Competitive Clustering of Stochastic Communication Patterns on the Ring

Chen Avin    Louis Cohen    Stefan Schmid

Nice to meet you!
The Network Matters

- Cloud-based applications generate significant network traffic
  - E.g., scale-out databases, streaming, batch processing applications

- E.g., Hadoop Terrasort job:

![Graph showing bandwidth usage over time]
Virtual machine placement affects bandwidth costs

Example: map reduce in Clos datacenter
Virtual machine placement affects bandwidth costs

Example: map reduce in Clos datacenter
Virtual machine placement affects bandwidth costs

Example: map reduce in Clos datacenter

Distributed across pods: costly shuffling!
Virtual machine placement affects bandwidth costs

Example: map reduce in Clos datacenter

Locally clustered within a rack or pod: efficient!
Virtual machine placement affects bandwidth costs.

Example: Map-reduce in a clos datacenter.

- Locally clustered within a rack or pod: efficient!
- Communication patterns are often clustered (but can change over time).
How to support local communication?
How to support local communication?

Option 1: Change the topology (?!)

How to support local communication?

Option 1: Change the topology (?!)
- Theory of demand-aware networks
- Prototypes emerging: e.g., ProjectToR (SIGCOMM 2016)
- Based on lasers and mirrors
How to support local communication?

Option 1: Change the topology (?!)
- The case of demand-aware networks
- Prototypes: e.g., ProjectToR (SIGCOMM 2016)
- Based on lasers and mirrors

We are working on it! E.g., „SplayNets @ TON 2016“. But not today!
How to support local communication?

Option 1: Change the topology (?!)
- Theory of demand-aware networks
- Prototypes emerging: e.g., ProjectToR (SIGCOMM 2016)
- Based on lasers and mirrors

Option 2: Cluster the nodes
- Migrate frequently communicating nodes closer together
How to support local communication?

Option 1: Change the topology (?!)
- Theory of demand-aware networks
- Prototypes emerging: e.g., ProjectToR (SIGCOMM 2016)
- Based on lasers and mirrors

Option 2: Cluster the nodes
- Migrate frequently communicating nodes closer together
How to support local communication?

**Option 1: Change the topology (?!)**
- Theory of demand-aware networks
- Prototypes emerging: e.g., ProjectToR (SIGCOMM 2016)
- Based on lasers and mirrors

**Option 2: Cluster the nodes**
- Migrate frequently communicating nodes closer together

**Challenges of communication pattern clustering:**
- Communication patterns are not known ahead of time...
- ... and may even change over time!
How to support local communication?

Option 1: Change the topology (?!)
- Theory of demand-aware networks
- Prototypes emerging: e.g., ProjectToR (SIGCOMM 2016)
- Based on lasers and mirrors

Option 2: Cluster the nodes
- Migrate frequently communicating nodes closer together

Challenges of communication patterns clustering:
- Communication patterns are not known ahead of time...
- ... and may even change over time!

Thus: Need to repartition clusters in an online manner, depending on demand!
Example: A *Re*Partitioning Problem

- Example: 4 clusters of size 4

How to cluster?
Example: A RePartitioning Problem

- Example: 4 clusters of size 4

Thickness of line = amount of communication

How to cluster?
Example: A RePartitioning Problem

- Example: 4 clusters of size 4
Example: A *Re*Partitioning Problem

- Example 1: 4 clusters of size 4

Most communication within cluster (intra-cluster)...

... little inter-cluster communication.
Example: A RePartitioning Problem

- Example: 4 clusters of size 4

- Now assume: changes in communication pattern!
  - E.g., more communication (1,3),(3,4),(2,5) but less (5,6)
Example: A RePartitioning Problem

- Example: 4 clusters of size 4

- Now assume: changes in communication pattern!
  - E.g., more communication (1,3),(3,4),(2,5) but less (5,6)
Example: A RePartitioning Problem

- Example: 4 clusters of size 4

- Now assume: changes in communication pattern

  - E.g., more communication (1,3),(3,4),(2,5) but less (5,6)

Nodes 1 and 5 change clusters!
A simple and fundamental model (e.g., a rack):

Online RePartitioning

size $k$ ("# slots")

$\ell$ servers ("clusters")
Online RePartitioning

A simple and fundamental model (e.g., a rack):

Minimize inter-cluster communication...

\( \ell \) servers („clusters“)

... maximize intra-cluster communication!
Online RePartitioning

A simple and fundamental model (e.g., a rack):

- Minimize inter-cluster communication...
- \( \ell \) servers („clusters“)
- Also: minimize migrations (=swap)!
- ... maximize intra-cluster communication!
A simple and fundamental model:

Online RePartitioning

In practice: $k \ll \ell$ (many more servers than VM slots per server)!

Minimize inter-cluster communication...

$\ell$ servers ("clusters")

Also: minimize migrations (=swap)!

... maximize intra-cluster communication!
Online RePartitioning

Problem inputs: $k, \ell, \sigma = \{u_1, v_1\}, \{u_2, v_2\}, \{u_3, v_3\}, \ldots$
Online RePartitioning

Problem inputs: $k, \ell, \sigma = \{u_1, v_1\}, \{u_2, v_2\}, \{u_3, v_3\}, \ldots$

Costs:

Objective: $\text{ALG}(\sigma) = \sum_{t=1}^{\sigma} \text{mig}(\sigma_t; \text{ALG}) + \text{com}(\sigma_t; \text{ALG})$
Online RePartitioning

Problem inputs: $k, \ell, \sigma = \{u_i\}_{i=1}^{\ell}$

Two flavors: (1) online (worst-case) pattern
(2) learning: from a fixed (unknown) distribution

Costs:

Objective: $\text{ALG}(\sigma) = \sum_{t=1}^{\ell} \text{mig}(\sigma_t; \text{ALG}) + \text{com}(\sigma_t; \text{ALG})$
The Crux: Algorithmic Challenges

A) Serve remotely or migrate ("rent or buy")? When to migrate? If a communication pattern is short-lived, it may not be worthwhile to collocate the nodes: the migration cost cannot be amortized.
A) **Serve remotely or migrate (“rent or buy”)?** When to migrate? If a communication pattern is short-lived, it may not be worthwhile to collocate the nodes: the migration cost **cannot be amortized**.

B) **Where to migrate, and what?** If nodes should be collocated, the question becomes **where**. Should the first node be **migrated to** the cluster of the second or vice versa? Or shall **both be moved** together to a new cluster? Moreover, an algorithm may be required to **pro-actively migrate (resp. swap)** additional nodes.
The Crux: Algorithmic Challenges

A) Serve remotely or migrate ("rent or buy")? When to migrate? If a communication pattern is short-lived, it may not be worthwhile to collocate the nodes: the migration cost cannot be amortized.

B) Where to migrate, and what? If nodes should be collocated, the question becomes where. Should the first node be migrated to the cluster of the second or vice versa? Or shall both be moved together to a new cluster? Moreover, an algorithm may be required to pro-actively migrate (resp. swap) additional nodes.

C) Which nodes to evict? There may not exist sufficient space in the desired destination cluster. In this case, the algorithm needs to decide which nodes to evict, to free up space.
Goal: minimize competitive ratio

\[ \rho(\text{ON}) = \max_{\sigma} \frac{\text{ON}(\sigma)}{\text{OFF}(\sigma)} \]
Online Variant: Competitive Ratio and Augmentation

- Goal: minimize competitive ratio

\[ \rho(ON) = \max_\sigma \frac{ON(\sigma)}{OFF(\sigma)} \]

- Two flavors: without and with augmentation
Let’s first look at special case: $k=2$
Let’s first look at special case: $k=2$

Need to find pairs!
Let’s first look at special case: $k=2$

Clusters of size 2: A new type of online matching problem!
Special Cases: $\ell = 2$
Special Cases: $\ell = 2$

2 Clusters: A generalization of online caching!
Special Cases: $\ell = 2$ ("Online Caching")

- For 2 clusters: can emulate online caching!
  - $k$ items, cache size $k - 1$
Special Cases: $\ell = 2$ ("Online Caching")

- For 2 clusters: can emulate online caching!
- $k$ items, cache size $k-1$

... plus some dummy item

Cache...

$\ell = 2$

Cache

Disk

Cache
For 2 clusters: can emulate online caching!
- $k$ items, cache size $k-1$

When item $i$ is requested in original caching problem:
- Introduce many requests between $d$ and $i$: forces $i$ to cache (if it is not yet)
Special Cases: $\ell=2$ ("Online Caching")

- For 2 clusters: can emulate online caching!
  - $k$ items, cache size $k-1$

- When item $i$ is requested in original caching problem:
  - Introduce many requests between $d$ and $i$: forces $i$ to cache (if it is not yet)
  - Which one to evict? Caching problem!
For 2 clusters: can emulate online caching!
- $k$ items, cache size $k-1$

When item $i$ is requested in original caching problem:
- Introduce many requests between $d$ and $i$: forces $i$ to cache (if it is not yet)
- Which one to evict? Caching problem!
- Note: add many requests between $d$ and nodes currently in cache: $d$ stays in cache
Special Cases: $\ell=2$ (“Online Caching”)

- For 2 clusters: can emulate online caching!
  - $k$ items, cache size $k-1$

- When item $i$ is requested in original caching problem:
  - Introduce many requests between $d$ and $i$: forces $i$ to cache (if it is not yet)
  - Which one to evict? Caching problem!
  - Note: add many requests between $d$ and nodes currently in cache: $d$ stays in cache

Lower bound $k$ follows from caching!
Intriguing: Lower bound even with augmentation!
Intriguing: Lower bound even with augmentation!

- Assume: requests only from a certain (ring) order
Assume: requests only from a certain (ring) order

Adversarial strategy: Whatever ON does, adversary will ask cut edge (exists even with augmentation): pays 1 each time!

Intriguing: Lower bound even with augmentation!
Assume: requests only from a certain (ring) order

Adversarial strategy: Whatever ON does, adversary will ask cut edge (exists even with augmentation): pays 1 each time!

Note: Adversarial request sequence only depends on ON! So online algo cannot learn anything about OFF.

Intriguing: Lower bound even with augmentation!
Assume: requests only from a certain (ring) order

Adversarial strategy: Whatever ON does, adversary will ask cut edge (exists even with augmentation): pays 1 each time!

Note: Adversarial request sequence only depends on ON! So online algo cannot learn anything about OFF.

OFF can safely move to a partition which will be asked least frequently (once and forever)! Pigeon-hole principle: pays only every k-th time (i.e. k times less)
Online RePartitioning: Overview of Results

- $k=2$ (online matching)
  - Greedy algorithm $7$-competitive
  - Lower bound: $3$-competitive

- $O(k \log k)$-competitive algorithm CREP for $4$-augmentation
  - based on growing components
Online *Re*Partitioning: Overview of Results

- $k=2$ (online matching)
  - Greedy algorithm 7-competitive
  - Lower bound: 3-competitive

- $O(k \log k)$-competitive
  - based on growing components

Open question: what about less augmentation?
Learning Variant

- Adversary cannot choose request sequence but only the distribution
- Adversary needs to sample i.i.d. from this distribution
- Moreover: Adversary knows (deterministic or randomized) «learning» algorithm
Learning Variant

- Adversary cannot choose request sequence but only the distribution
- Adversary needs to sample i.i.d. from this distribution
- Moreover: Adversary knows (deterministic or randomized) «learning» algorithm

- Let’s start simple: communication along ring only
- I.e., adversary picks distribution over ring
Learning Variant

- Adversary cannot choose request sequence but only the distribution
  - Adversary needs to sample i.i.d. from this distribution
  - Moreover: Adversary knows (deterministic or randomized) «learning» algorithm

- Let’s start simple: communication along ring only
  - I.e., adversary picks distribution over ring

Avoid high-weight edges on the cut!
The Crux: *Joint* Optimization of Efficient Learning *and* Searching

- **Naive idea 1:** Take it easy and first learn distribution
  - Do not move but just sample requests in the beginning: until exact distribution has been learned whp
  - Then move to the best location for good
Naive idea 1: Take it easy and first learn distribution

- Do not move but just sample requests in the beginning: until exact distribution has been learned whp
- Then move to the best location for good

Waiting can be very costly: maybe start configuration is very bad and others similarly good! Not competitive! Need to move early on, away from bad locations!
The Crux: *Joint* Optimization of Efficient Learning and Searching

- **Naive idea 1:** Take it easy and first learn distribution
  - Do not move but just sample requests in the beginning: until exact distribution has been learned whp
  - Then move to the best location for good

- **Naive idea 2:** Pro-actively always move to the lowest cost configuration seen so far
The Crux: *Joint* Optimization of Efficient Learning *and* Searching

- **Naive idea 1:** Take it easy and first learn distribution
  - Do not move but just sample requests in the beginning: until exact distribution has been learned whp
  - Then move to the best location for good

- **Naive idea 2:** Pro-actively always move to the lowest cost configuration seen so far

  Bad: if requests are uniform at random, you should not move! Migration costs cannot be amortized. Crucial difference to classic distribution learning problems: guessing costs!
The Crux: *Joint Optimization of Efficient Learning and Searching*

- Naive idea 1: Take it easy and first learn distribution
  - Do not move but just sample requests in the beginning: until exact distribution has been *learned*.
  - Then move to the best location for good.

- Naive idea 2: Pro-actively always move to the lowest cost configuration.
  - Bad, e.g., if requests are distributed uniformly at random: better not to move at all (moving costs cannot be amortized).

Only move when it pays off! But e.g., how to differentiate between uniform and "almost uniform" distribution?
Learning Algorithm: Rotate Locally!

- Mantra of our algorithm: Rotate!
  - Rotate early, but not too early!
  - And: rotate locally
Learning Algorithm: Rotate Locally!

- **Define conditions** for configurations: if met, **never go back** to it (we can afford it w.h.p.: seen enough samples)

- **Mantra of the Algorithm: Rotate!**
  - Rotate early, but not too early!
  - And: rotate **locally**
Mantra of our algorithm: Rotate!
- Rotate early, but not too early!
- And: rotate locally

If current configuration is eliminated, go to nearby configuration (in directed manner: no frequent back and forth)!
Learning Algorithm: Rotate Locally!

- Mantra of our algorithm: Rotate!
  - Rotate early, but not too early!
  - And: rotate \textit{locally}

- If current configuration is \textbf{eliminated}, go to \textit{nearby configuration} (in directed manner: no frequent back and forth)!

- \textbf{Growing radius strategy:} allow to move further only once amortized!
Learning Algorithm: Rotate Locally!

- Mantra: Rotate!
- Rotate early, but not too early!
- And: rotate locally!
  If current configuration is eliminated, go to nearby configuration (in directed manner: no frequent back and forth)!

$log(n)$-competitive w.h.p.
Conclusion

- Dynamic repartitioning: a natural new problem!

- Competitive ratio super-linear in $k$: ok in practice (independent of number of servers!)

- Open questions:
  - Online variant: With less augmentation? Randomized?
  - Learning variant: General communication pattern, beyond ring?