고려대학교
KOREA UNIVERSITY

KU-The Future

European Conference on Computer Systems '20 (EuroSys)

A Fault-Tolerance Shim for Serverless Computing

2021.05.11

컴퓨터학과 김명현

Distributed and Cloud Computing Lab.

Table of Contents

Section 1

“ Introduction ”

1. Introduction

Serverless Computing

: an cloud computing architecture that does **not require developers to manage servers**

- users upload code and select trigger events (e.g., *API invocation*, *file upload*)
- than the cloud provider **transparently** manages deployment, scaling and billing.
- users are charged for resource time *used*

1. Introduction

Serverless Computing

FaaS (Function-as-a-Service)

- FaaS platforms allow user to construct applications in high-level languages
  - ex) AWS Lambda, Google Cloud Functions, Azure Function

- there is a requirement that programs be **stateless**
  - because requests are not guaranteed to be routed to any particular instance of a program

⇒ applications must **be purely functional** or **modify state in a shared storage system**

1. Introduction

Serverless Computing

Fault Tolerance in FaaS

: FaaS platform provide fault tolerance with ***retries***

- if functions fail, FaaS platforms retry it automatically
- and clients will re-issue requests after a timeout.

- retries ensure that functions are executed at-least once

- developers are recommended to write **idempotent** programs

1. Introduction

Serverless Computing

Idempotence

- idempotent functions can be applied multiple times
  without changing the result beyond the initial application
  ex) $f(f(x)) = f(x)$

- retry-based & idempotent execution would seems to guarantee exactly once execution

- but, this idempotence requirement is unreasonable for programmers
  and also insufficient to guarantee exactly once execution

1. Introduction

Serverless Computing

Idempotence

- function f : writes two keys, k and l, to storage

- if f fails between the writes of k and l,
  parallel requests might read the new version of k while reading an old version of l

  ⇒ developers are forced to explicitly reason about the correctness of read operations

1. Introduction

Serverless Computing

Solution

- a simple solution is to use serializable transactions
- functions have well-defined beginnings and endings ⇒ guarantee atomicity
- but strongly consistent transactional systems have **scaling bottleneck**s

1. Introduction

Serverless Computing

Atomic Fault-Tolerance Shim

- **AFT** : Atomic Fault Tolerance shim between FaaS platforms and a cloud storage engine

- during a single logical request, updates written to storage are buffered by AFT
- and atomically committed at the end of request

- AFT enforces the read atomic isolation guarantee,
- ensuring that transactions never see partial side effects and only read data from sets of atomic transactions

- AFT only relies on the storage engine for durability

1. Introduction

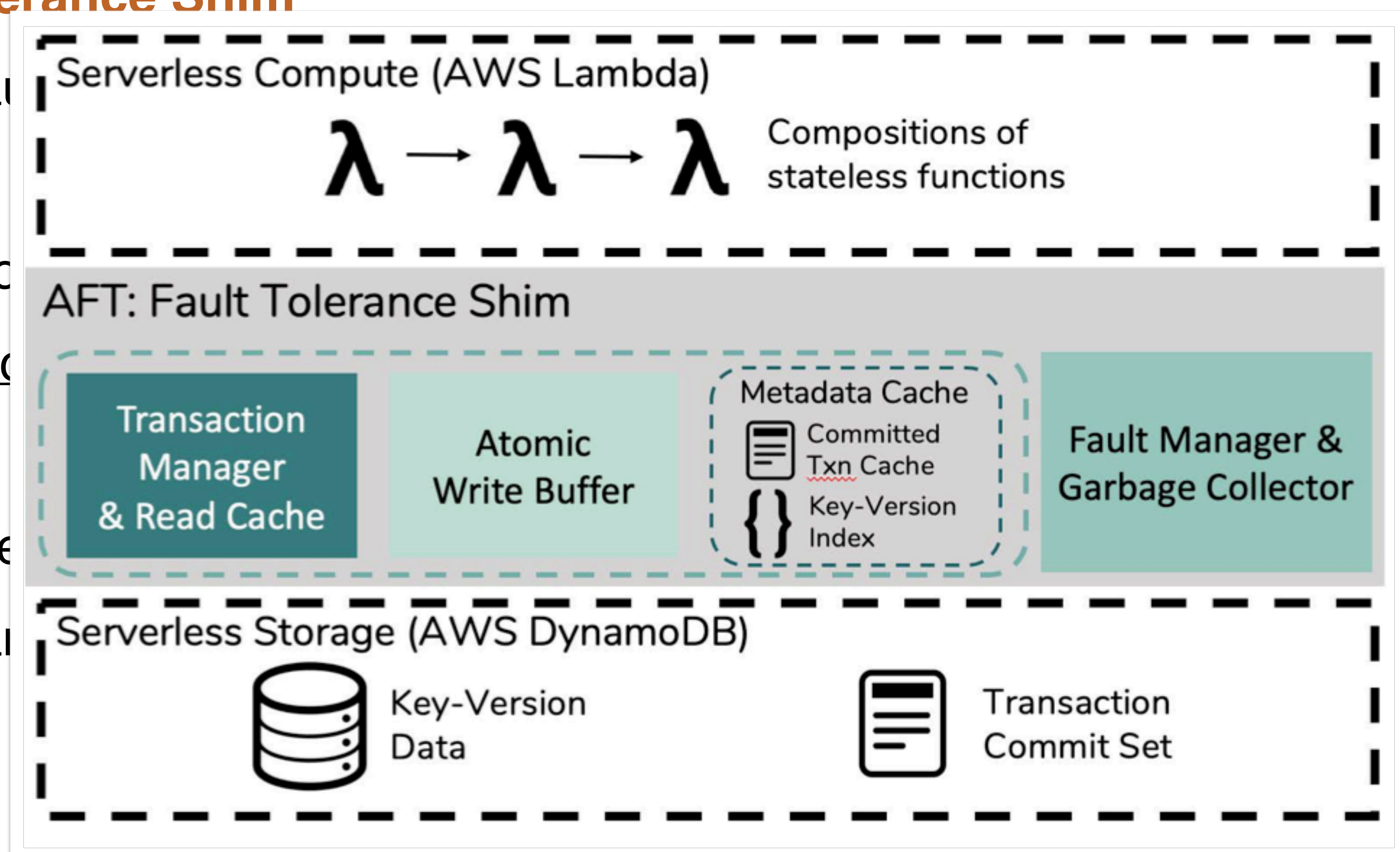
Serverless Computing

Atomic Fault-Tolerance Shim

- **AFT** : Atomic Fau e engine

- during a single lo
- and atomically co

- AFT enforces the
- ensuring that tra n sets of atomic transactions



- AFT only relies on the storage engine for durability

Section 2

“Background and Motivation”

2. Background and Motivation

Read Atomic Isolation (RA)

- RA aims to ensure that transactions do not consider partial effects of other transactions

- "A system provides RA if it prevents fractured reads anomalies
  and also prevents transactions from reading uncommitted, aborted, or intermediate data*."

Dirty Reads

Fractured Read

- "transaction T_i writes x_m and y_n, and T_j reads x_m and y_k. $(k < n)$"

* Peter Bailis, et al. (2014)

2. Background and Motivation

Challenges and Motivation

Serverless Applications

- serverless applications typically consist of multiple functions
- each requests are modeled as a linear composition of functions

- AFT must ensure atomic read/writes across all function in the composition,
  each function might be executed on a different machine
- and must ensure that retries guarantee idempotence

Section 3

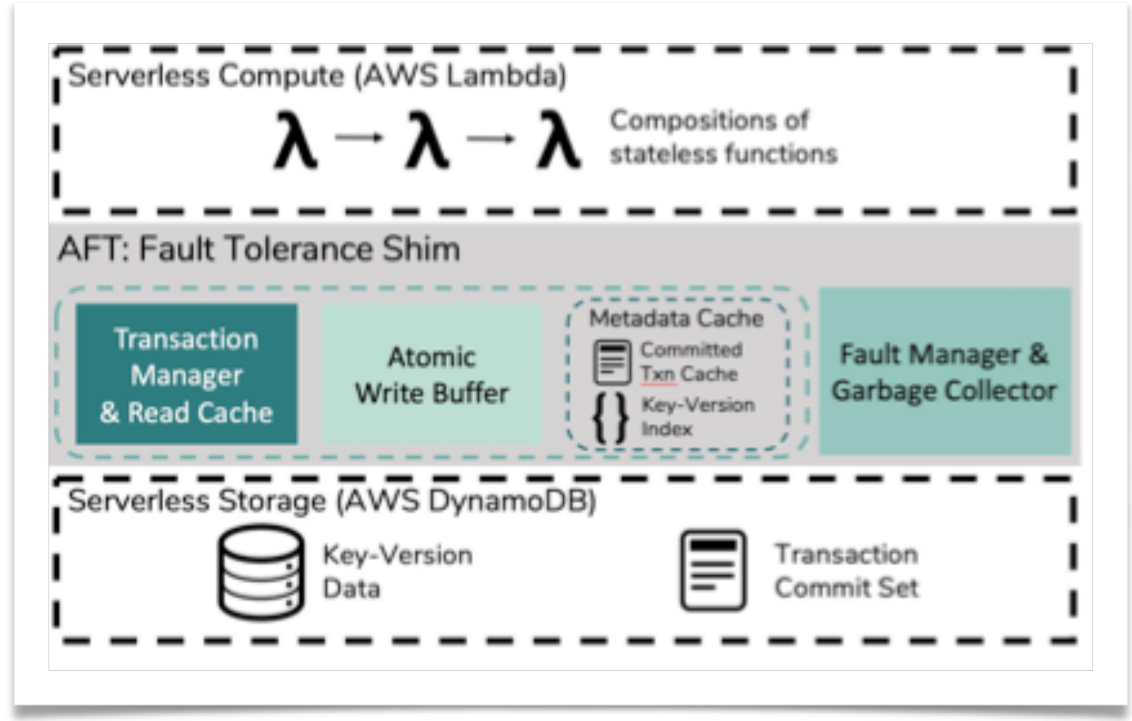
“Achieving Atomicity”

3. Achieving Atomicity

Architecture and API

Transaction : each logical request

- transaction might span multiple FaaS functions



3. Achieving Atomicity

Architecture and API

API	Description
`StartTransaction()->txid`	Begins a new transaction and returns a transaction ID.
`Get(txid, key)->value`	Retrieves key in the context of the transaction keyed by `txid`.
`Put(txid, key, value)`	Performs an update for transaction `txid`.
`AbortTransaction(txid)`	Aborts transaction `txid` and discards any updates made by it.
`CommitTransaction(txid)`	Commits transcation `txid` and persists its updates; only acknowledges after all data and metadata has been persisted.

Transaction : each logical request

- transaction might span multiple FaaS functions

- a new transaction begins when a client calls
  StartTransaction()
  - the transaction is assigned globally unique UUID
  - the ID of transaction ($txid$) consists of $< timestamp, uuid >$ pair

- AFT uses $txid$ to ensure that its updates are only persisted once
  assuring idempotence in the face of retries

3. Achieving Atomicity

Architecture and API

API	Description
`StartTransaction()->txid`	Begins a new transaction and returns a transaction ID.
`Get(txid, key)->value`	Retrieves key in the context of the transaction keyed by `txid`.
`Put(txid, key, value)`	Performs an update for transaction `txid`.
`AbortTransaction(txid)`	Aborts transaction `txid` and discards any updates made by it.
`CommitTransaction(txid)`	Commits transcation `txid` and persists its updates; only acknowledges after all data and metadata has been persisted.

- each transaction sends all operations
  to a single AFT node

- within the bounds of a transaction,
  clients interact with AFT by calling *Get()*, *Put()*

- when a client calls *CommitTransaction()*,
  AFT assigns a commit timestamp, and persists all of the transaction's updates

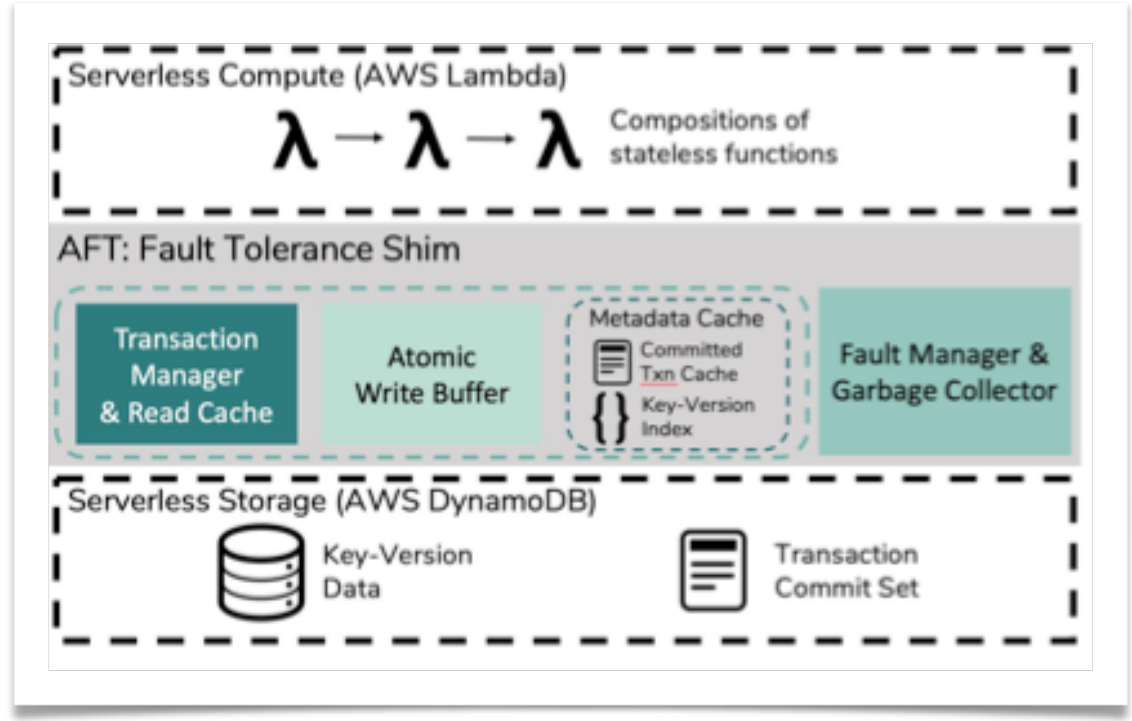
- if a client calls *AbortTransaction()* anytime, its all updates are aborted, and the data is deleted

3. Achieving Atomicity

Architecture and API

AFT node

- consists of *transaction manager*, *atomic write buffer*, *local metadata cache*



Transaction Manager

- tracks key versions each transaction has read, and enforces the read atomicity guarantee

Atomic Write Buffer

- gathers each transaction's writes, and persists them atomically at commit time

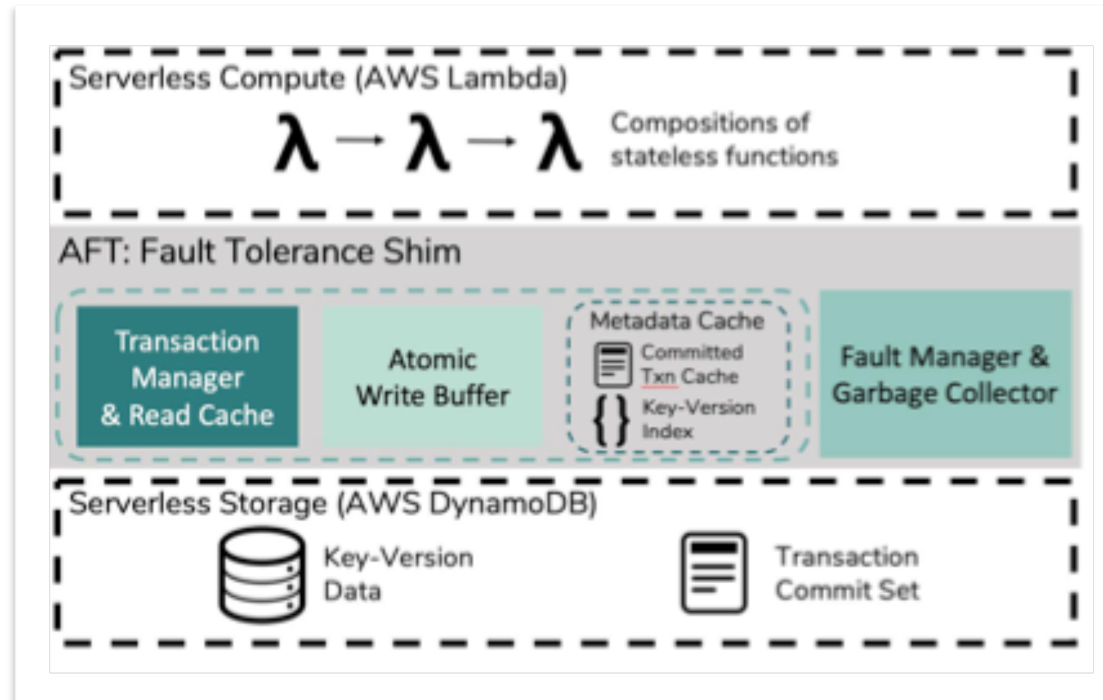
AFT maintains a *Transaction Commit Set Storage*

3. Achieving Atomicity

Architecture and API

- AFT caches recently committed transactions' ID, and locally maintains indexes of them

- also, AFT has a *data cache*, which stores values for the key versions in metadata cache
- *data cache* improves performance by avoiding storage lookups for frequently accessed versions



3. Achieving Atomicity

Definitions

i : transaction T_i 's ID ($< timestamp, uuid >$)

- T_i is newer than T_j ($T_i > T_j$) if $i > j$

k_i : version of key k written by transaction T_i

k : any version of that key k

- key versions are hidden from users
- clients just requests reads and writes of keys
- AFT automatically determines which versions are compatible with each request

T_i.*writeset* : set of key versions written by transaction T_i

k_i.*cowritten* : cowritten set of key versions (T_i.*writeset* == k_i.*cowritten*)

3. Achieving Atomicity

Definitions

Read Atomic Isolation requires preventing ***dirty read***s and ***fractured read***s

Preventing Dirty Read

- if transaction T_i reads key version k_j, transaction T_j must have successfully committed

Preventing Fractured Read

- each transaction's read set must form an ***Atomic Readset***

3. Achieving Atomicity

Definitions

Definition 1. Atomic Readset

- let R be a set of key versions
- R is an *Atomic Readset* if $\forall k_i \in R, \;\; \forall l_i \in k_i.cowritten, \;\; l_j \in R \quad \Rightarrow \quad j \geq i$

Example)

- there are two committed transactions in storage:

  $T_1 : \{l_1\}$ and $T_2 : \{k_2, l_2\}$

- if a new transaction T_n first requests k and reads k_2,

  a subsequent read of l must return <u>l_2 or some newer version of l</u>

3. Achieving Atomicity

Definitions

Read-Your-Writes

- transaction must read the most recent version of a key it previously wrote

Example)

- if a transaction T_i wrote key version k_{i_1}, following read should return k_{i1}
- then if T_i writes k_{i_2}, future reads will return k_{i_2}

Repeatable Read

- transaction should view the same key version if it requests the same key repeatedly

Example)

- if T_i reads version k_j then later requests a read of k again, it should read k_j unless it wrote k_i

3. Achieving Atomicity

Preventing Dirty Reads

AFT's write-ordering protocol

- Atomic Write Buffer sequesters all updates for each transaction

- when *CommitTransaction()* is called, AFT first writes transaction's updates to storage

- after all updates have successfully been persisted,

  AFT writes the transaction's write set, timestamp, and UUID to the Transaction Commit Set

- only after the Commit Set updated, AFT make the transaction's data visible to other requests

- if *AbortTransaction()* is called, updates are simply deleted from Atomic Write Buffer

3. Achieving Atomicity

Preventing Dirty Reads

AFT's write-ordering protocol

- AFT doesn't overwrite keys in place (each key version is mapped to a **unique** storage key)

  ⇒ this increases AFT's storage and metadata footprints

3. Achieving Atomicity

Preventing Dirty Reads - Atomic Fault Tolerance

AFT's write-ordering protocol

- when an application function fails, none of its updates will be persisted
- and the transaction will be aborted after a timeout

- if the function retries, it can use the same transaction ID to continue the transaction

3. Achieving Atomicity

Preventing Fractured Reads

AFT's atomic read protocol

- after every consecutive read, the set of key versions read forms an Atomic Readset (Definition 1)

- algorithm 1 ensures that reads are only issued from already-committed transaction
  by using the local cache of committed transaction metadata and recent key versions

3. Achieving Atomicity

Preventing Fractured Reads

AFT's atomic read protocol - Algorithm 1

line 3-5

- the lower bound of *target* is computed by selecting the largest transaction ID in R that modified k

Algorithm 1 AtomicRead: For a key k, return a key version k_j such that the read set R combined with k_j does not violate Definition 1.

Input: $k, R, WriteBuffer, storage, KeyVersionIndex$

```
1:  lower := 0 // Transaction ID lower bound.
2:  // Lines 3-5 check case (1) of the inductive proof.
3:  for l_i ∈ R do
4:      if k ∈ l_i.cowritten then // We must read k_j such that j ≥ i.
5:          lower = max(lower, i)
6:  // Get the latest version of k that we are aware of.
7:  latest := max(KeyVersionIndex[k])
8:  if latest == None ∧ lower == 0 then
9:      return NULL
10: target := None // The version of k we want to read.
11: candidateVersions := sort(filter(KeyVersionIndex[k], kv.tid ≥
    lower)) // Get all versions of k at least as new as lower.
12: // Loop through versions of k in reverse timestamp order —
    lines 13-23 check case (2) of the inductive proof.
13: for t ∈ candidateVersions.reverse() do
14:     valid := True
15:     for l_i ∈ k_t.cowritten do
16:         if l_j ∈ R ∧ j < t then
17:             valid := False
18:             break
19:     if valid then
20:         target = t // The most recent valid version.
21:         break
22: if target == None then
23:     return NULL
24: R_new := R ∪ {k_target}
25: return storage.get(k_target), R_new
```

3. Achieving Atomicity

Preventing Fractured Reads

AFT's atomic read protocol - Algorithm 1

line 13-23

- iterates all cowritten keys of candidate versions

- if any cowritten key's version is in R,

  AFT declares the candidate version valid

  if only if the cowritten key's version is not newer

  than the version in R

Algorithm 1 AtomicRead: For a key k, return a key version k_j such that the read set R combined with k_j does not violate Definition 1.

Input: $k, R, WriteBuffer, storage, KeyVersionIndex$

1: $lower := 0$ // Transaction ID lower bound.
2: // Lines 3-5 check case (1) of the inductive proof.
3: **for** $l_i \in R$ **do**
4:     **if** $k \in l_i.cowritten$ **then** // We must read k_j such that $j \geq i$.
5:         $lower = max(lower, i)$
6: // Get the latest version of k that we are aware of.
7: $latest := max(KeyVersionIndex[k])$
8: **if** $latest == None \land lower == 0$ **then**
9:     return NULL
10: $target := None$ // The version of k we want to read.
11: $candidateVersions := sort(filter(KeyVersionIndex[k], kv.tid \geq lower))$ // Get all versions of k at least as new as $lower$.
12: // Loop through versions of k in reverse timestamp order — lines 13-23 check case (2) of the inductive proof.
13: **for** $t \in candidateVersions.reverse()$ **do**
14:     $valid := True$
15:     **for** $l_i \in k_t.cowritten$ **do**
16:         **if** $l_j \in R \land j < t$ **then**
17:             $valid := False$
18:             break
19:     **if** $valid$ **then**
20:         $target = t$ // The most recent valid version.
21:         break
22: **if** $target == None$ **then**
23:     return NULL
24: $R_{new} := R \cup \{k_{target}\}$
25: return $storage.get(k_{target}), R_{new}$

3. Achieving Atomicity

Other Guarantees

Read-Your-Writes

- when a transaction requests a key currently stored in its own write set,

    (Atomic Write Buffer does not yet assign a commit timestamp)

    AFT simply return the data immediately

Section 4

“ Scaling AFT ”

4. Scaling AFT

Introduction

- each machine has a background thread:
    - periodically gathers all transactions committed recently on this node
    - broadcasts them to all other nodes
    - listens for messages from other replicas

    - when the thread receives a new commit set,
      it adds all transactions to its local Commit Set Cache and updates its key version index

- in a distributed environment, the cost of communicating this metadata can be extremely high

4. Scaling AFT

Pruning Commit Sets

- AFT prunes the set of transactions that each node multicasts

- **locally superseded transaction**s do not need to be broadcast
  - a transaction T_i is locally superseded if $\forall k_i \in T_i.writeset, \exists k_j \mid j > i$

4. Scaling AFT

Pruning Commit Sets

Algorithm 2 : *IsTransactionSuperseded*

- check whether T_i has been superseded

- each node's background multicast protocol checks before sending it to other replicas

Algorithm 2 IsTransactionSuperseded: Check whether transaction T_i has been superseded—if there is a newer version of every key version written by T_i.

Input: $T_i, keyVersionIndex$

```
for k_i ∈ T_i.writeset do
    latest := k.latest_tid()
    if latest == i then
        return False
return True
```

- the receiving node checks whether the transaction has been superseded
  - if it is true, AFT do not merge the transaction into the metadata cache

- once a transaction is superseded, the node can safely delete the transaction metadata

4. Scaling AFT

Fault Tolerance

AFT considers both **safety** and **liveness**

guaranteeing safety

- AFT relies on *write-ordering protocol*
- ensures that each node does not persist dirty data and commits are correctly acknowledged

4. Scaling AFT

Fault Tolerance

AFT considers both **safety** and **liveness**

guaranteeing liveness

- if a replica commits a transaction, and fails before broadcasting the commit, (situation α)

  AFT must ensure that other replicas are still made aware of the committed data

- distributed deployments of AFT have a **fault manager**
- fault manager periodically scans the **Transaction Commit Set** in storage,

  and checks for commit records that it has not received via broadcast

- in situation α, fault manager simply read the transaction's commit record again

4. Scaling AFT

Deployment and Autoscaling

Deploying AFT

- AFT is deployed using **Kubernetes**
- fault manager run in separate Docker containers and on separate machines
- fault manager detects failed nodes and configuring their replacements

Section 5

"Garbage Collection"

5. Garbage Collection

Introduction

these data sources cause **overheads**:

- the list of all transactions committed by AFT
- the set of key versions

    (AFT does not overwrite key versions)

5. Garbage Collection

Local Metadata Garbage Collection

- locally garbage collect transaction metadata
- background GC process periodically sweeps all committed transactions in the metadata cache

- conditions of being garbage collected:
  - if the transaction is **superseded** (Algorithm 2)
  - if no currently-executing transactions have read from the transaction's write set

- GC remove that transaction from the Commit Set Cache

5. Garbage Collection

Global Data Garbage Collection

- **the fault manager** also serves as a global garbage collector
  - fault manager receives commit broadcasts

- global GC executes Algorithm 2 to determine which transaction is superseded
  - generates a list of transactions considered superseded
  - asks all nodes if they have locally deleted those transactions

  - when the GC process receives acknowledgements from all nodes,
    it deletes the transaction's write and commit metadata

5. Garbage Collection

Global Data Garbage Collection - Limitation

- if a running transaction T_j has read from T_i's write set, GC protocol will not delete T_i
- but we do not know each running transaction's full read set

- example)
  - $T_1 : \{k_1\},\ T_2 : \{l_2\},\ T_3 : \{k_3, l_3\}$
  - a transaction T_r first reads k_1 then requests key l

  - GC process will not delete T_1, because T_r is reading T_1 and is not committed yet
  - GC process might delete T_2
  - if T_r attempts to read l, it will find no valid version since l_c is invalid

Section 6

“ Evaluation ”

6. Evaluation

Environment

- key design goal for AFT is to run on a **variety of cloud storage backend**s
- AFT is implemented over *AWS S3** and *AWS DynamoDB***
- AFT is run over *Redis* (deployed via AWS ElastiCache)

- AFT is implemented in Go & Python, and runs on top of Kubernetes
- simple stateless load balancer routes request to AFT nodes in a **round-robin** fashion

- all experiments were run in the us-east-1a AWS availability zone
  - AFT node ran on a *c5.2xlarge* EC2 instance with **8 vCPUs** (4 physical cores) and **16GB RAM**

* AWS S3 (Simple Storage Service) : large scale object storage system
** AWS DynamoDB : cloud-native key-value store

6. Evaluation

AFT Overheads - I/O Latency

- the experiment compares the cost of writing data **directly to DynamoDB**,
  and to the cost of the same set of writes **using AFT's commit protocol**

- to isolate the overheads from FaaS platforms, we issue writes from a single thread in VM

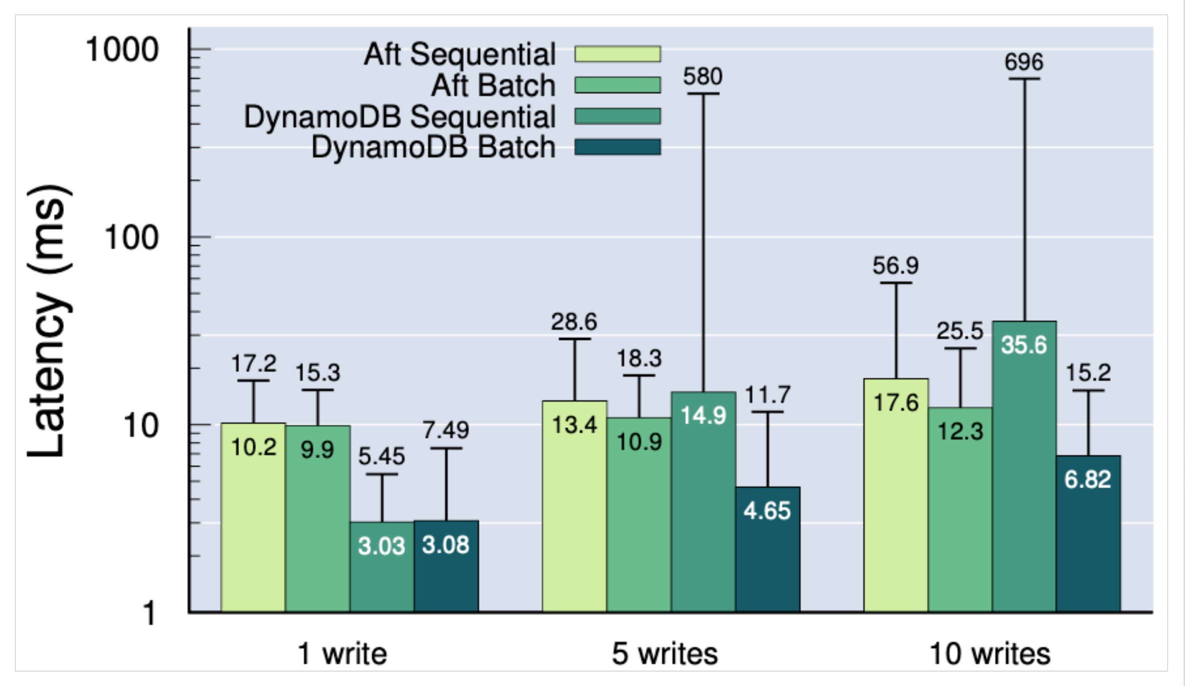
- two baselines write directly to DynamoDB,
  one with **sequential write**s and the other with **batching**

- AFT uses batched writes by default
- but the experiment measures two configurations over AFT:
  sequential writes and single batched write

6. Evaluation

AFT Overheads - I/O Latency

- the latency of DynamoDB Sequential writes increases **linearly** with the number of writes



- AFT Sequential's performance scales better than DynamoDB Sequential's

- overheads:
  - the cost of the extra network overhead by shipping data to AFT
  - the cost of writing the extra commit record

6. Evaluation

AFT Overheads - End-to-End Latency

- the experiment compares the latency of executing transaction on **AWS Lambda with AFT**, and **without AFT** interposed between the compute and storage layers

- evaluates three storage systems: *AWS S3*, *AWS DynamoDB*, *Redis*

- each transaction is composed of 2 functions, each functions consists of 1 read and 2 writes (each reads/writes are of 4KB objects)
  - these small transactions reflects real-world CRUD applications

6. Evaluation

AFT Overheads - End-to-End Latency

Consistency Anomalies

- Read Atomic Consistency Guarantee is AFT's key advantage over S3, DynamoDB and Redis

- the experiment* measure *Read-Your-Write* anomalies (RYW) and *Fractured Read* (FR)
    - RYW : occurs when a transaction writes a key version and does not later read the same data
    - FR : occurs when a later transaction read the old data

Storage Engine	Consistency Level	RYW Anomalies	FR Anomalies
AFT	Read Atomic	0	0
S3	None	595	836
DynamoDB	None	537	779
DynamoDB	Serializable	0	115
Redis	Shard Linearizable	215	383

* 10000 transactions

6. Evaluation

AFT Overheads - End-to-End La

Storage Engine	Consistency Level	RYW Anomalies	FR Anomalies
AFT	Read Atomic	0	0
S3	None	595	836
DynamoDB	None	537	779
DynamoDB	Serializable	0	115
Redis	Shard Linearizable	215	383

Consistency Anomalies

Redis

- each Redis shard is linearizable but no guarantees are made across shards
- Redis shows anomalies on 6% of requests

DynamoDB with transaction mode

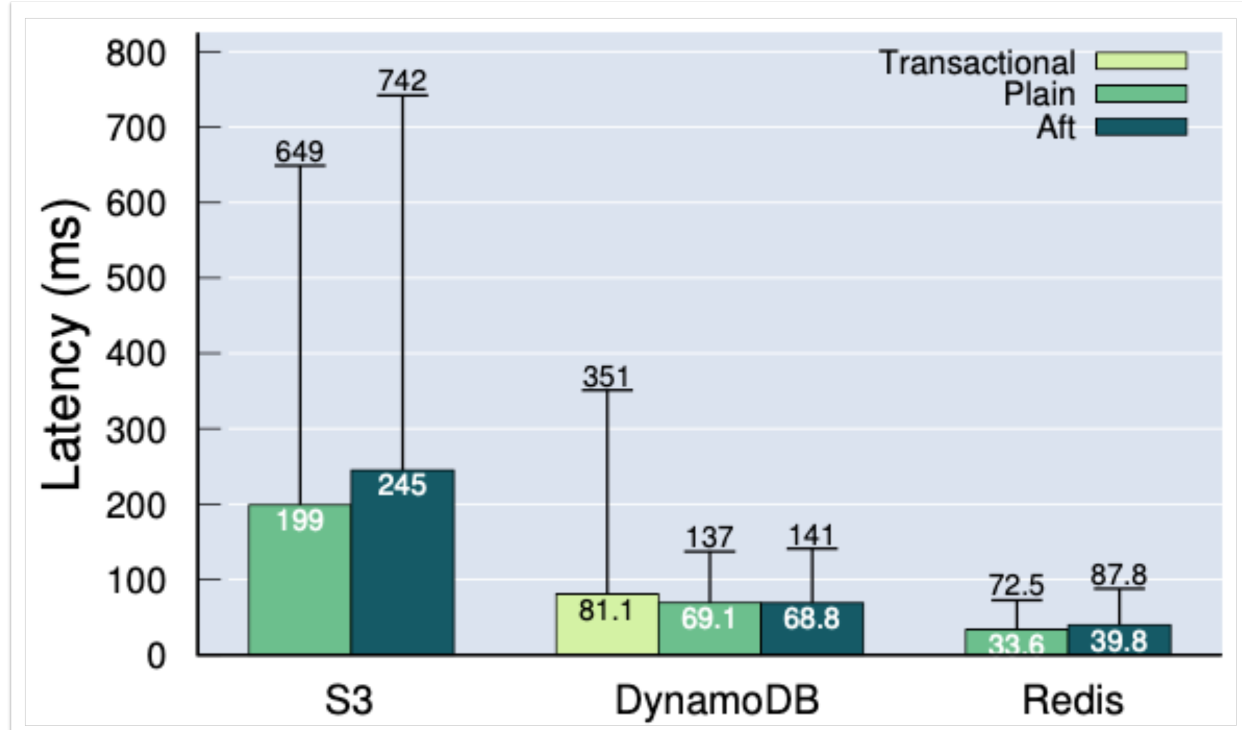
- this supports transaction that are either read-only or write-only
- all operations in a transaction either succeed or fail as a group
- but, there is no atomicity guarantee for transactions that span multiple functions

6. Evaluation

AFT Overheads - End-to-End Latency

Performance

- AFT with S3 is 25% slower than plain S3
- S3 is a throughput-oriented object store
- each transaction is composed of 2 functions, which consists of 1 read and 2 writes



Summary

- AFT offers performance that is competitive with cloud storage engines and eliminates a significant number of anomalies

6. Evaluation

Read Caching & Data Skew

- AFT's data caching, its performance effect,

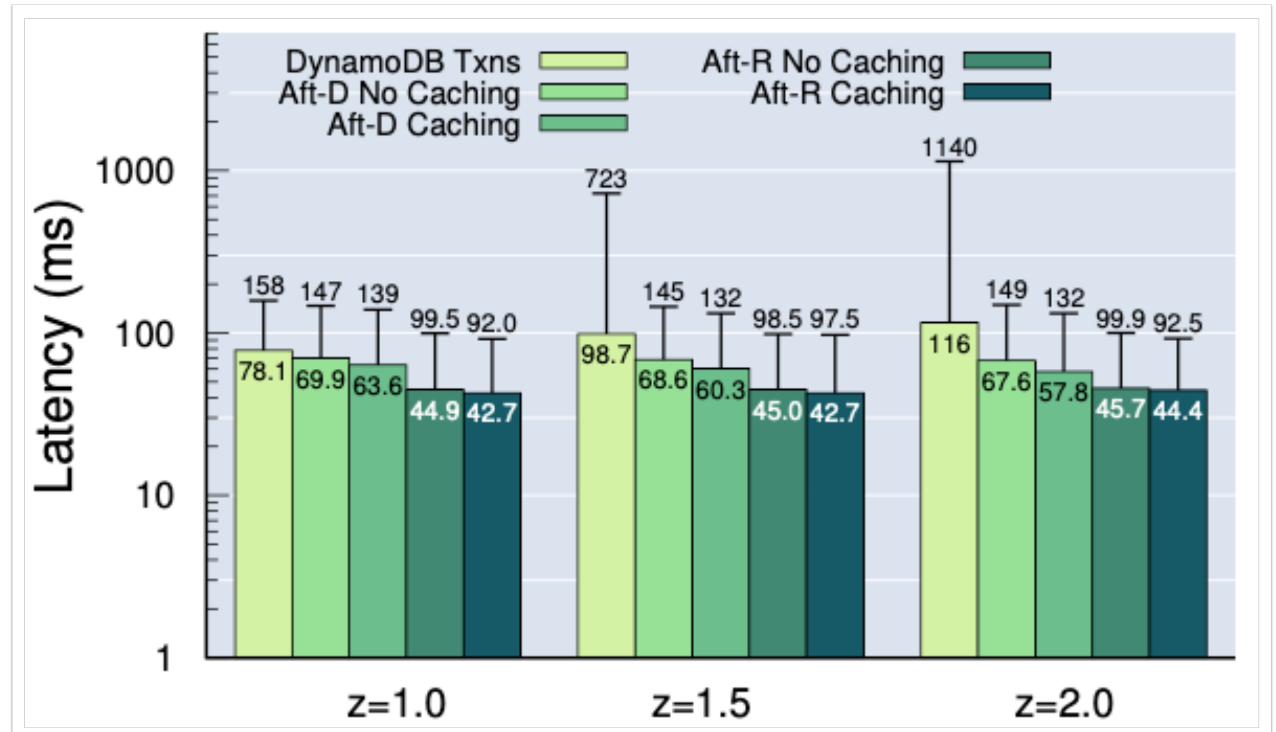
  its interaction with skewness

- the experiment uses same workload

  2-function transaction with 2R & 1W

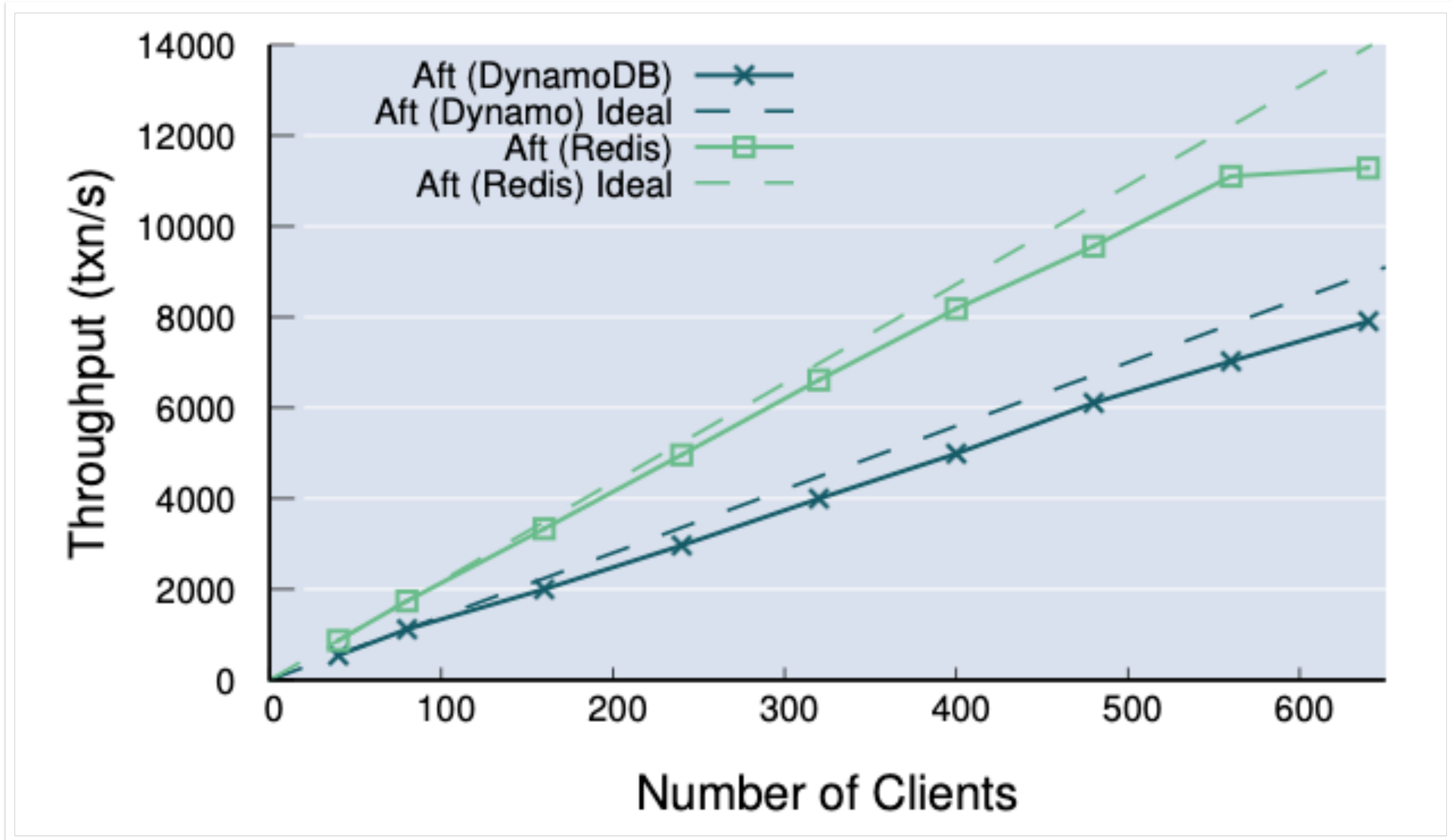
- *AFT-D* : DynamoDB with AFT
- *AFT-R* : Redis with AFT

- z : Zipfian Distributions:
  - 1.0 (lightly contended), 1.5 (moderately contended), 2.0 (heavily contended)



6. Evaluation

Scalability



Summary

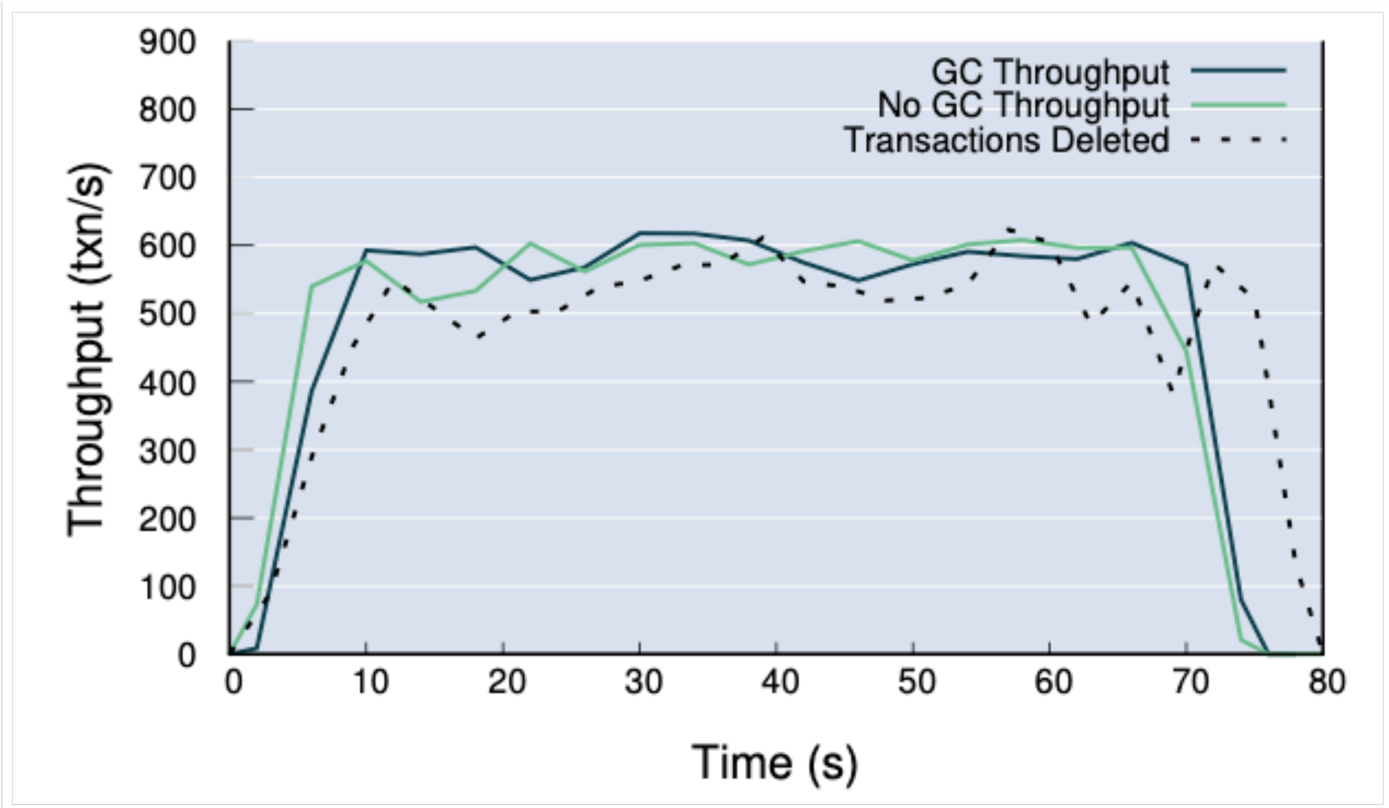
- AFT is able to efficiency scale to thousands of transactions per second and hundreds of parallel clients within 90% of ideal throughput

6. Evaluation

Garbage Collection Overheads

- run a single AFT node with 40 clients and measure throughput with GC enabled and disabled

- the cost of this GC process is that we require **separate cores** (4 AFT cores, 1 GC core)



- however, the resources allocated to GC are much smaller than the resources required to run the system

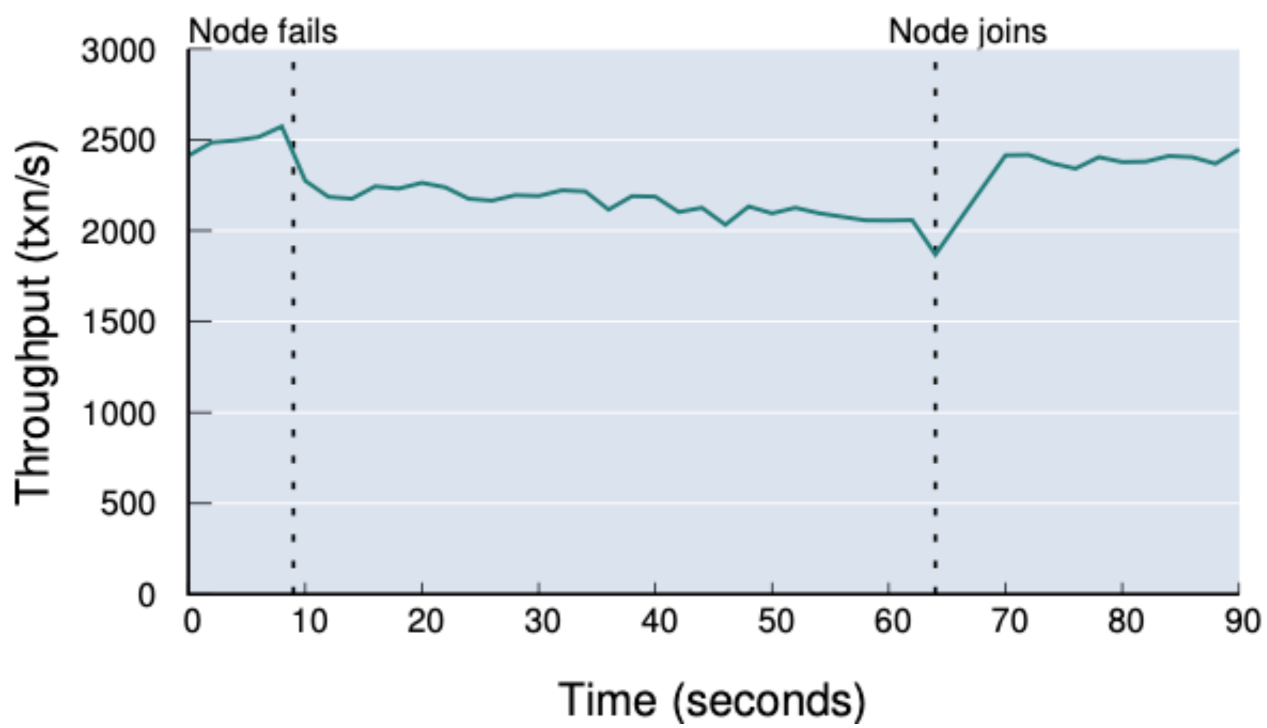
6. Evaluation

Fault Tolerance

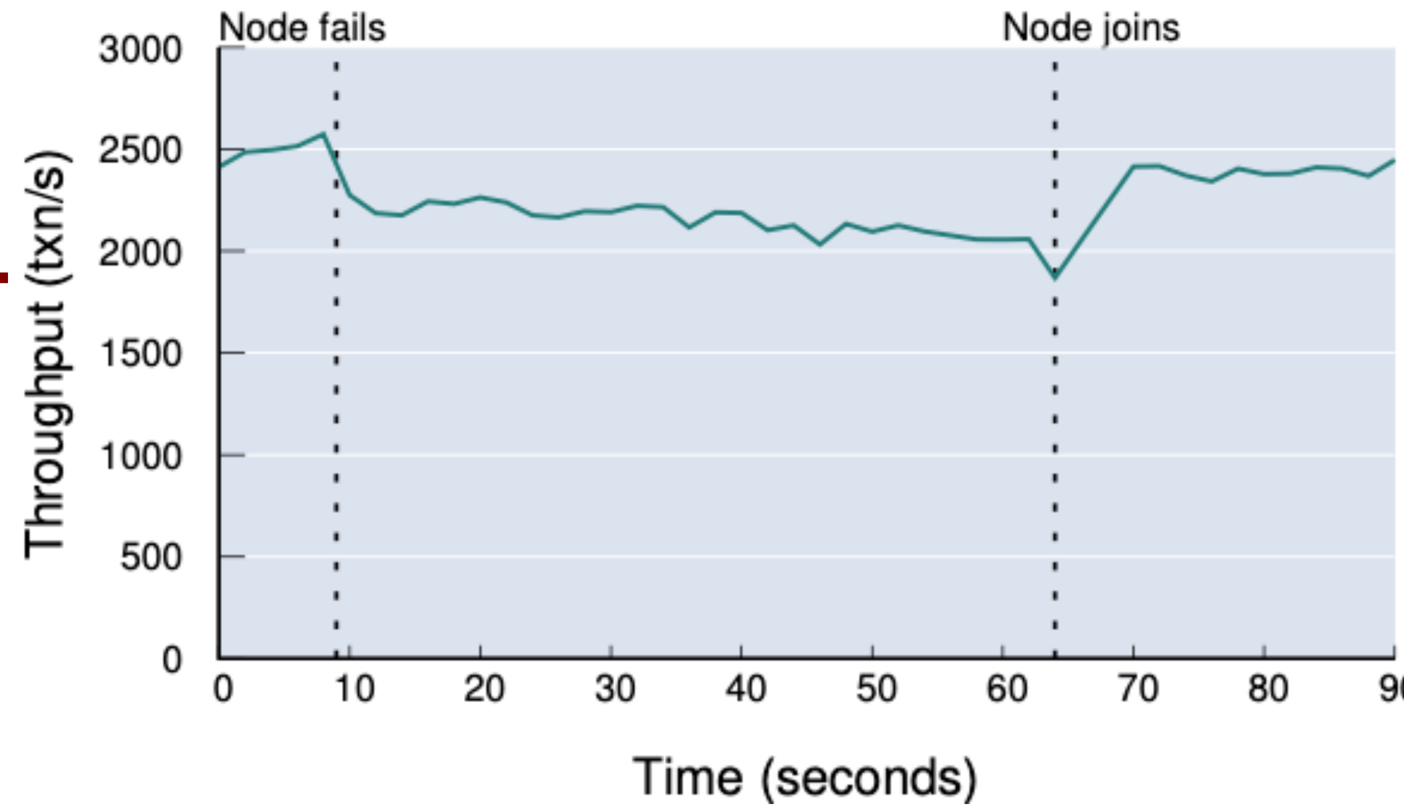
- this experiment measures the <u>effect of a node failure</u> and <u>cost of recovering from failure</u>

  (how long it takes for a node to cold start and join the system)

- we run AFT with 4 nodes and 200 parallel clients, and terminate a node just before 10 sec.



6. Evaluation



Fault Tolerance

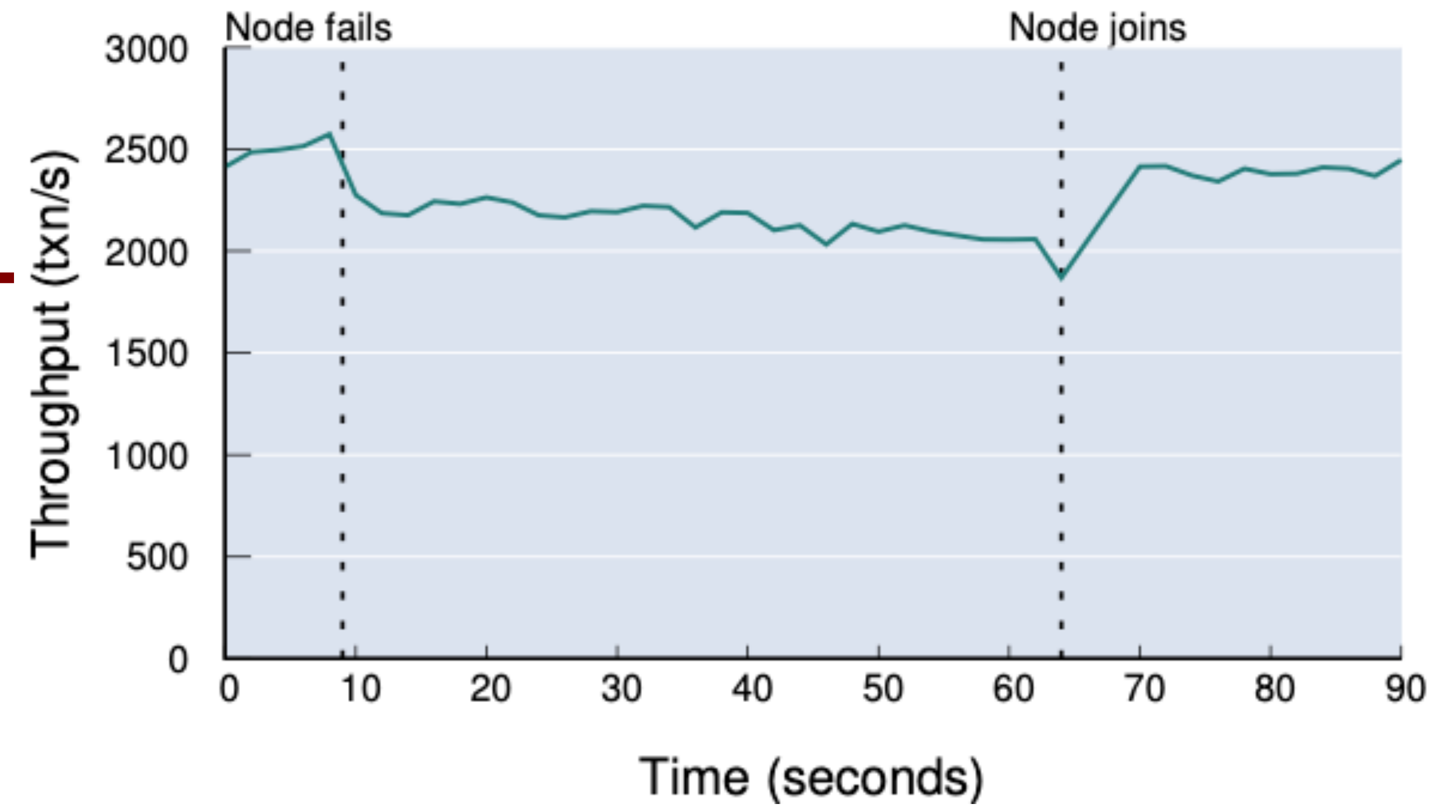
- throughput immediately drops about 16%

- the AFT management process determines
  that a node has failed within 5 seconds
- and it adds assigns a new node to join the cluster

- the system <u>pre-allocate **standby nodes**</u> to avoid having to wait for new VMs to start
  (starting new VM can take up to 3 minutes)

6. Evaluation

Fault Tolerance



- over the next 45 seconds, the new node downloads the AFT docker container, updates its local metadata cache

- after 60 seconds, the node joins the cluster successfully

- the overheads from starting a new node can be mitigated by pre-downloading containers,
  - and by maintaining standby nodes with warm metadata cache

Section 7

“Related Work”

7. Related Work

Atomic Reads

Transactional Casual Consistency (TCC)*

- guarantees atomic reads
- guarantees that reads and writes respect *Lamport's Happened-Before Relation***

- but TCC model relies on fixed node at the storage layer
- also, TCC model achieve read atomicity at the storage layer

* Syed Akbar Mehdi, *et al*. (2017)
** Leslie Lamport. (1998)

7. Related Work

Fault Tolerance

- there is a rich literature on fault-tolerance for distributed systems:
    - **checkpoint / restart** of containers or VMs*
    - **log based replay****

- AFT is based on the simple retry model used by existing serverless platforms
- and AFT ensures atomicity and idempotence to achieve exact-once semantics

* Wubin Li, Ali Kanso. (2015)
** Stephanie Wang, *et al*. (2019)

Section 8

“Conclusion and Future Work”

8. Conclusion and Future Work

Conclusion

AFT (Atomic Fault Tolerance) Shim

- a low-overhead fault tolerance shim for serverless computing
- AFT interposes between commodity FaaS platforms and key-value stores
- AFT achieves fault-tolerance by guaranteeing Read Atomic Isolation
- AFT guarantees exactly-once execution in the face of failures

8. Conclusion and Future Work

Future Work

Efficient Data Layout

- AFT's key-per-version data layout works well over *DynamoDB* and *Redis*,
  but performs poorly over *S3*
- in settings with high-volume updates, the authors would like to optimize data layouts for *S3*

Auto Scaling Policies

- policies for efficient and accurate autoscaling decisions

Data and Cache Partitioning

- AFT's caching scheme is very naïve approach
- AFT will result in every node caching largely the same data, particularly for skewed workloads

Q & A

End of Document