Motivation:
How to provide a predictable cloud application performance in unpredictable environments?
Kraken: Online and Elastic Resource Reservations for Multi-tenant Datacenters

Carlo Fuerst
TU Berlin

Stefan Schmid
Aalborg University

Lalith Suresh
TU Berlin

Paolo Costa
Microsoft Research
Kraken: Online and Elastic Resource Reservations for Multi-tenant Datacenters

Carlo Fuerst  Stefan Schmid  Lalith Suresh  Paolo Costa
TU Berlin  Aalborg University  TU Berlin  Microsoft Research

Related Work:
Oktopus (SIGCOMM 2011)
Proteus (SIGCOMM 2012)
Kraken: Online and Elastic Resource Reservations for Multi-tenant Datacenters

Carlo Fuerst  Stefan Schmid  Lalith Suresh  Paolo Costa
TU Berlin  Aalborg University  TU Berlin  Microsoft Research

Related Work:
Oktopus (SIGCOMM 2011)
Proteus (SIGCOMM 2012)

Offline only, only elastic bandwidth, no migrations, no worst-case guarantees, etc.
Cloud Computing + Networking?! 
*Network matters!*

- Scale-out databases, batch processing applications etc.: significant network traffic
  - Example Facebook: 33% of *execution time* due to communication
  - **Focus today: Batch-Processing / Map Reduce:** shuffle phase

- Therefore: predictable performance requires performance isolation and bandwidth reservations

---

Today's clouds: No bandwidth reservation!
Example: Bandwidth Requirements in Hadoop

Bandwidth utilization of a TeraSort job over time.

In red: desired bandwidth reservation

(Tasks inform Hadoop controller prior to shuffle phase; reservation with Linux tc.)

- Predictable performance requires reservations!
- But how to minimize reservations? And when?
Example: Bandwidth Requirements in Hadoop

Bandwidth usage over time

A virtual network embedding problem: minimize reserved bandwidth = path lengths.

- Predictable performance requires reservations!
- But how to minimize reservations? And when?
Performance Isolation in Virtualized Environments

App 1: Mobile Service
- Quality-of-Service & Resource Requirements

App 2: Big Data Analytics
- Computational & Storage Requirements

Realization and Embedding

Virtualization and Isolation
Performance Isolation in Virtualized Environments

App 1: Mobile Service
- Quality-of-Service & Resource Requirements

App 2: Big Data Analytics
- Computational & Storage Requirements

Focus today: network

Virtualization and Embedding

Virtualization and Isolation
Performance Isolation in Virtualized Environments

App 1: Mobile Service
Quality-of-Service & Resource Requirements

App 2: Big Data Analytics
Computational & Storage Requirements

Focus today: network

Embedding VMs closer: less bandwidth reservation (shorter paths)
Virtual Network Embeddings: Hard?

Start simple: exploit flexible routing between given VMs

- Integer multi-commodity flow problem with 2 flows?
- Oops: NP-hard
Virtual Network Embeddings: Hard?

Start simple: exploit flexible routing between given VMs

- Integer multi-commodity flow problem with 2 flows?
- Oops: NP-hard

Forget about paths: exploit VM placement flexibilities!

- Most simple: Minimum Linear Arrangement without capacities
- NP-hard (min-max and avg) 😞
Theory vs Practice

Goal in theory:
Embed as general as possible guest graph to as general as possible host graph

Reality:
Datacenters, WANs, etc. exhibit much structure that can be exploited! But also guest networks come with simple specifications
Theory vs Practice

This talk: Kraken: focus on batch-processing applications (like Hadoop) and virtual cluster abstractions

Embed as general as possible **guest graph** to as general as possible **host graph**

Reality:
Datacenters, WANs, etc. exhibit much **structure** that can be exploited! But also guest networks come with **simple specifications**

This talk: focus on (oversubscribed) Clos topologies.
Theory vs Practice

Goal in theory:
Embed as general as possible guest graph to as general as possible host graph.

Reality:
Datacenters, WANs, etc. exhibit much structure that can be exploited! But also guest networks come with simple specifications.

Comes in 2 flavors: unsplittable and Hose model. This talk: unsplittable.
Virtual Clusters: Abstraction for Batch Processing

- A prominent abstraction for batch-processing applications: Virtual Cluster $VC(n,b)$
  - Connects $n$ virtual machines to a «logical» switch with bandwidth guarantees $b$
  - A simple abstraction

![Diagram showing two clusters with bandwidth $b_1$ and $b_2$, each connecting $n_1$ and $n_2$ virtual machines respectively.](image-url)
Efficient Embedding of Virtual Clusters: Clos Topology

- Kraken is based on dynamic programming

Dynamic Program = optimal solutions for subproblems can efficiently be combined into an optimal solution for the larger problem!
Efficient Embedding of Virtual Clusters: General Topology

Algorithm:
- Try all possible locations for virtual switch
- Extend network with artificial source \( s \) and sink \( t \)
- Add capacities
- Compute min-cost max-flow from \( s \) to \( t \)
  (or simply: min-cost flow of volume \( n \))

Guaranteed integer if links are integer!
(E.g., successive shortest paths)
Long story short: By efficient virtual cluster embedding, Kraken solves unpredictable network performance problem!
Long story short: By efficient virtual cluster embedding, Kraken solves unpredictable network performance problem!

Problems solved!?
Not really: Resource needs change over time...
Temporal resource patterns are hard to predict

Resource allocations must be changed online

>20% variance

>50% variance in killed tasks

Completion times of jobs in the presence of speculative execution (left) and the number of speculated tasks (right)

Bandwidth utilization of 3 different runs of the same TeraSort workload (without interference)
Temporal resource patterns are hard to predict.

Resource allocations must be changed *online*.

EC2 likely to be much more noisy!

>20% variance

Bandwidth utilization of 3 different runs of the same *TeraSort workload* (without interference)

>50% variance in killed tasks

Completion times of jobs in the presence of speculative execution (*left*) and the number of speculated tasks (*right*)
Kraken: Elastic Reconfigurations

monitor progress

up/down BW
up/down CUs

unpredictable

interference

speculative execution
Kraken: Elastic Reconfigurations

Rational: User knows requirements best.

Rational: Provider knows substrate best.

monitor progress

up/down BW
up/down CUs

unpredictable
interference
speculative execution
Kraken: Elastic Reconfigurations

- Efficient virtual cluster embedding
- Scale up and down network resources and cluster size at runtime
- Support task migrations
- Provable performance guarantees: Tradeoff embedding quality vs migration cost
Kraken: Elastic Reconfigurations

- Efficient virtual cluster embedding
- Scale up and down network resources and cluster size at runtime
- Support task migrations
- Provable performance guarantees:
  - Tradeoff embedding quality vs migration cost

Allows to adjust performance and compensate for stragglers and interference!
Kraken: Elastic Reconfigurations

- Local migrations only: no need for re-embeddings!
- Scale up and down network resources and cluster size at runtime
- Support task migrations
- Provable performance guarantees: Tradeoff embedding quality vs migration cost

Can also help to improve acceptance ratio!
Kraken: Elastic Reconfigurations

- Efficient virtual cluster embedding
- Scale up and down network resources and cluster size at runtime
- Also allows to answer questions such as: by how much can I reduce the embedding footprint at migration cost $x$?
- Provable performance guarantees: Tradeoff embedding quality vs migration cost
Example: Elastic Resource Allocation Can Benefit From (Local) Migrations

- Upgrade of virtual cluster: bandwidth and compute unit
- Need to re-embed locally: insufficient bandwidth
Example: Elastic Resource Allocation Can Benefit From (Local) Migrations

- Upgrade of virtual cluster: bandwidth and compute unit
- Need to re-embed locally: insufficient bandwidth

Objective: Migrate as few as possible
highest priority: satisfy change request by trying all options (linear number)
compute minimal reconfiguration cost (dynamically, and given current configuration)
Among all solutions with min migration costs, minimize footprint.

**Algorithm 1 Algorithm upgrade(VC,x,δ)**

*Output: success or failure*

1: for all nodes v in the fat-tree: compute slotCount(v) values
2: \( m^* \leftarrow \infty; \quad F^* \leftarrow \infty; \quad \text{cog}^* \leftarrow \bot; \)
3: for all v in substrate do
4: \( M \leftarrow \text{minMig}(v) \)
5: if \( |M| \leq m^* \) then
6: \( F \leftarrow \text{footprint}(v,|M|) \)
7: if \( F < \infty \land (|M| < m^* \lor F < F^*) \) then
8: \( \text{cog}^* \leftarrow v \)
9: \( m^* \leftarrow |M| \)
10: \( F^* \leftarrow F \)
11: end if
12: end if
13: end for
14: if \( m^* = \infty \) then
15: return failure
16: end if
17: \( \mu \leftarrow \text{computeEmbedding}(VC,\text{cog}^*) \)
18: return success
Algorithm 1 Algorithm upgrade(VC,x,δ)

Output: success or failure

1: for all nodes v in the fat-tree: compute slotCount(v) values
2: \( m^* \leftarrow \infty; F^* \leftarrow \infty; cog^* \leftarrow \perp \)
3: for all v in substrate do
4: \( M \leftarrow \text{minMig}(v) \)
5: \( \text{if } |M| \leq m^* \text{ then} \)
6: \( F \leftarrow \text{footprint}(v, |M|) \)
7: \( \text{if } F < \infty \land \left( |M| < m^* \lor F < F^* \right) \text{ then} \)
8: \( \text{cog}^* \leftarrow v \)
9: \( m^* \leftarrow |M| \)
10: \( F^* \leftarrow F \)
11: end if
12: end if
13: end for
14: \( \text{if } m^* = \infty \text{ then} \)
15: return failure
16: end if
17: \( \mu \leftarrow \text{computeEmbedding}(VC, cog^*) \)
18: return success
Theorem IV.1. Kraken guarantees:

1) Request Satisfiability: As long as a feasible solution exists all upgrade and downgrade requests are satisfied.

2) Minimal Reconfiguration: The reconfiguration costs is always minimized. In particular, if a solution without migrations exists, it is used.

3) Optimal Allocation: Among all possible solutions with minimal reconfiguration costs, Kraken computes the one with the minimal embedding footprint.

4) Complexity: The time complexity of re-configuring (or embedding) a virtual cluster is bounded by $O(N \cdot n \cdot \Delta)$ in the worst-case, where $N$ is the size of the substrate (number of servers), $n$ is the virtual cluster size, and $\Delta = S + R + P$ is the number of servers in a single rack $S$ (i.e., the degree of a ToR switch), plus the number of racks in a single pod $R$ (i.e., the degree of an access switch), plus the number of pods $P$ (i.e., the degree of a core switch).
Benefit 1: Supporting Elasticity with Local Migrations

- Setting: oversubscribed Clos topology (16k servers, 10 pods at 40 racks at 40 servers), Oktopus workloads

Kraken can satisfy most reconfiguration requests locally.

Oktpus: resort to rembedding costly.
Benefit 2: Improved Acceptance Ratio with Reconfigurations

Migrations allow to accept additional requests!

- Cluster size upgrade
- Bandwidth upgrade
- Joint upgrade
Migrations are Feasible: Available Bandwidth

- Reconfigurations allow to accept additional requests!
- Less bandwidth for CU migrations

Sufficient available bandwidth for flow reconfigurations / migrations.
Migrations are Feasible: Available Bandwidth

- Reconfigurations allow to accept additional requests!
- Less bandwidth for CU migrations

For CU migrations, out-of-band control network may make sense.
Conclusion

- Predictable performance requires reservations: not always a computationally hard problem!

- But reservations need to be changed over time

- Kraken: tailored to batch-processing applications ("virtual cluster")
  - Optimal virtual cluster embeddings in linear time
  - Support for adjustments at runtime: bandwidth and nodes
  - Truly leverages the elastic allocation flexibilities of the cloud computing paradigm
  - Limited number of migrations, improved acceptance ratio
### Algorithm 1 Algorithm upgrade(VC, x, δ)

Output: success or failure

1. for all nodes v in the fat-tree: compute slotCount(v) values
2. m* ← ∞; F* ← ∞; cog* ← ⊥;
3. for all v in substrate do
   1. M ← minMig(v)
   2. if |M| ≤ m* then
      1. F ← footprint(v, |M|)
      2. if F < ∞ ∧ (|M| < m* ∨ F < F*) then
         1. cog* ← v
         2. m* ← |M|
         3. F* ← F
   3. end if
4. end for
5. return cog* or success

### Algorithm 2 minMig(substrate node v)

Output: set of CUs

1. M ← ∅
2. L ← computeConflictLinks(v)
3. sort L with decreasing distance from v
4. for all links ℓ ∈ L do
   1. while ℓ oversubscribed do
      2. let c be an arbitrary CU below ℓ
      3. M ← M ∪ {c}
   2. end while
5. end for
6. M ← M ∪ extraCUs(v)
7. return M

### Algorithm 3 footprint(substrate node v, number of CUs to migrate m)

Output: cost value

1. done ← 0
2. for all children v' of v in the fat-tree do
   1. done ← done + slotCount(v')
3. end for
4. return ST(v) + height(v) · n + costsAbove(v, m - done)