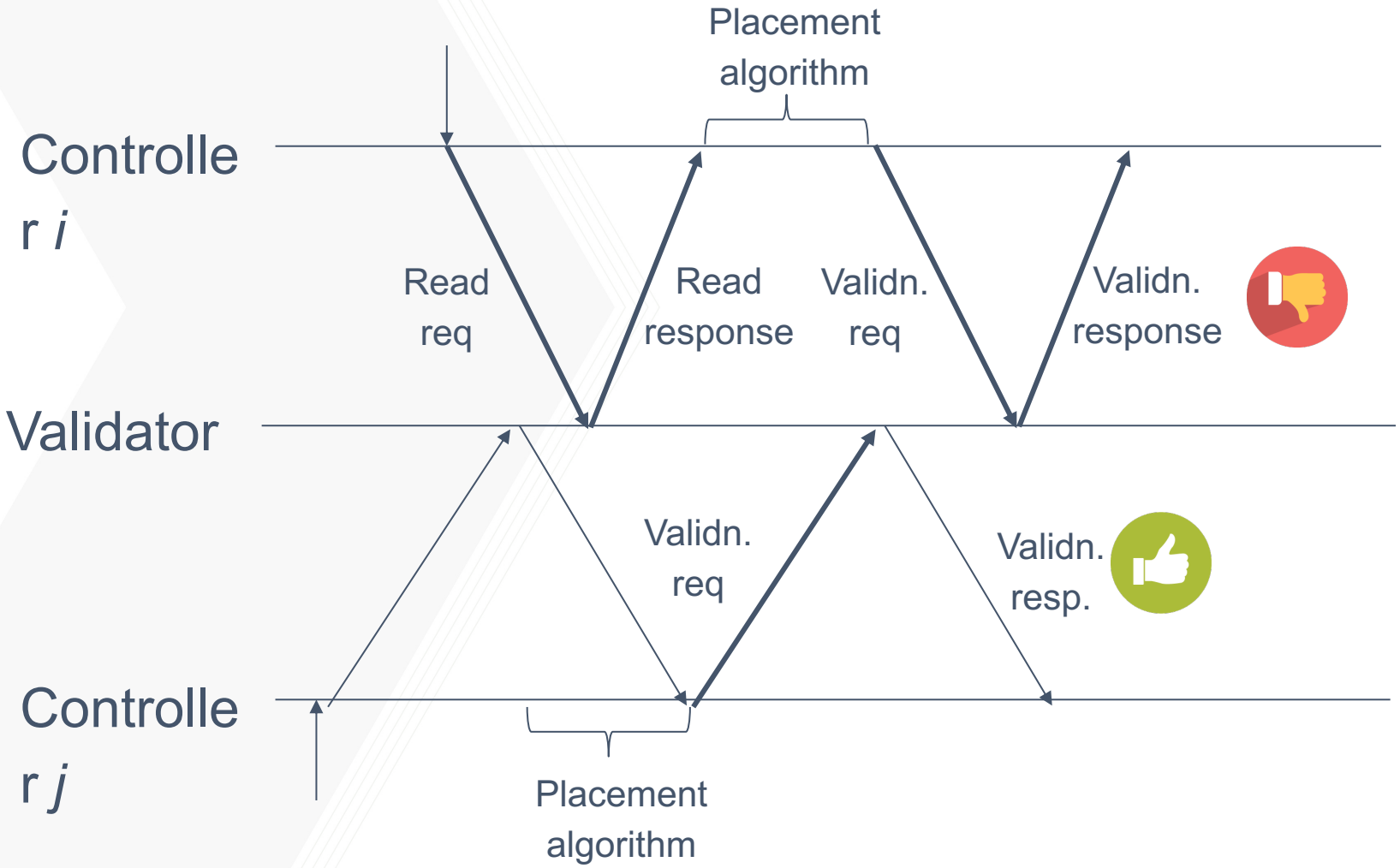
Variant 1 : Each controller maintains copy of state





Georgia Tech
CREATING THE NEXT

Variant 2 : Single shared-copy of state



Conditions for T_i to fail

1. T_j is validated before T_i
2. T_j validates between Ti's read and validate req
3. Tj was successfully validated
4. Tj and Ti conlict (update common servers)

Georgia Tech
CREATING THE NEXT

Variant 2 : Single shared-copy of state

For a transaction T_i that attempts validation, there have been M transactions that successfully validated between T_i's read and validation request.

$$\mathbb{E}[M] = \left(\frac{M-1}{M}\right) \cdot \lambda \cdot (2L) \cdot \mathbb{P}[\text{validation success}]$$

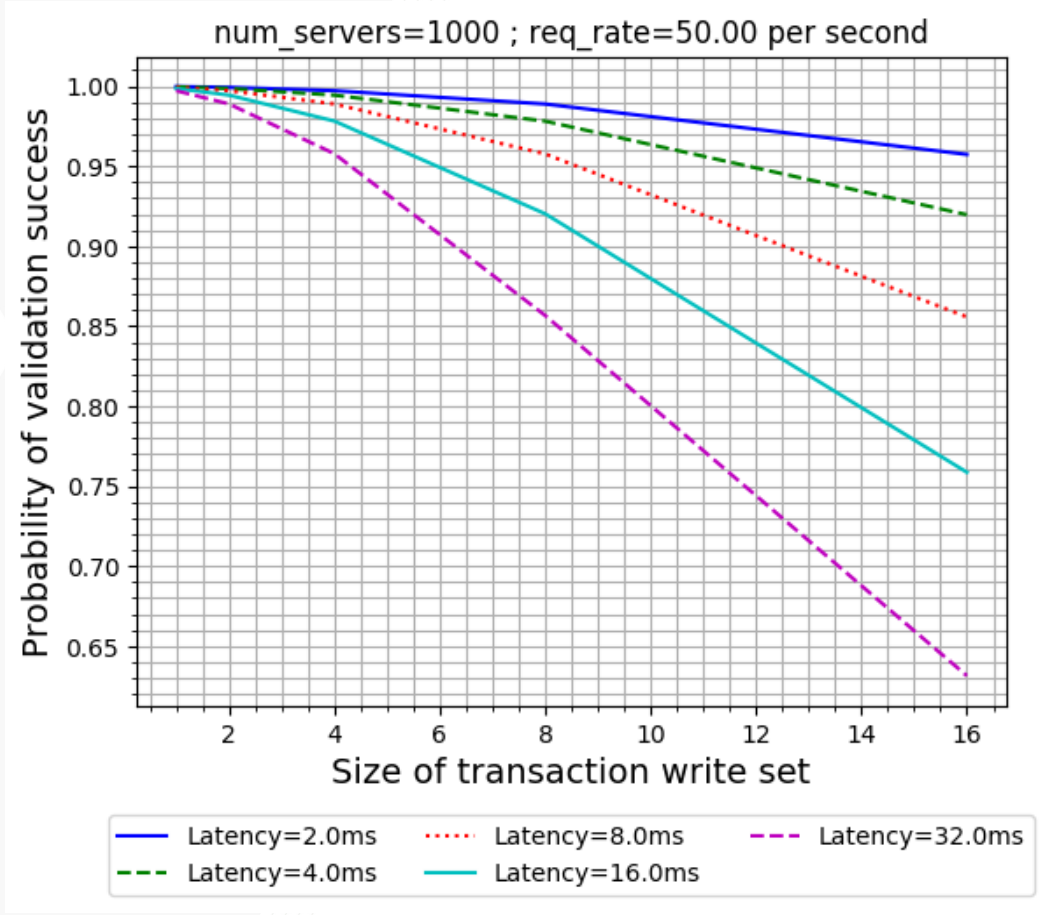
$$\mathbb{P}[T_i \text{ succeeds validation}] = \left(\frac{\binom{N-W}{W}}{\binom{N}{W}}\right)^{\mathbb{E}[M]}$$

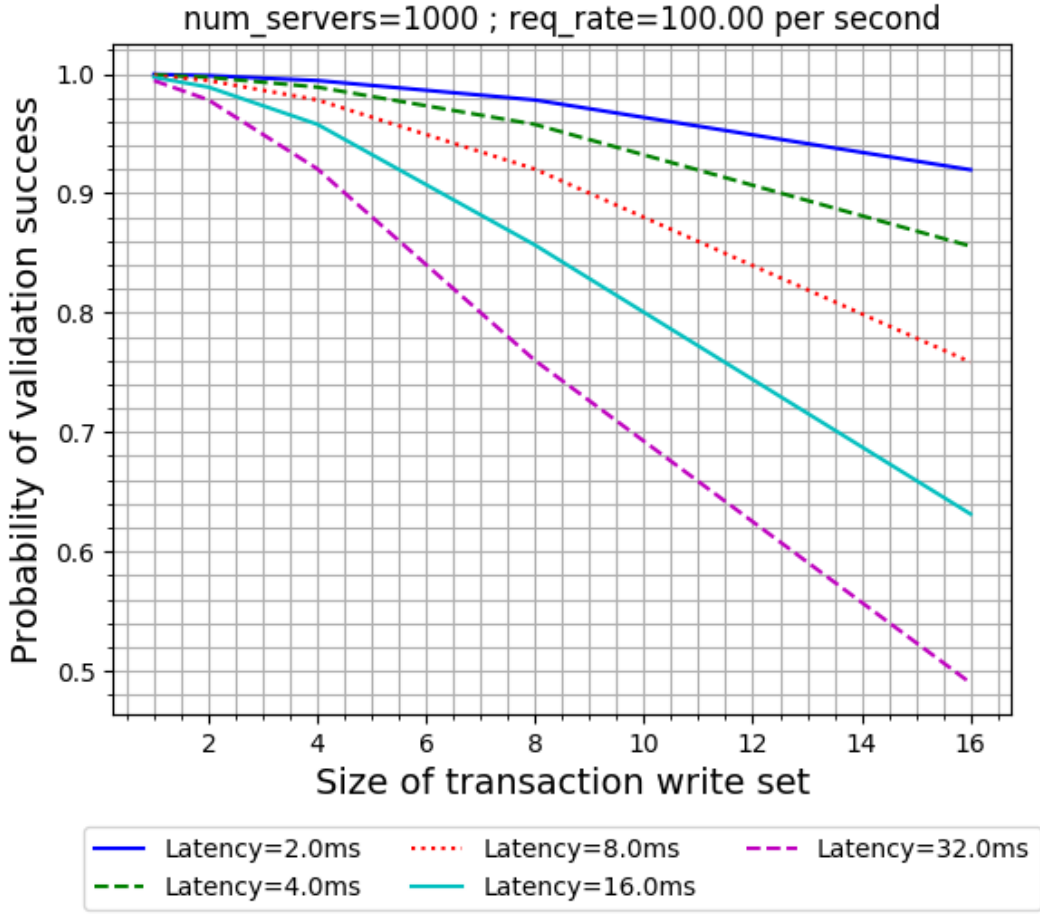
$$\mathbb{P} = \alpha^{\mathbb{P}}$$

where

$$\alpha = \left(\frac{\binom{N-W}{W}}{\binom{N}{W}}\right)^{\left(\frac{M-1}{M}\right) \cdot \lambda \cdot (2L)}$$

Variant 2 : Single shared-copy of state

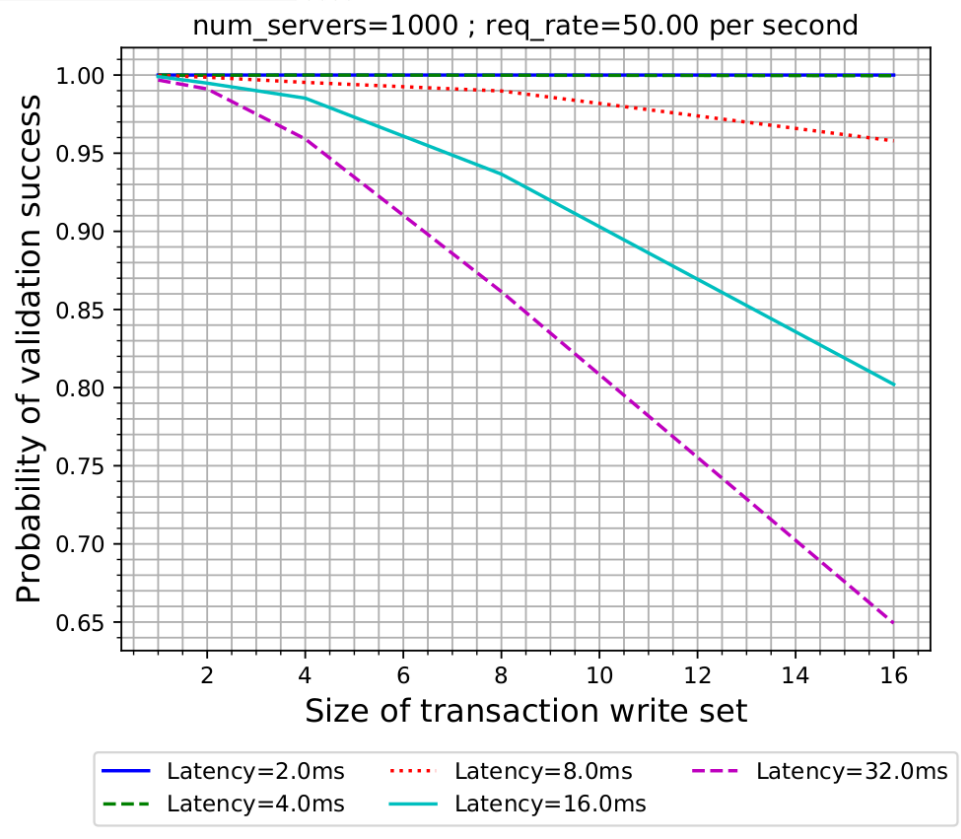




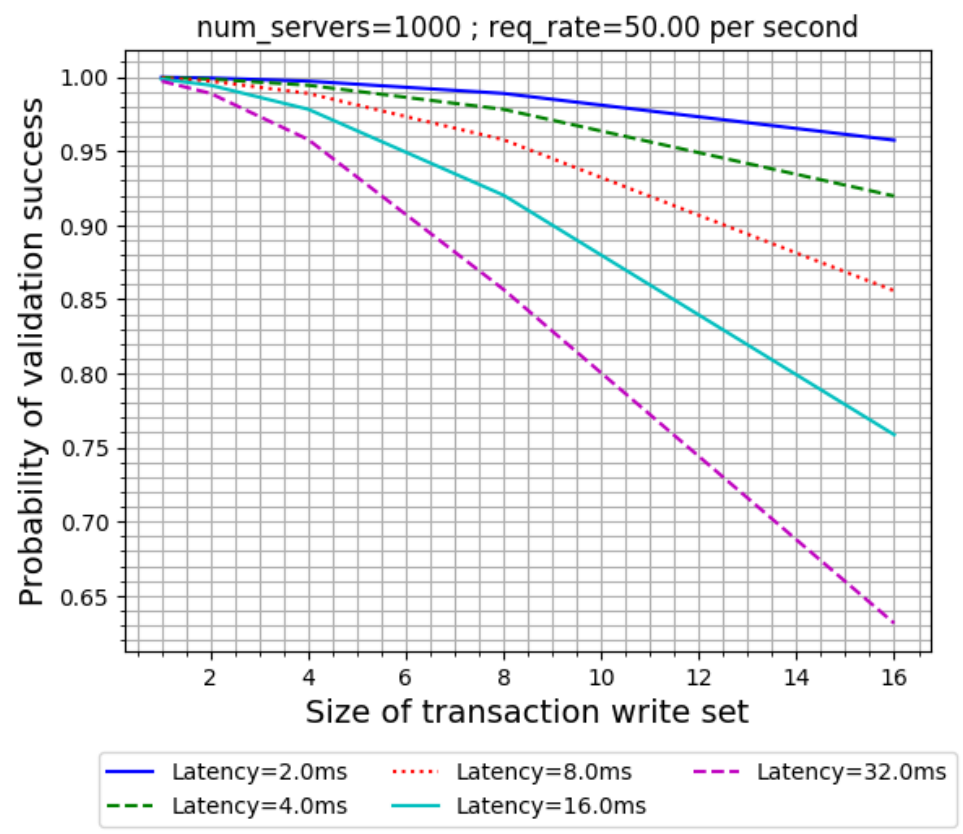
Georgia Tech
CREATING THE NEXT

Comparison of OCC variants

Async replicated state



Shared single-copy state

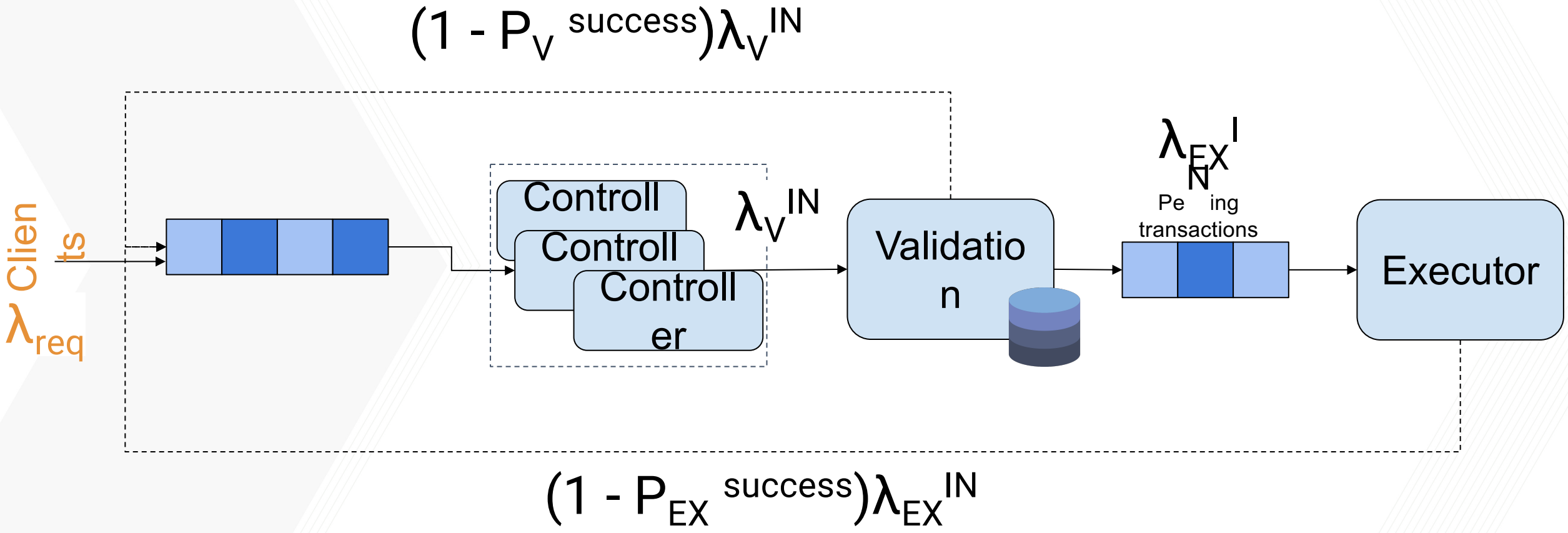


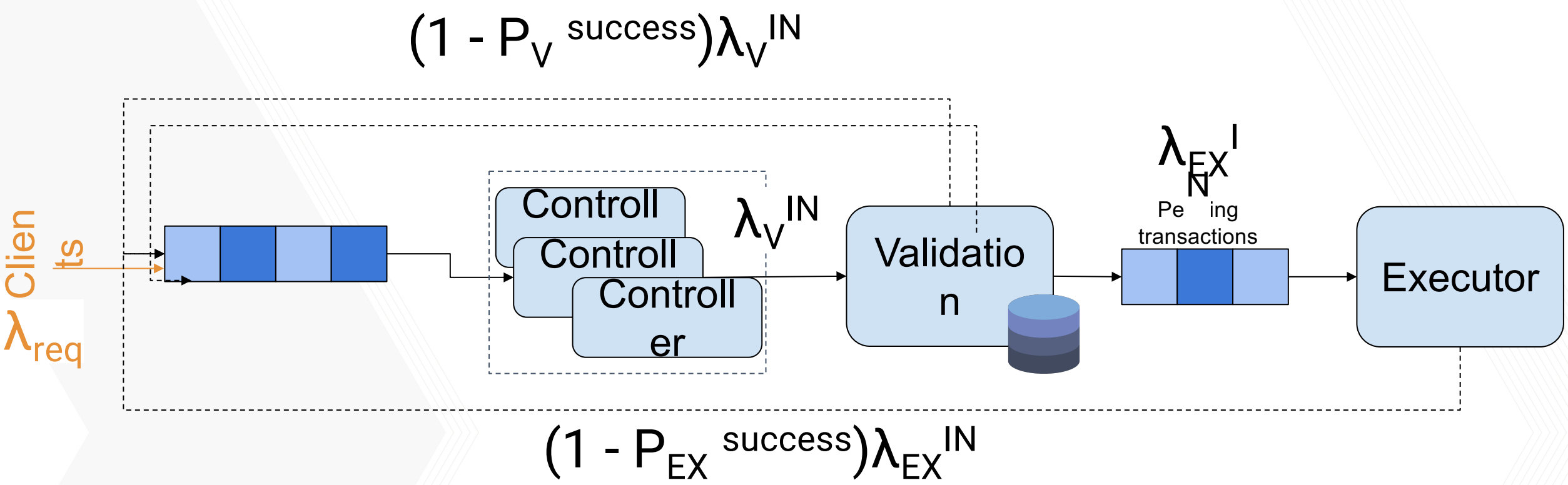
Variant 1 better for lower latency between controller & validator

For higher latencies, behaviours converge

Georgia Tech
CREATING THE NEXT

Big picture


$(1 - P_V^{success})\lambda_V^{IN}$
$\lambda_{Clients}$
λ_{req}
Controll
Controll
Controll
er
λ_V^{IN}
Validatio
n
λ_{EX}^{IN}
Pe ing
transactions
Executor
$(1 - P_{EX}^{success})\lambda_{EX}^{IN}$



Overall system throughput = $P_{EX}^{success} . \lambda_{EX}^{IN}$

$$T_{E2E} = T_{placement} + T_{valid} + (1-P_V^{success}).T_{E2E}$$

$$+ P_V^{success}. [T_{pending} + (1-P_{EX}^{success}).T_{E2E}]$$

Future work

1. Using resource-management semantics to resolve conflicts
   a. Not every “version conflict” will cause a transaction to fail
   b. Need to check if server has enough resources
2. Incorporating async updates from servers into staleness model
   a. Window of inconsistency/staleness does not consider updates coming from servers
3. Using real-world workloads to emulate of autonomous allocation updates
   a. Incorporate failures (server- & application- failures)
4. Model a system with multiple controllers
   a. That DON’T share state (validate) with each other
   b. Their conflicts are resolved via 2-phase-locking & 2-phase-commit at servers

Thank you!