北京大学高能效计算与应用中心
Center for Energy-efficient Computing and Applications

m2Clock: Handling IO Performance for Shared Multi-Tenant Cloud Storage

Tong Meng, Xiaoyang Wang, Guangyu Sun

2019.05.09

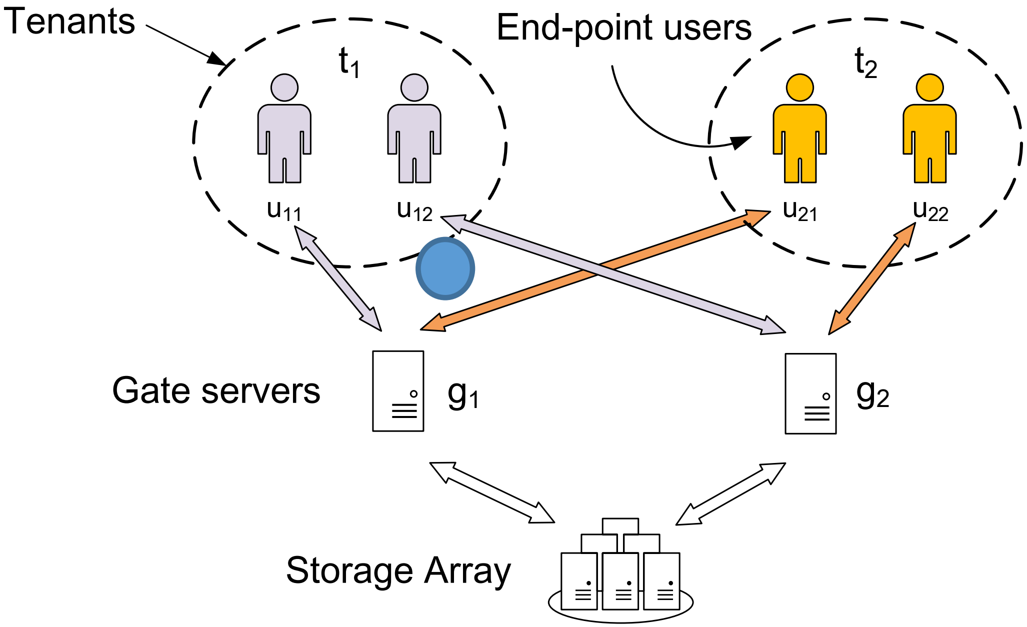
CECA, Peking University

Outline

- □ Introduction
  - Shared multi-tenant cloud storage
  - Classic approaches to solve similar problems
- □ m2Clock methods
- □ Evaluation results
- □ Conclusion

Basics: Shared multi-tenant cloud storage

- **Vendor**: the storage service provider.
  - **Gate servers**: special nodes that **schedule** and keep track of the execution of I/O requests from each tenant.
  - **Storage array**: a large cluster of nodes to provide storage service.
- **Tenant**: the **basic unit** to allocate resources.
  - **User**: each tenant consists of **multiple** standalone end-point users.



Scheduling targets

- □ Quality of Service (QoS)
  - – Predictable IOPS
    - Reservation and limit
  - – Lower latency
- □ Scalability
  - – The ability for the system to serve more tenants.
- □ Scheduling target
  - – Minimizing the latency while bound the IOPS for each tenant between a minimum reservation and a maximum limit.

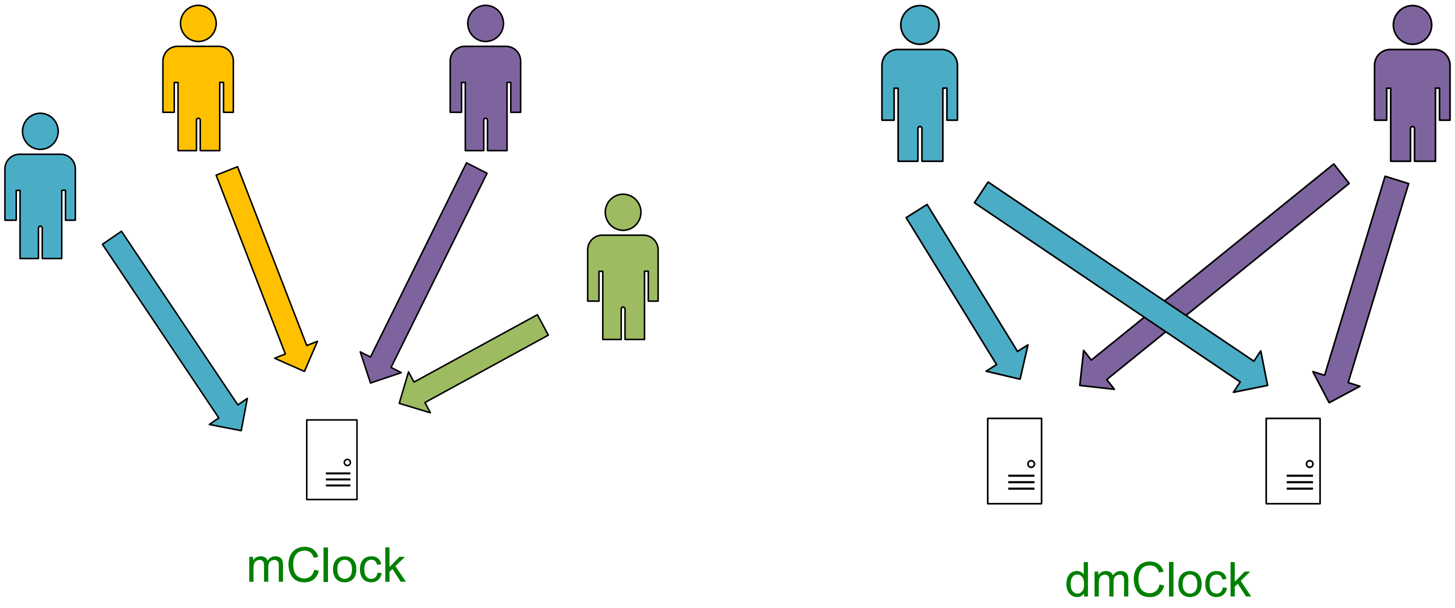
Outline

- □ Introduction
  - Shared multi-tenant cloud storage
  - Classic approaches to solve similar problems
- □ m2Clock methods
- □ Evaluation results
- □ Conclusion

mClock and dmClock methods

- I/O resource allocation for virtual machines
  - **Proportional**-share fairness subject to minimum **reservations** and maximum **limit**s on the IO allocations for VMs.
  - Per-VM parameters: Reservations, Limits and Proportion

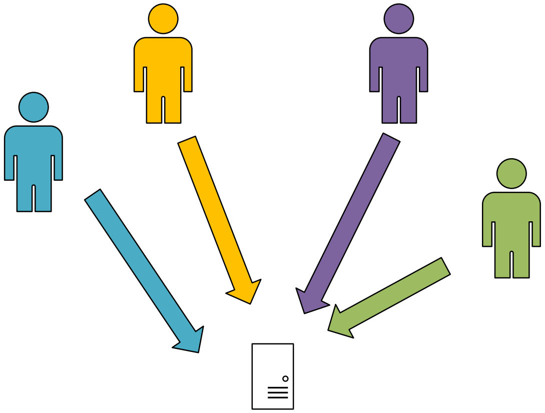
- mClock and dmClock



mClock Method

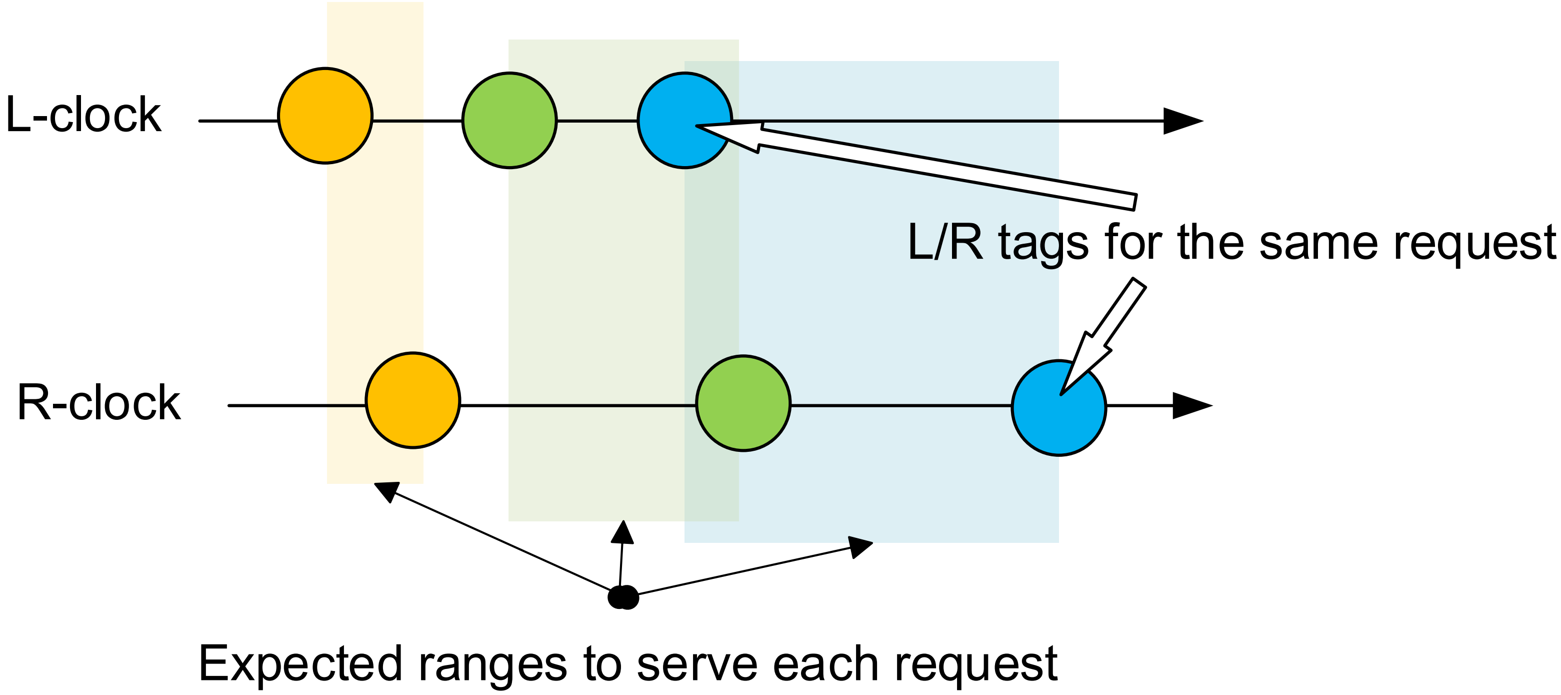
- mClock uses two main ideas:
  - Multiple real-time clocks
    - Reservation-based R-clock, Limit-based L-clock, and Proportion-based clocks
  - Dynamic clock selection
    - Dynamically select one from multiple real-time clocks for scheduling.

- Tag assignment for *i*-th request from the VM *v*
  - Reservation Tag $R_i^v = \max\{R_{i-1}^v + \frac{1}{r}, t\}$
  - Limit Tag $L_i^v = \max\{L_{i-1}^v + \frac{1}{l}, t\}$
  - Proportion Tag P_i^v



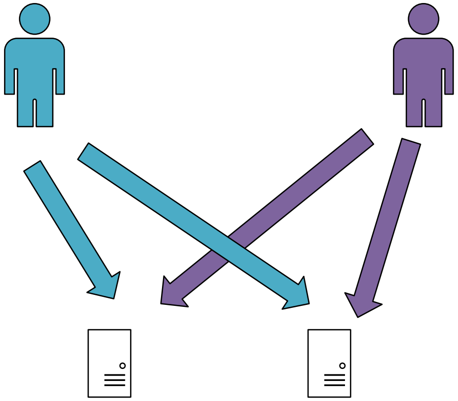
Basic idea behind mClock

- A request is expected to be served in $L_i^r \sim R_i^r$


L-clock
R-clock
L/R tags for the same request
Expected ranges to serve each request

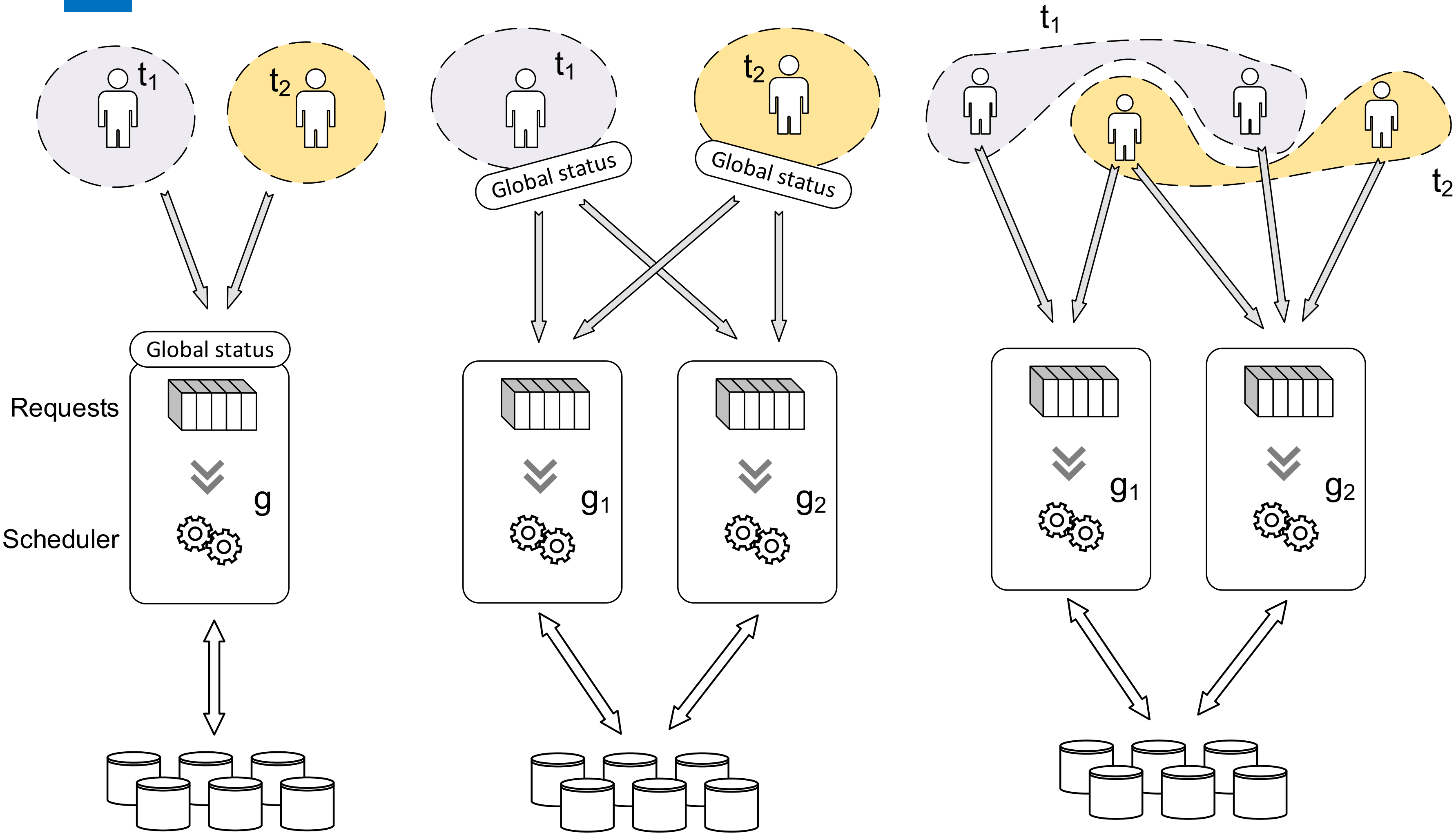
dmClock: Distributed mClock

- dmClock runs a modified version of *mClock*
  - It piggybacks two integers ρ_v and δ_v with each request of VM *v* to a storage server *s*.
    - ρ_v : the number of IO requests from *v* that have been served as reservation-based between the previous request to s and the current request.
    - δ_v : the number of IO requests from *v* that have completed service at all the servers between the previous request (from v) to the server s and the current request.



- Modified tags
  - $R_i^v = \max\{R_{i-1}^v + \frac{\rho_v}{r}, t\}$
  - $L_i^v = \max\{L_{i-1}^v + \frac{\delta_v}{l}, t\}$

Multiple-tenant cloud storage systems



mClock: A single gate serves multiple tenants

dmClock: Multiple gates serve multiple tenants

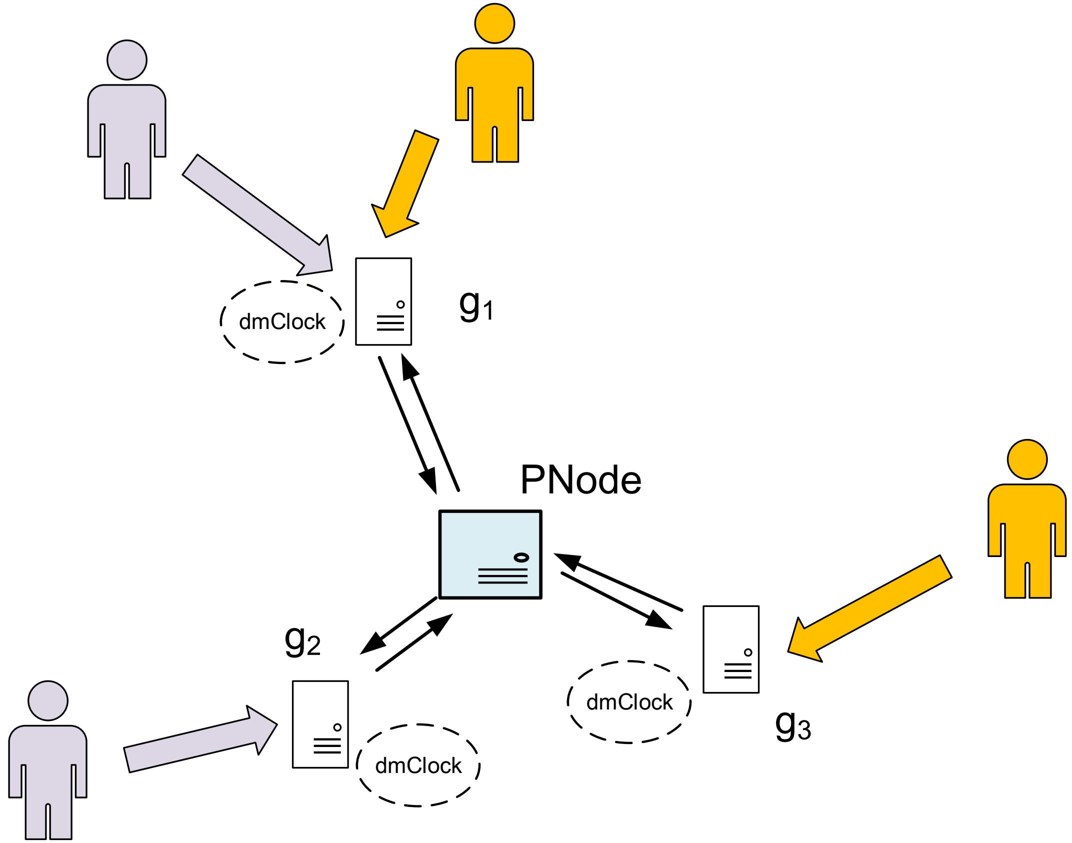
Each tenant consists of several individual users

Outline

- Introduction
- m2Clock methods
  - Version 1: Centralized dmClock
  - Version 2: Updating in batch
  - Version 3: Local adjustment
  - Version 4: Burst broadcast
- Evaluation results
- Conclusion

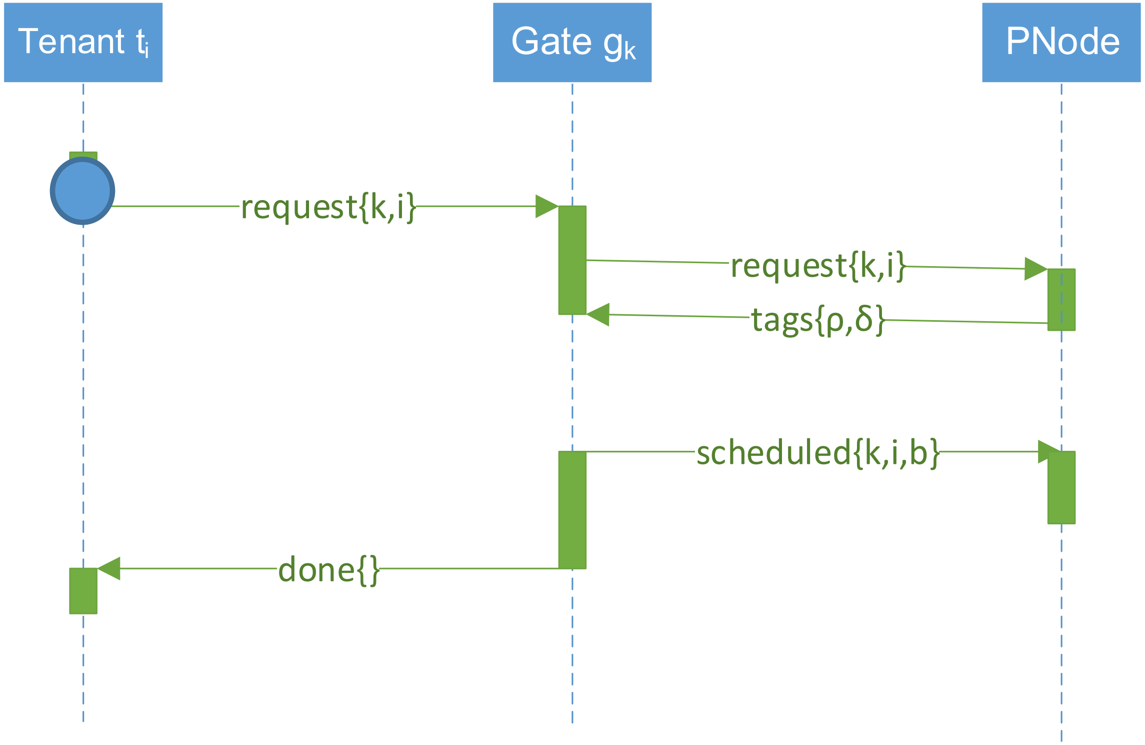
m2Clock architecture

- A centralized component called **PNode** is used to maintain counting information for δ_v and ρ_v used in the dmClock method.



m2Clock v1: Centralized dmClock

- □ Request arriving
  - – The gate forwards the message to PNode to get ρ and δ
- □ Request scheduled
  - – The gate should inform PNode about it



m2Clock v1: disadvantages

□ Heavy workload for PNode

- PNode has to react twice on average for **every** request

□ Long latency before scheduling

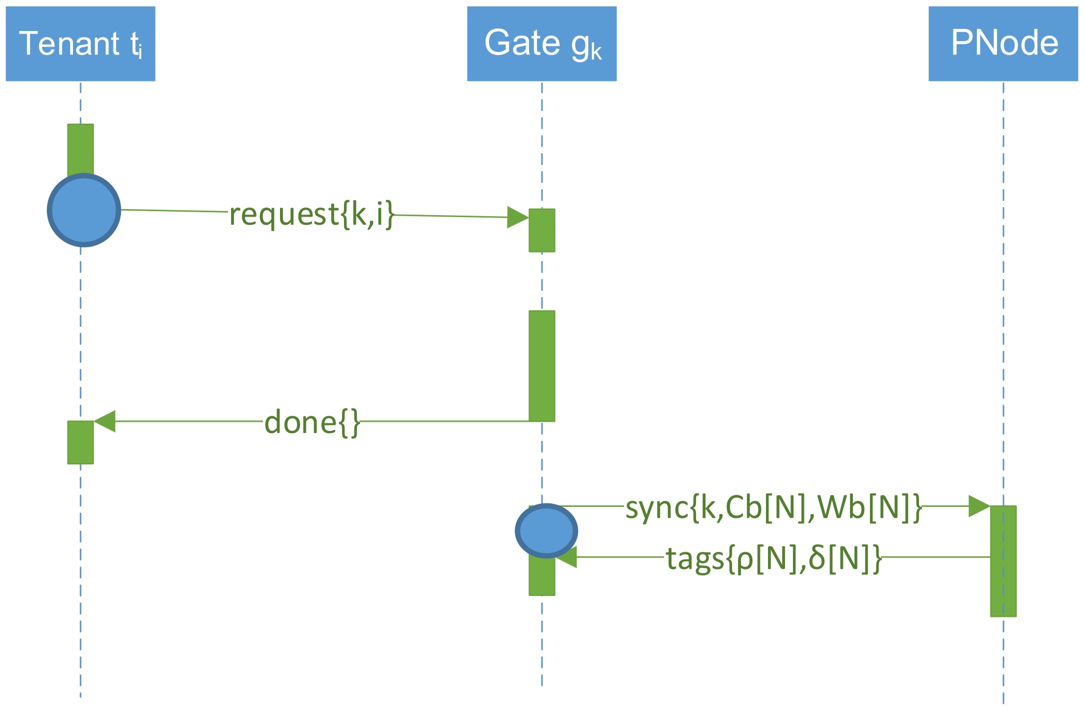
- The gate should inform the PNode about each request and wait for response to get parameters ρ and δ, which introduces a constant round-trip latency.

□ Single point failure

- When PNode crushes, it takes time to switch to a backup node. During the process, gates cannot continue their scheduling.

m2Clock v2: Updating in batch

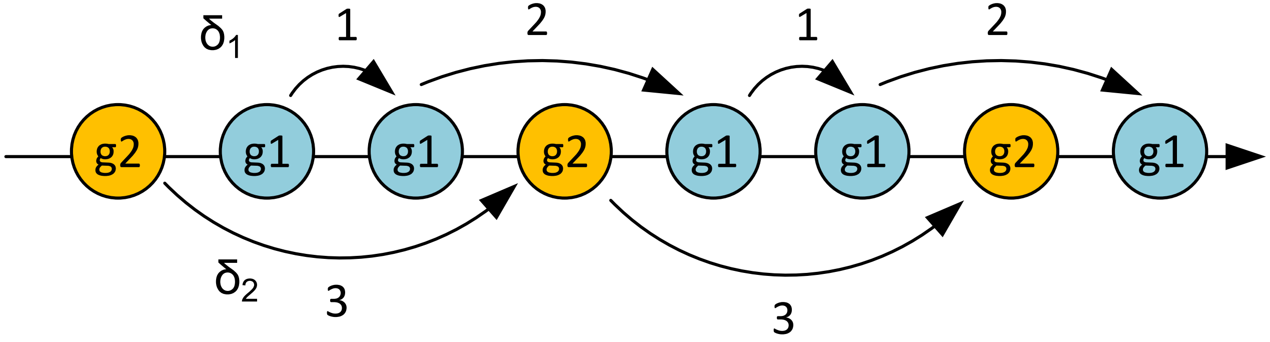
- Relax the strict bounding for better performance
  - Each gate has a local version of ρ and δ, and assign the requests accordingly on their arrival.
  - Gates synchronize those parameters from PNode periodically in background.



m2Clock v2: how to calculate ρ and δ

- Rethinking the physical meaning of ρ and δ arguments in dmClock
  - They are related to the ***proportion*** of requests that is handled by the given Gate

- E.g. Gate handles about 1-in-δ of all requests that is generated by a tenant
  - g_1: 1,2,1,2,… => $\delta_1 = \frac{3}{2}$, so **2/3** requests are sent to g_1
  - g_2: 2,2,2,… => $\delta_2 = 3$, so **1/3** requests are sent to g_2
  - Inversely, we can get δ_1 and δ_2 from the proportion of requests



m2Clock v2: pros & cons

- □ Advantage over v1
  - – Reduce the workload for PNode
  - – Avoid the round-trip latency before assigning tags
  - – Gates are able to schedule requests with old ρ and δ arguments even if the PNode crashes

- □ Disadvantage
  - – Not that accurate as dmClock, especially in burst scenarios

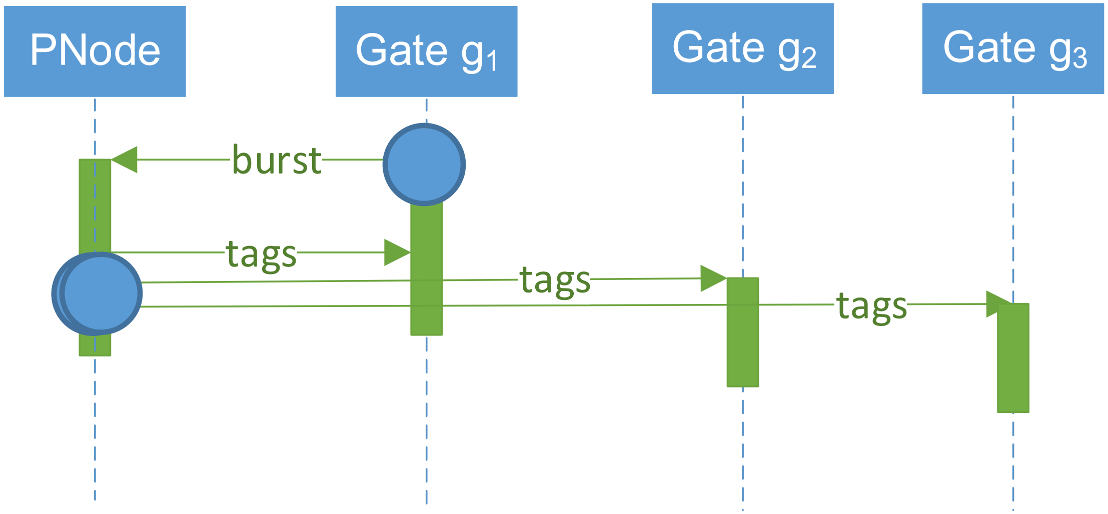
m2Clock v3: Local adjustment

- Allow each gate to adjust its local ρ and δ accordingly.
  - When a tenant starts to send a burst of I/O requests, the gate may perform a local adjustment for ρ and δ.

- Calculate the parameters from the time intervals of adjacent requests $\{\tau\}$:
  - Forecast: $\begin{cases} \rho_i = c_1 + \sum_{k=0}^{n} \phi_k \tau_{i-k} \\ \delta_i = c_2 + \sum_{k=0}^{n} \phi'_k \tau_{i-k} \end{cases}$
  - Learning: the model is trained on PNode, which has a complete collection of time series of requests and gets the actual value of ρ and δ

m2Clock v4: Burst broadcast

- Another way is to **do the synchronization immediately** when a burst occurs:
  - Gate g meets a burst from Tenant t, it will inform PNode with the information
  - Besides a common response with ρ and δ, PNode will also inform all other gates to update their ρ and δ



m2Clock v4: Burst detection

- If any ρ and δ varies k%, then it is identified as a burst
  - A smaller k: PNode will have to do the broadcast all the time.
  - A larger k: v4 may just degrades to the origin v2 without broadcasting.

- Simple adaptive burst detection
  - Given a range of broadcast density: M~N times per second, and adjust the k accordingly

Outline

□ Introduction

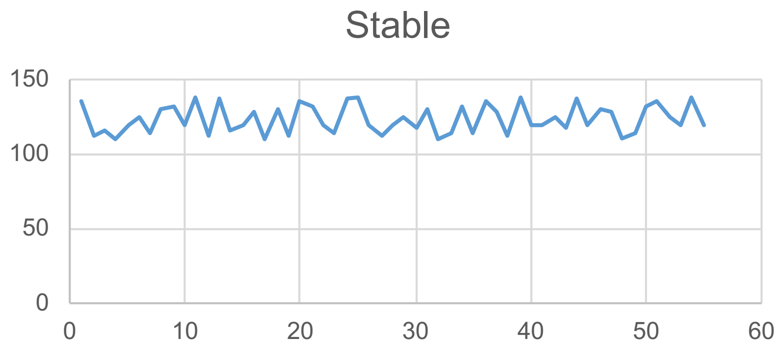
□ m2Clock methods

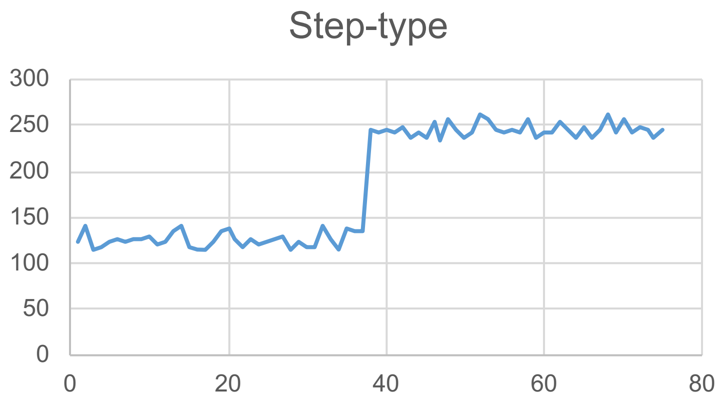
□ Evaluation results

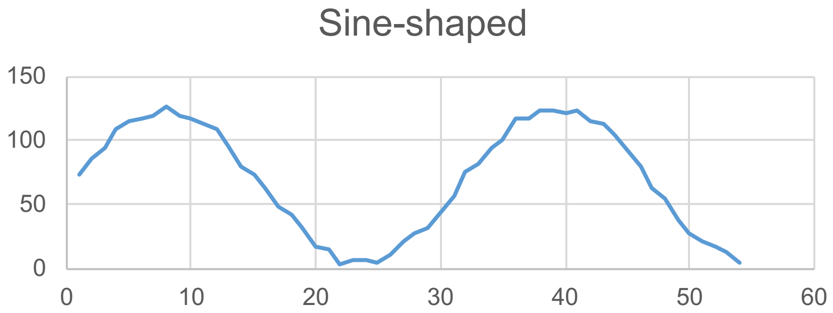
□ Conclusion

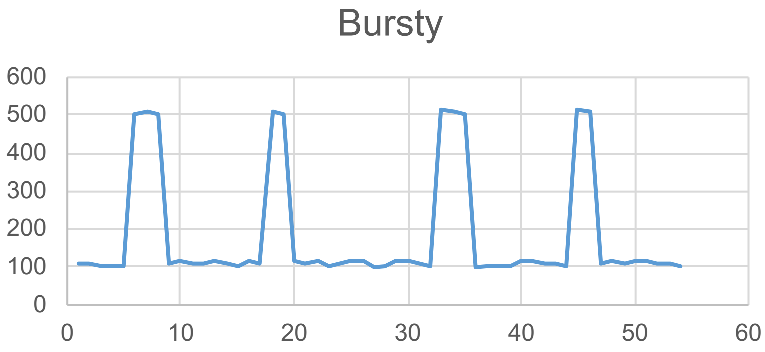
Evaluation

- Workload types
  - Stable, Step-type, Sine-shaped, Bursty


Stable

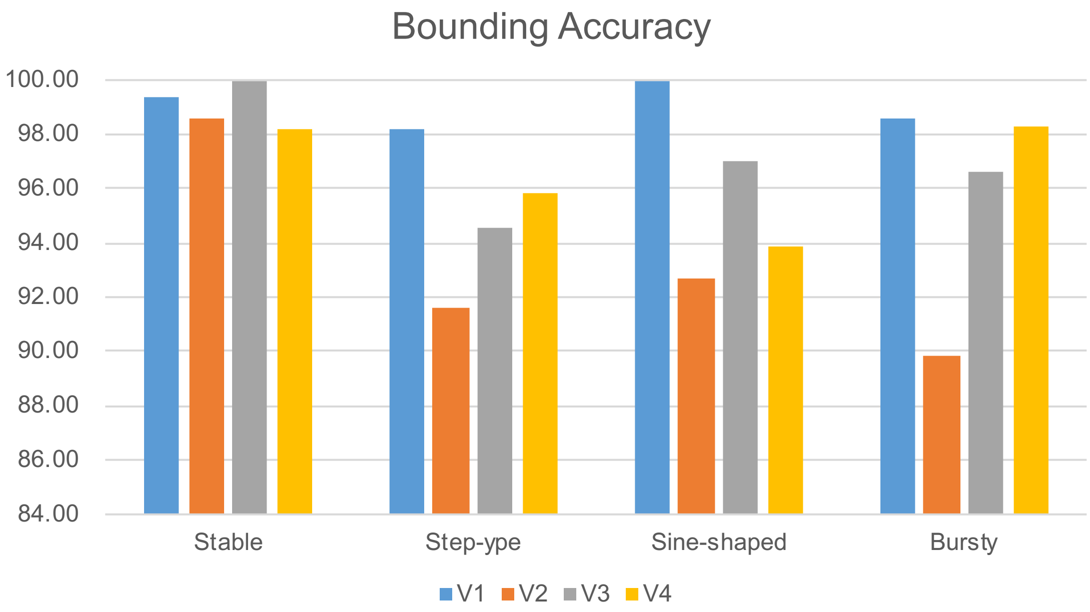

Step-type


Sine-shaped


Bursty

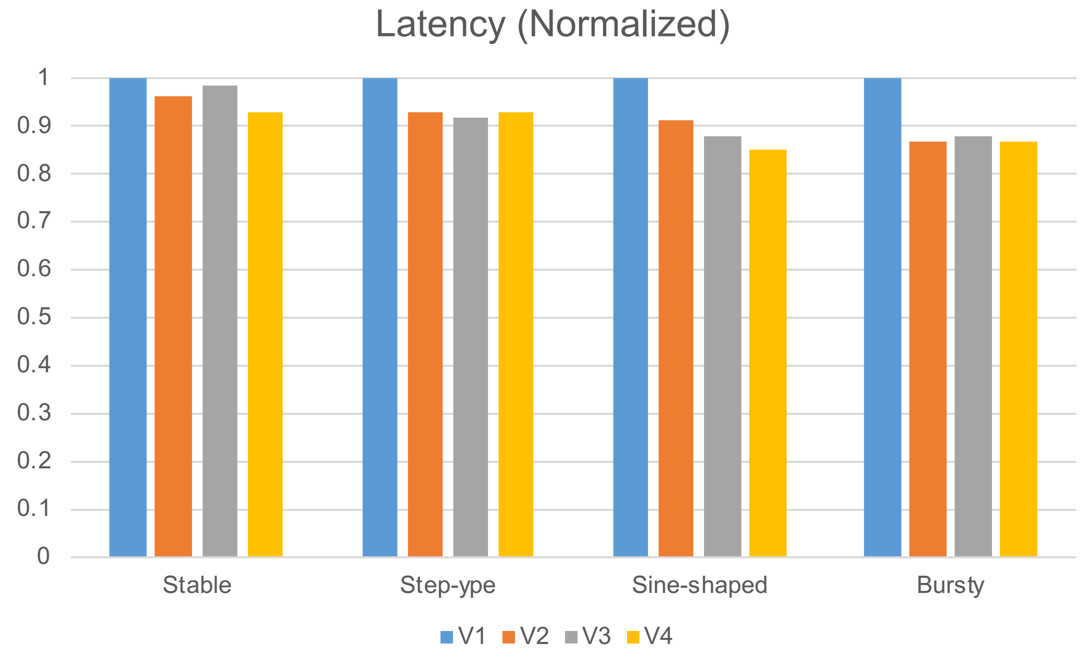
Evaluation results: Bounding Accuracy

- Percentage of time that IOPS is bounded in <reservation, limit>.



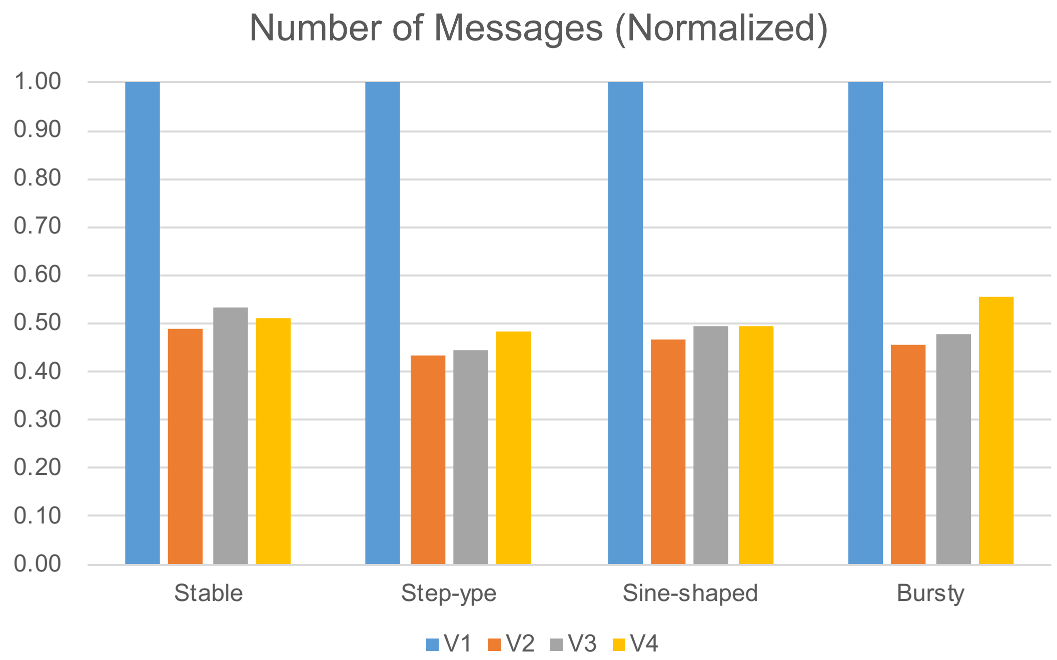
Evaluation results: Latency

□ The latency result is normalized according to V1.



Evaluation results: Number of Messages

- Messages that is passed between nodes. The result is normalized according to V1.



m2Clock: A brief comparison

	Accuracy	Latency	Scalability
V1: Centralized dmClock	High	High	Low
V2: Updating in batch	Low	Low	High
V3: Local adjustment	Medium	Low	High
V4: Burst broadcast	Medium	Low	High

- V3: Also works for cases that number of I/O requests changes smoothly.
- V4: performs better with abrupt bursts.

Conclusion

- □ We extend the dmClock method to work with shared cloud storage service.
  - – Bound the IOPS between <reservation, limit> for each tenant.
  - – Adding a centralized parameter node, called **PNode**.

- □ Four m2Clock methods
  - – Mitigate the communication overhead
  - – Make the bounding more accurate

Thank you!