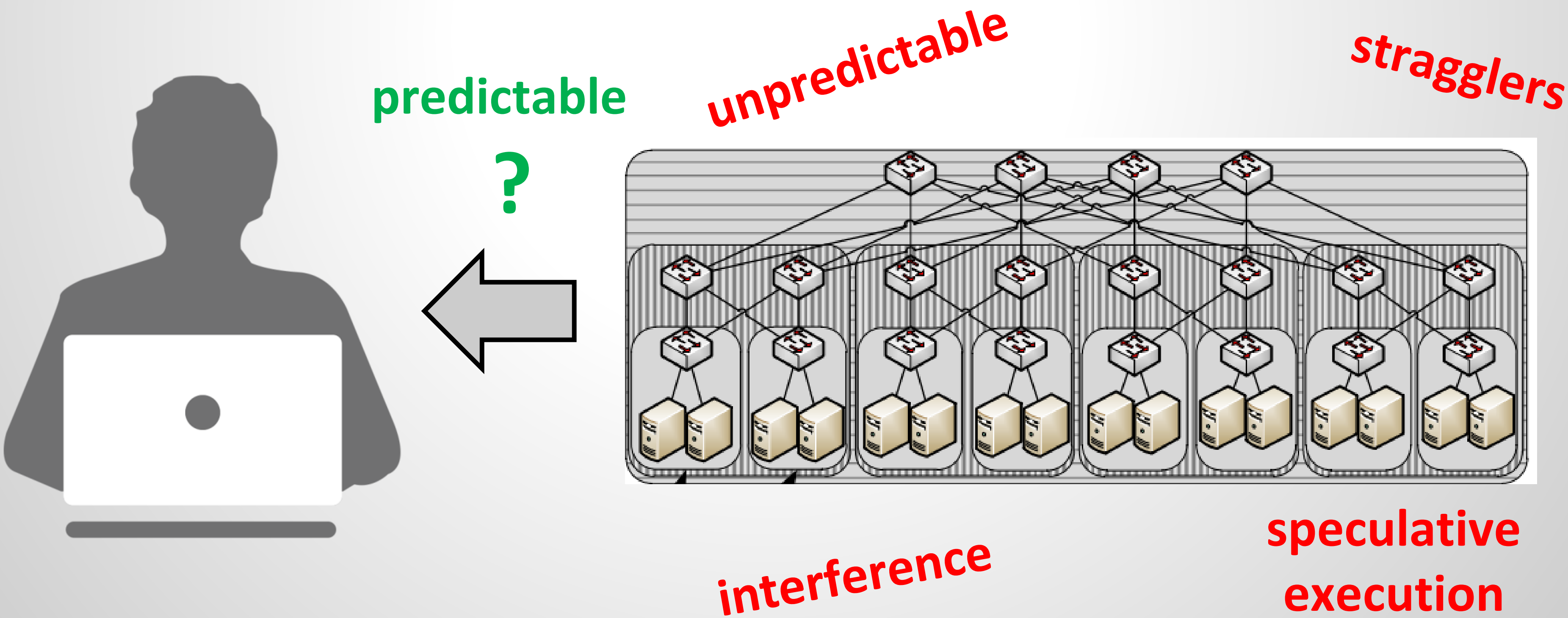
Motivation:

How to provide a predictable cloud application performance in unpredictable environments?



Kraken: Online and Elastic Resource Reservations for Multi-tenant Datacenters

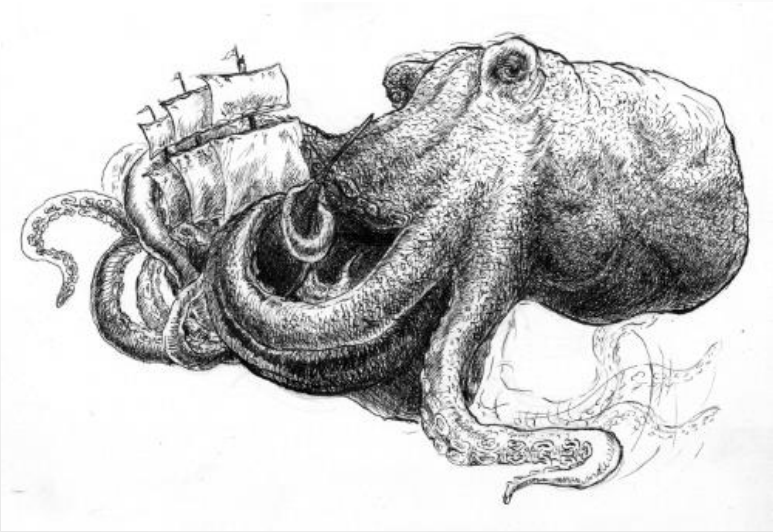
Carlo Fuerst **Stefan Schmid** Lalith Suresh Paolo Costa

TU Berlin Aalborg University TU Berlin Microsoft Research

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:

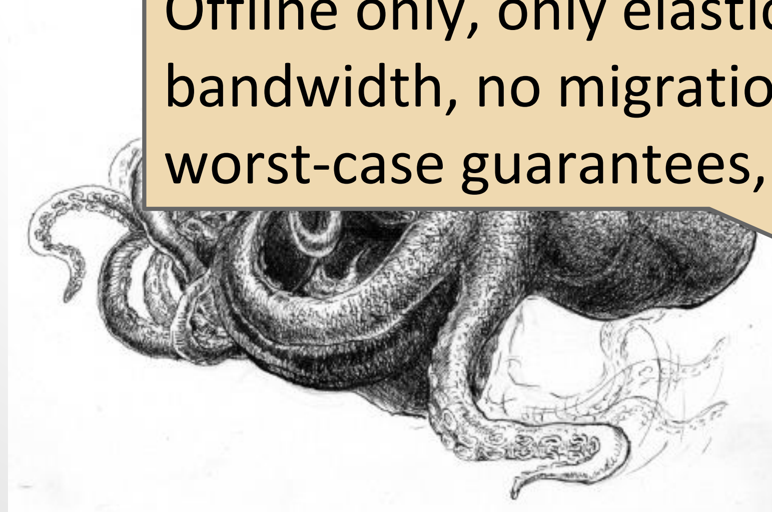
Oktopus (SIGCOMM 2011)

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



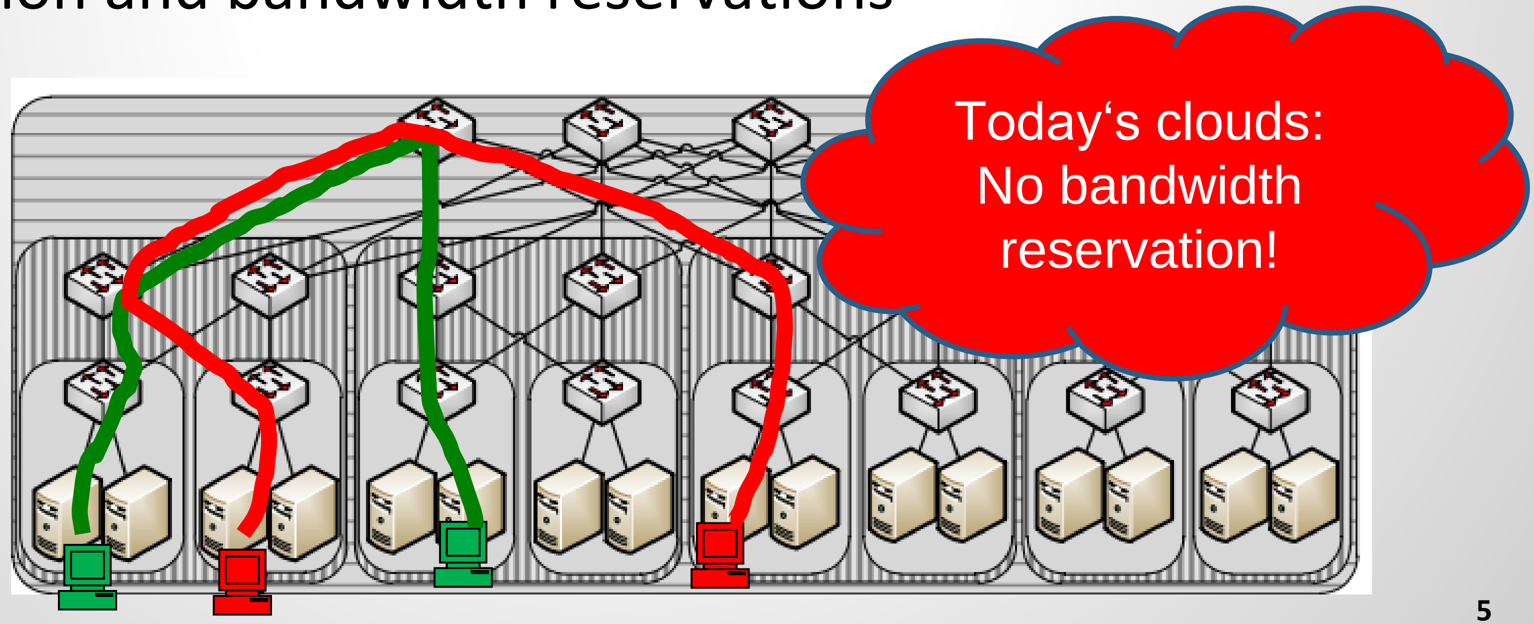
Offline only, only elastic bandwidth, no migrations, no worst-case guarantees, etc.

Related Work:

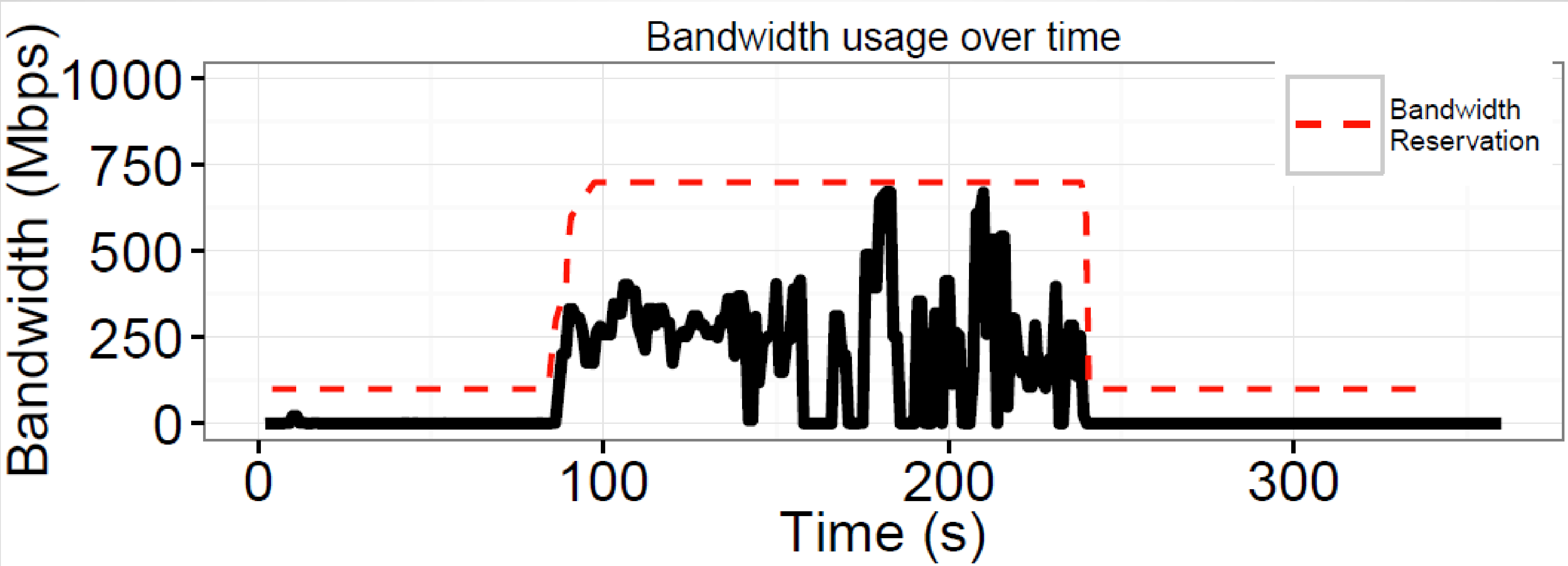
Oktopus (SIGCOMM 2011)
Proteus (SIGCOMM 2012)

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



Example: Bandwidth Requirements in Hadoop



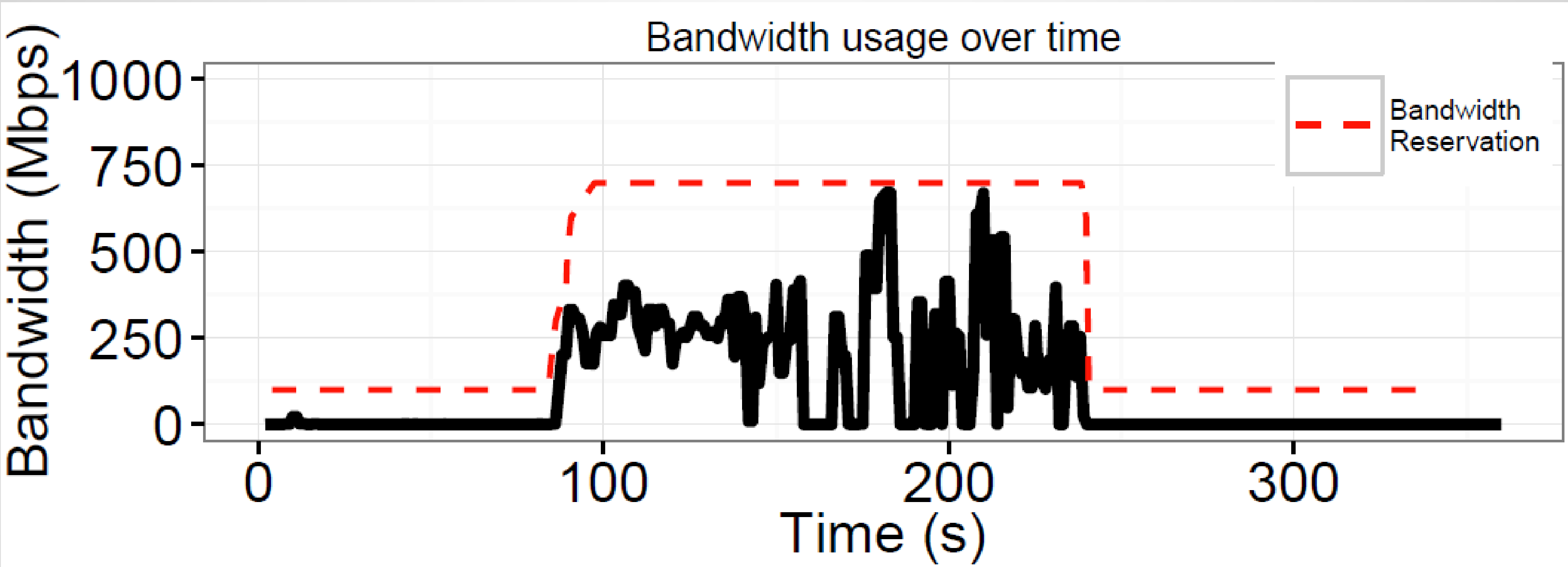
Bandwidth utilization of a TeraSort job over time.

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



eraSort job over time.

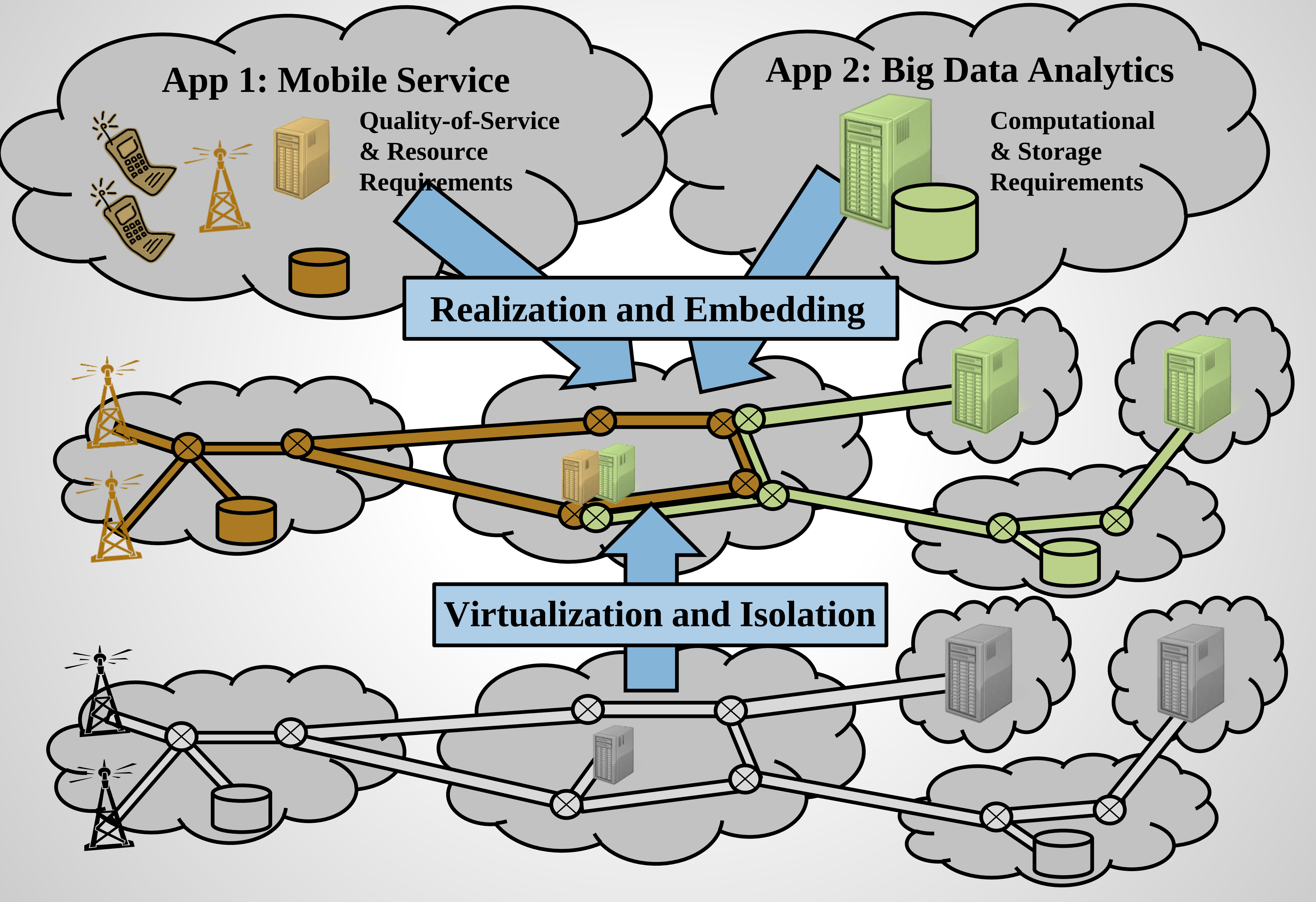
idth reservation

ffle phase; reservation with Linux *tc*.)

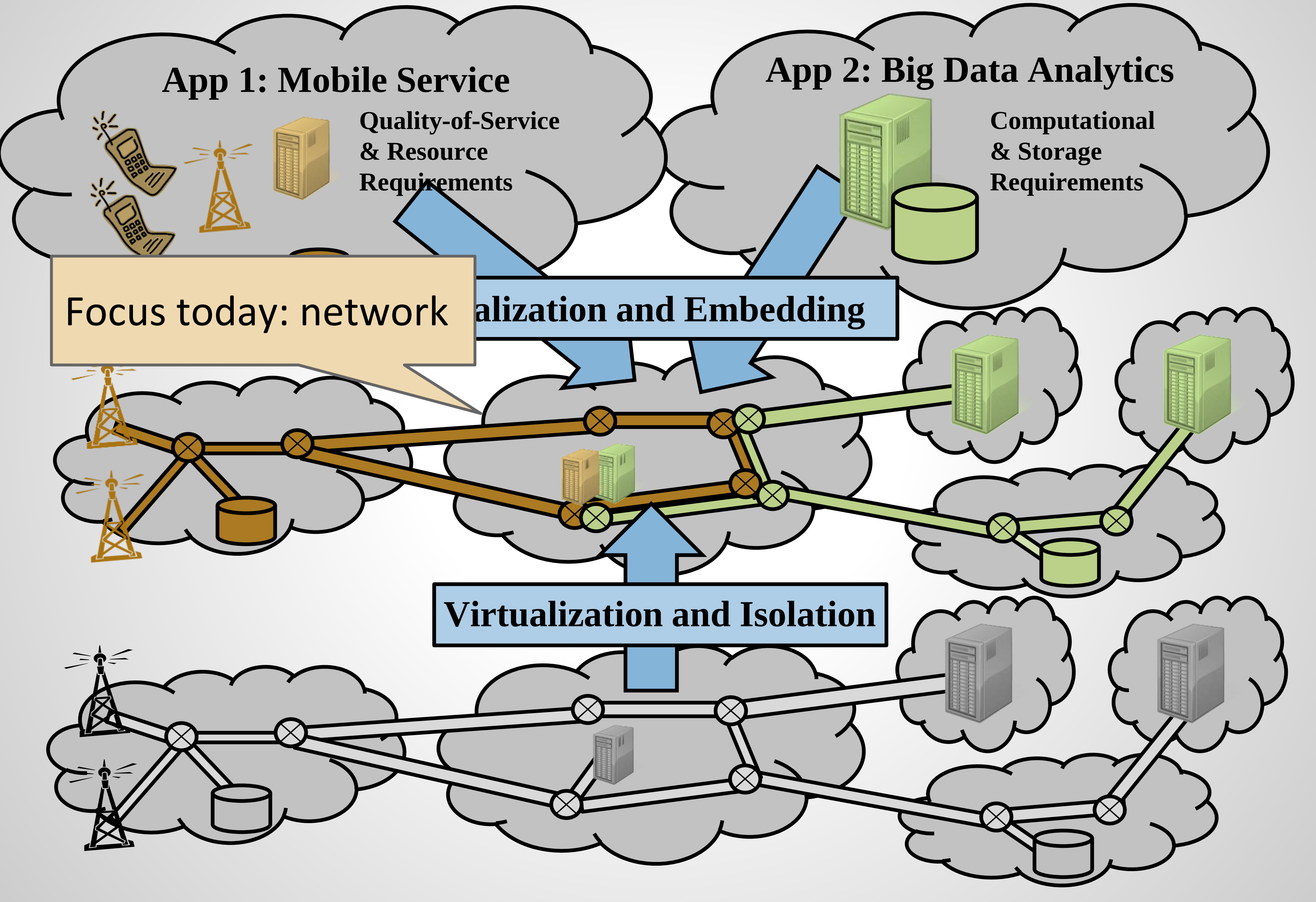
A virtual network embedding problem: minimize reserved bandwidth = path lengths.

- Predictable performance re...uires reservations!
- But how to minimize reservations? And when?

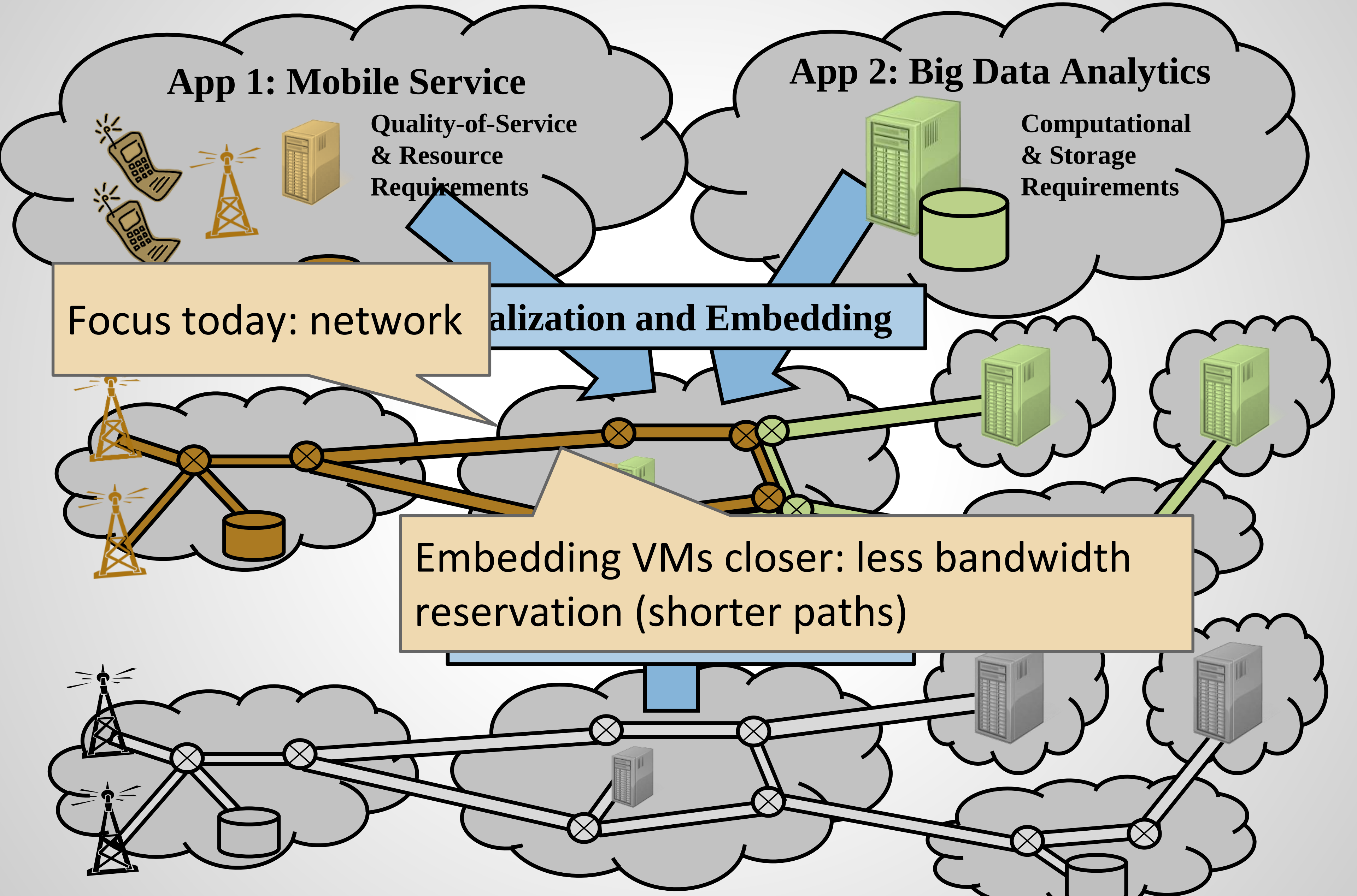
Performance Isolation in Virtualized Environments



Performance Isolation in Virtualized Environments



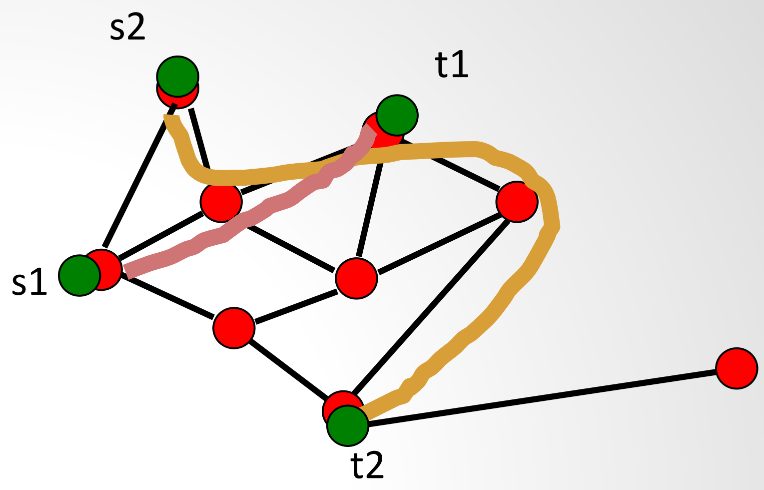
Performance Isolation in Virtualized Environments



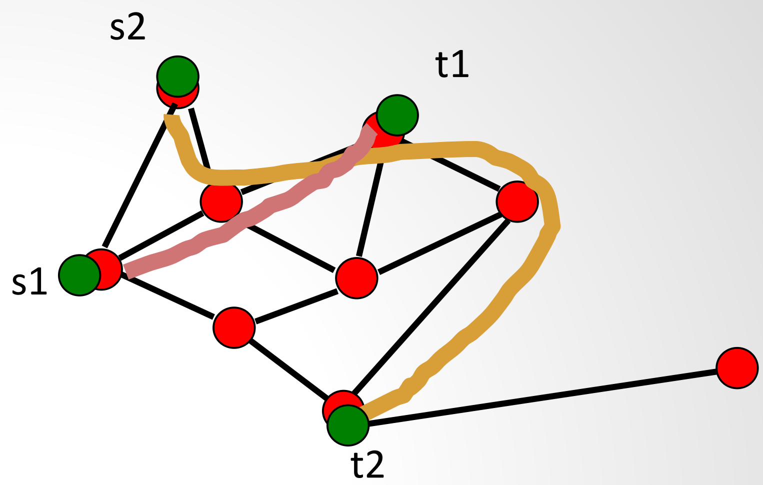
Virtual Network Embeddings: Hard?

Start simple: exploit flexible routing between given VMs

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

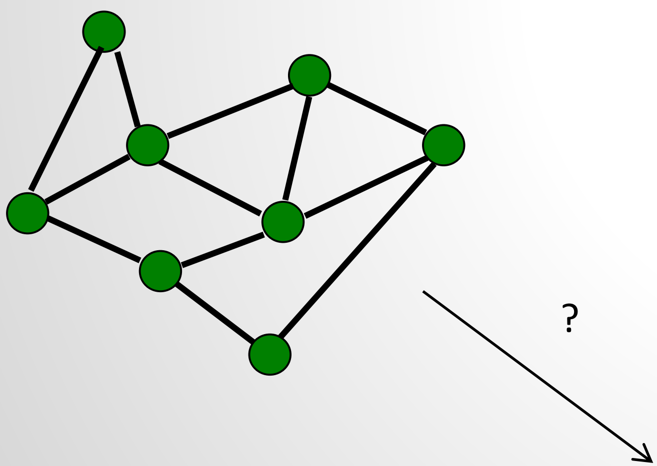


Virtual Network Embeddings: Hard?



Start simple: exploit flexible routing between given VMs

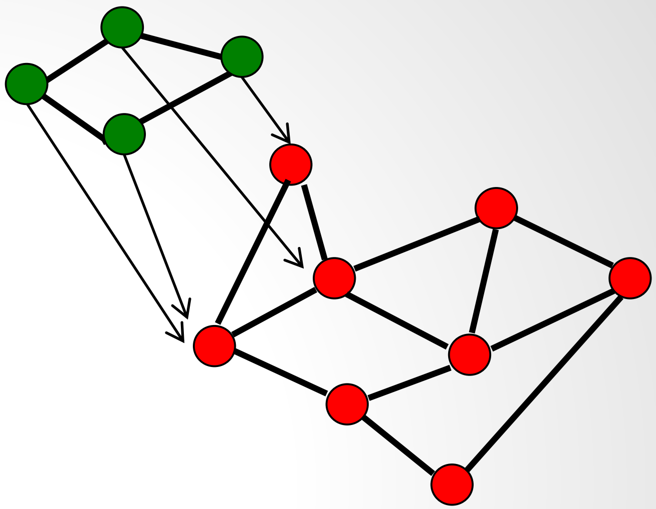
- Integer multi-commodity flow problem with 2 flows?
- Oops: NP-hard



Forget about paths: exploit VM placement flexibilities!

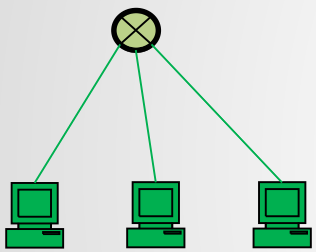
- Most simple: Minimum Linear Arrangement without capacities
- NP-hard (min-max and avg) ☹

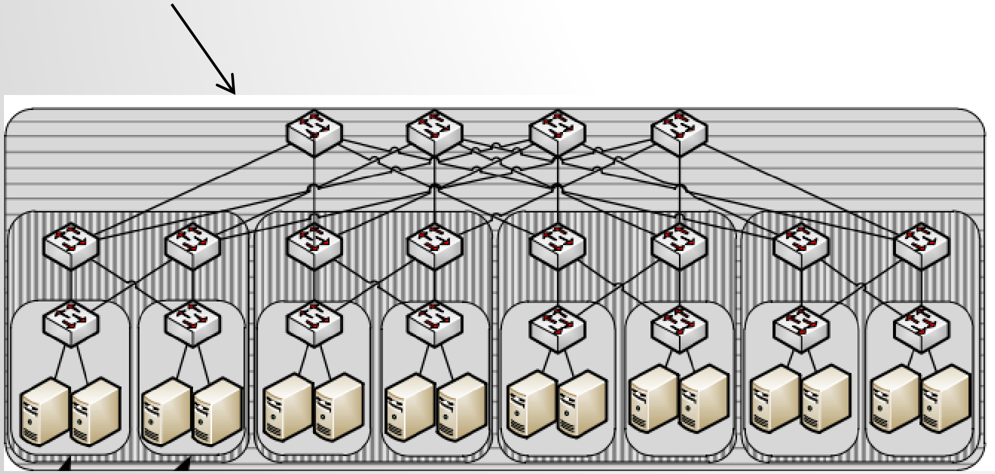
Theory vs Practice



Goal in theory:

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





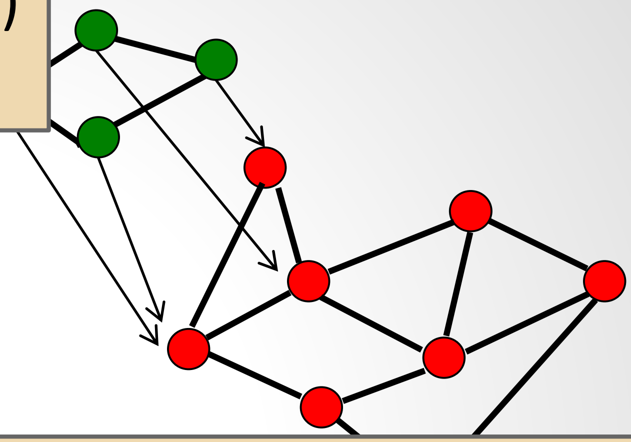
Reality:

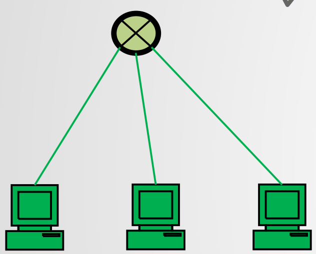
Datacenters, WANs, etc. exhibit much **structure** that can be exploited! But also guest networks come with **simple specifications**

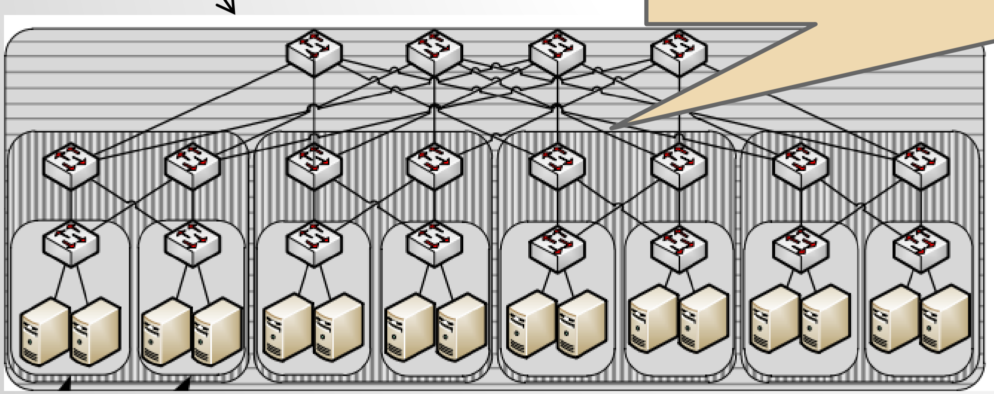
Theory vs Practice

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

Embed as … as possible ***guest graph*** to as gener… possible ***host graph***







This talk: focus on (oversubscribed) Clos topologies.

Reality:

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

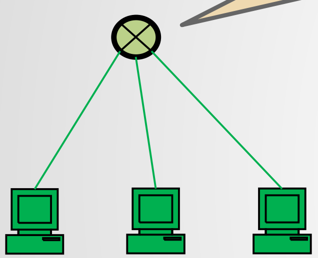
Theory vs Practice

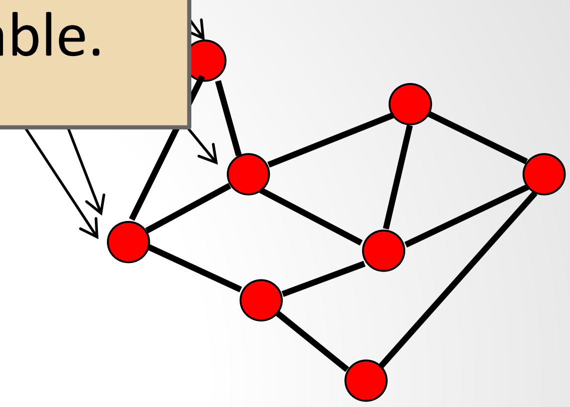
Comes in 2 flavors: unsplittable and Hose model. This talk: unsplittable.

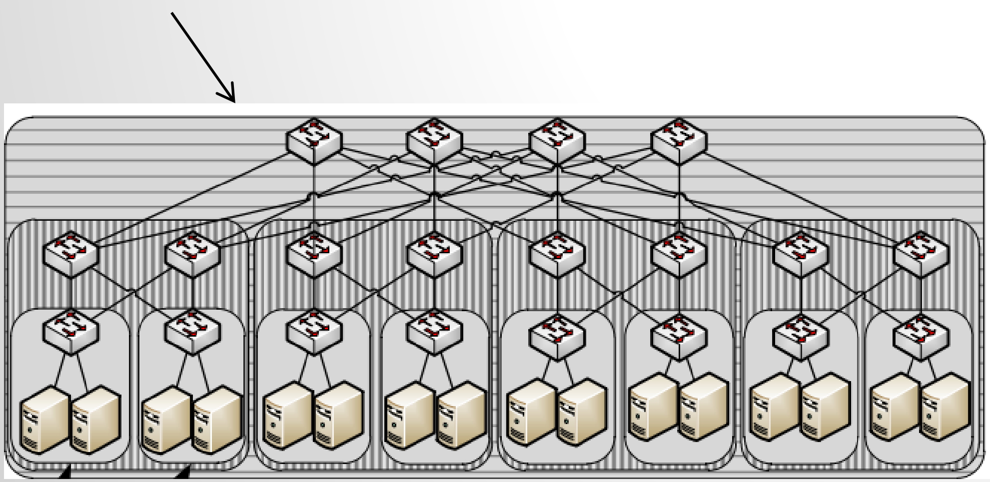
Goal in th

Embed as

to as general as ... host graph





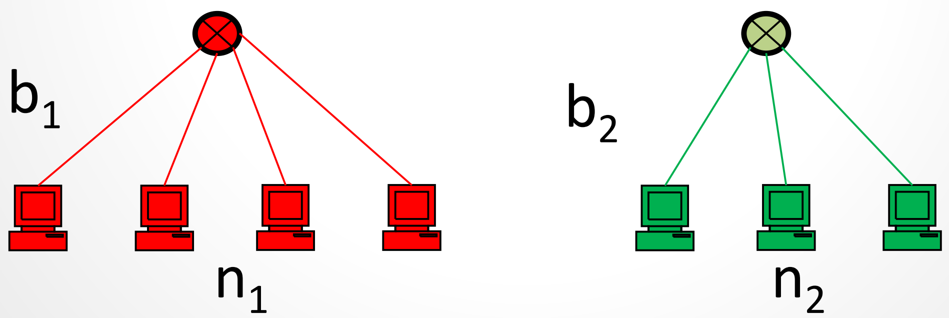


Reality:

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

Virtual Clusters: Abstraction for Batch Processing

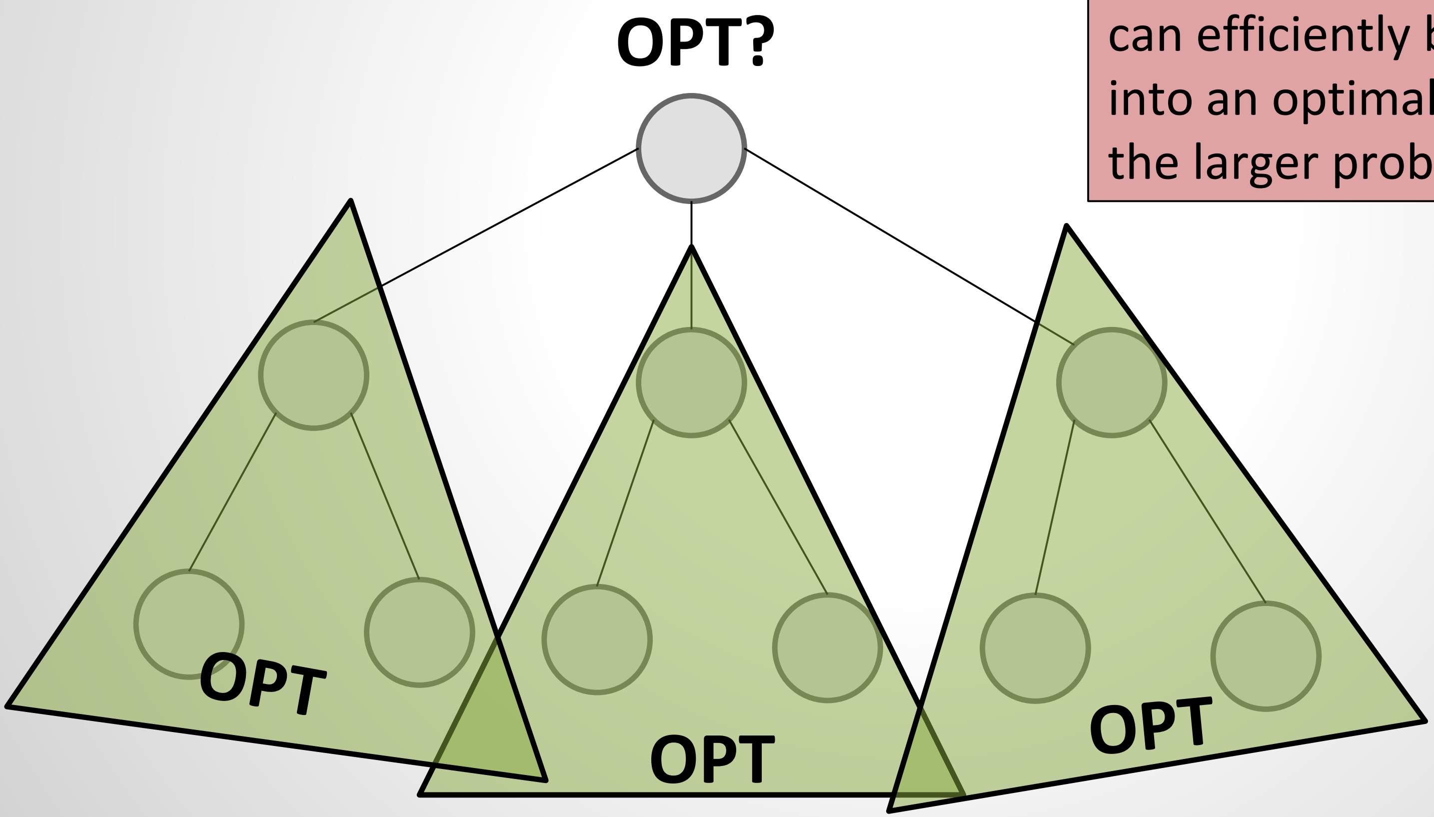
- A prominent abstraction for batch-processing applications: Virtual Cluster *VC(n,b)*
  - Connects *n* virtual machines to a «logical» switch with bandwidth guarantees *b*
  - A simple abstraction



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!

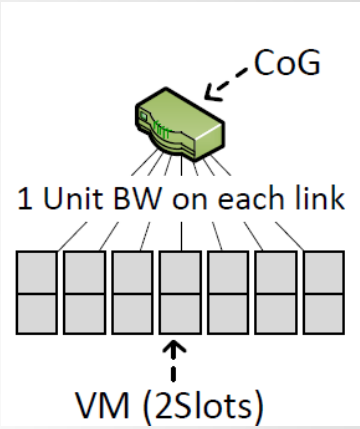


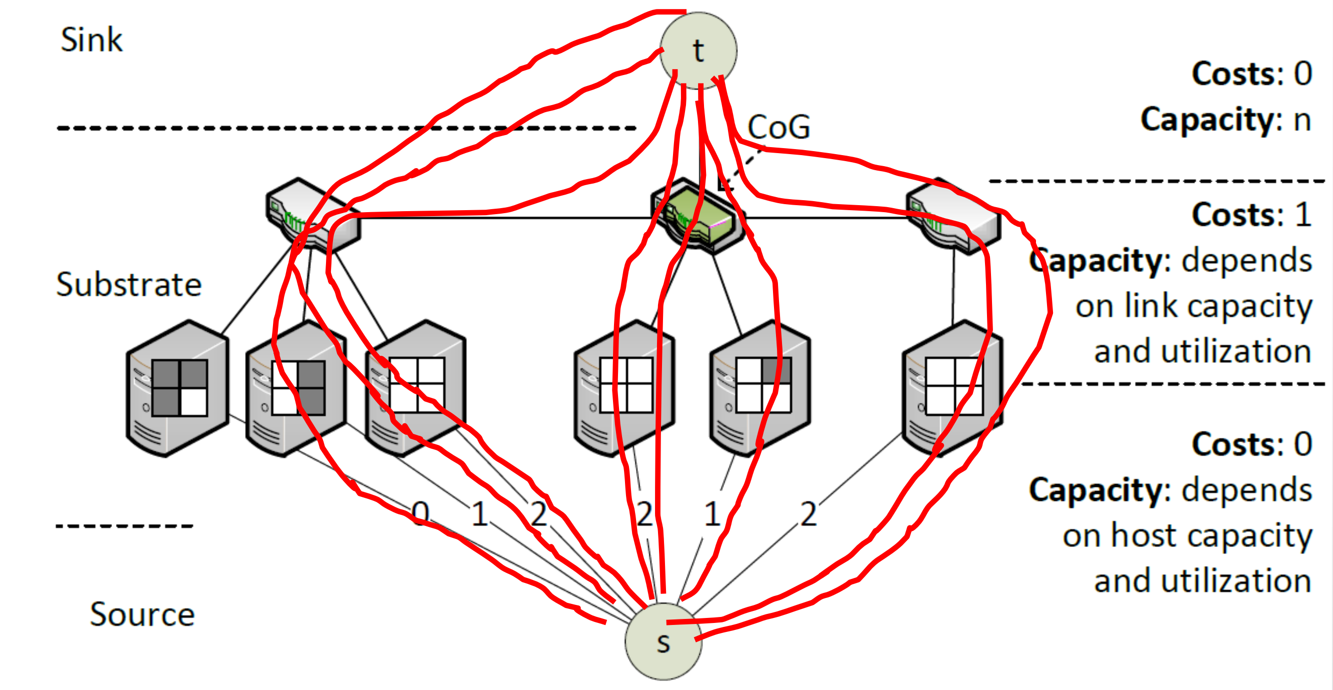
Efficient Embedding of Virtual Clusters: General Topology

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

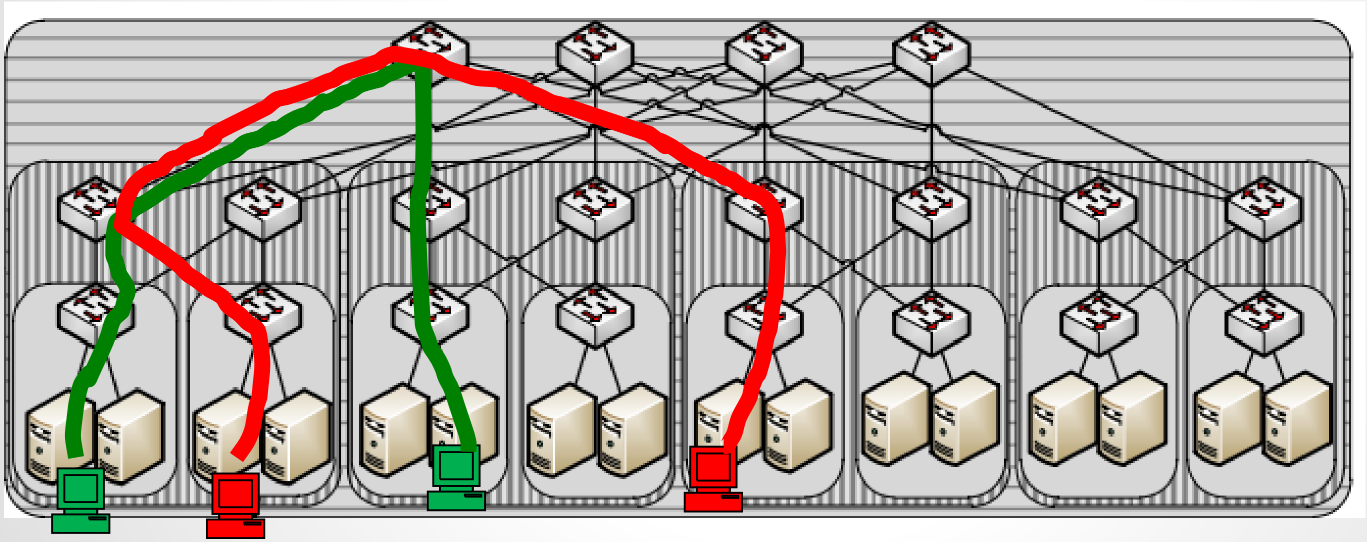
  (or simply: min-cost flow of volume n)

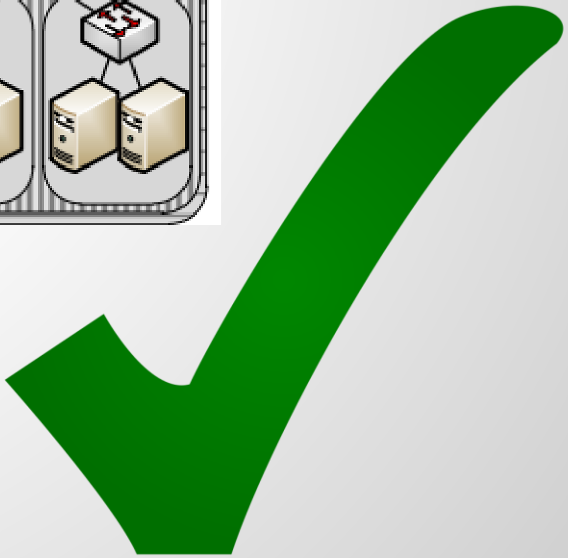




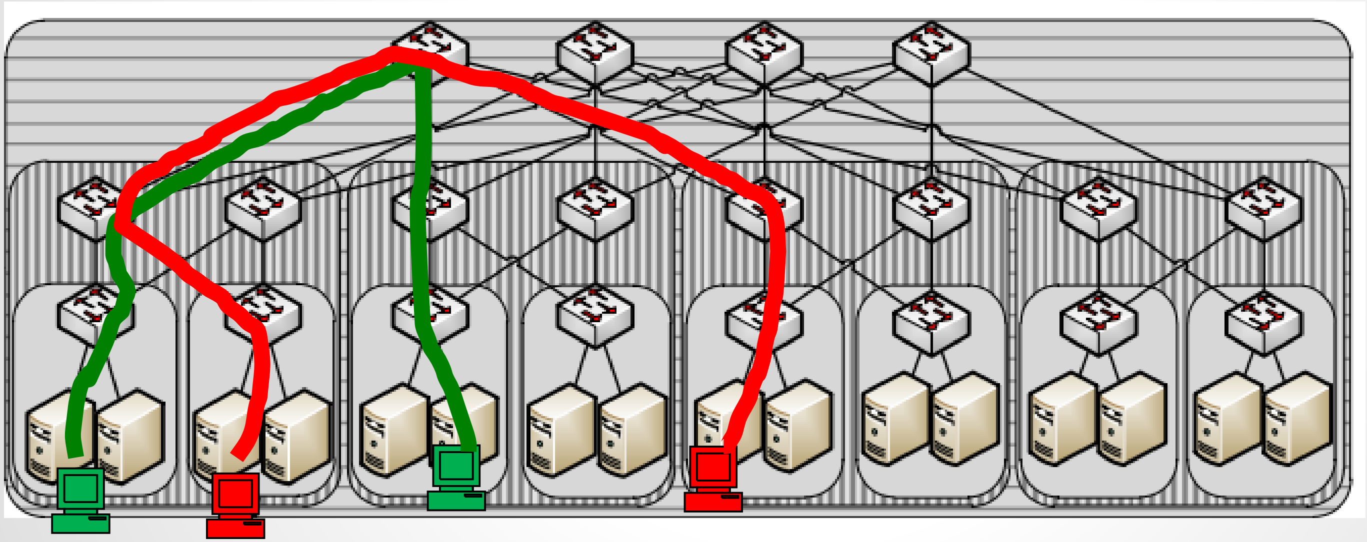
Guaranteed integer if links are integer! (E.g., successive shortest paths)

Long story short: By efficient virtual cluster embedding, Kraken solves unpredictable network performance problem!



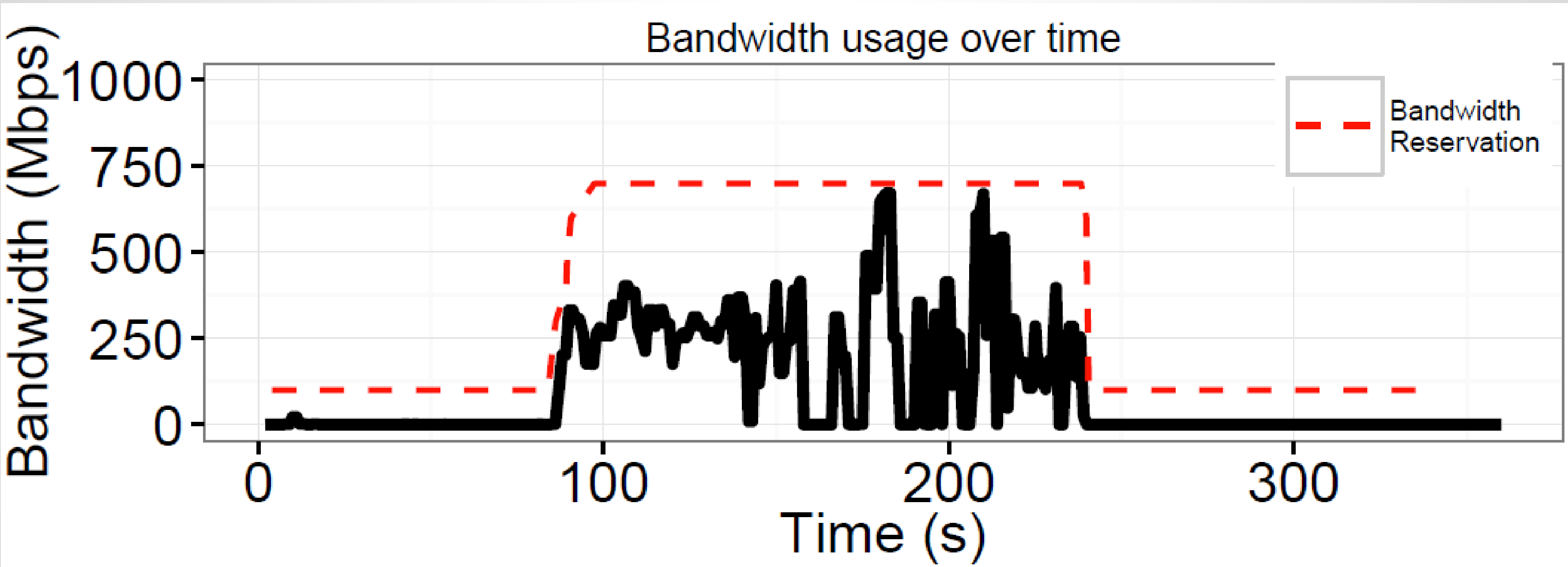


Long story short: **By efficient virtual cluster embedding, Kraken solves unpredictable network performance problem!**



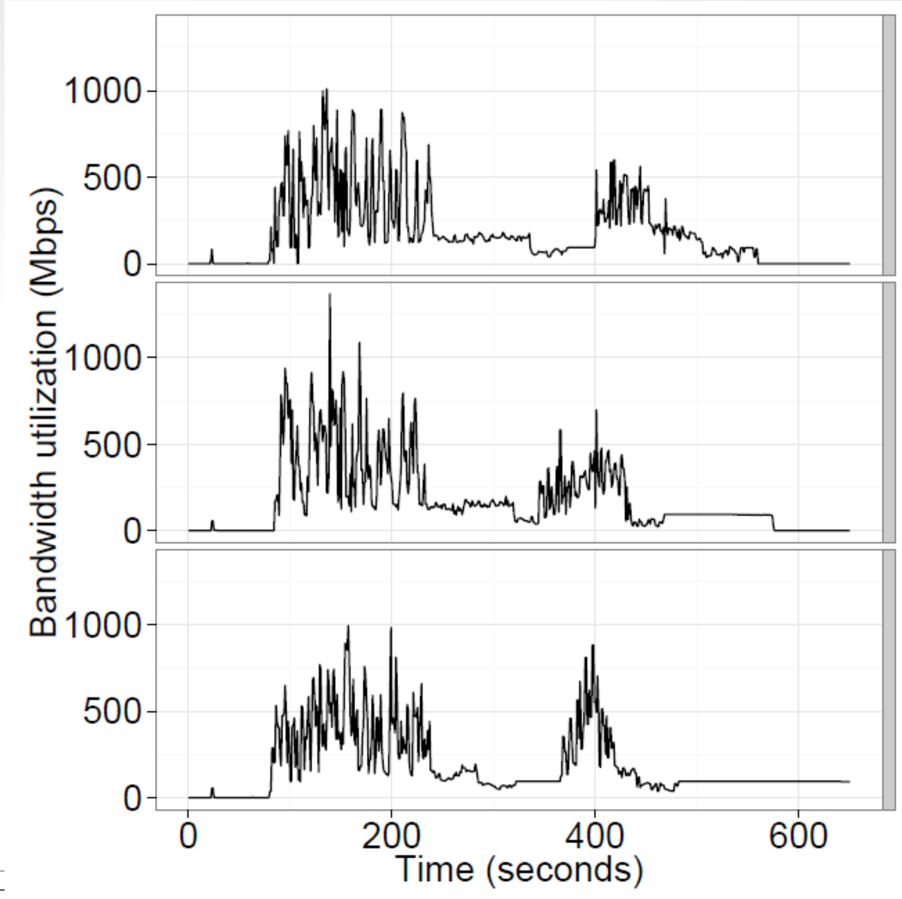
Problems solved!?

Not really: Resource needs change over time…

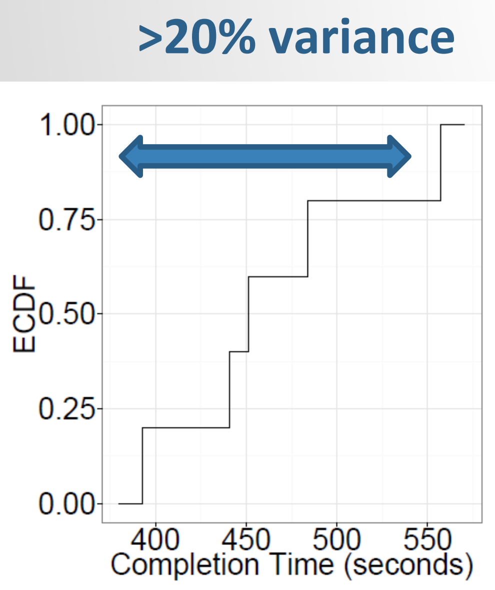


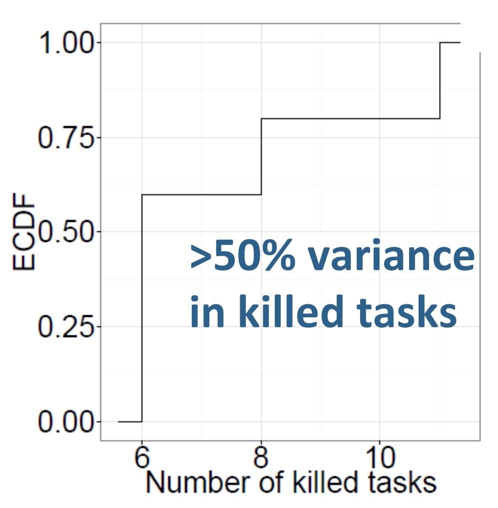
… often hard to predict!

- ❑ ***Temporal*** resource patterns are hard to predict
- ❑ Resource allocations must be changed ***online***



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





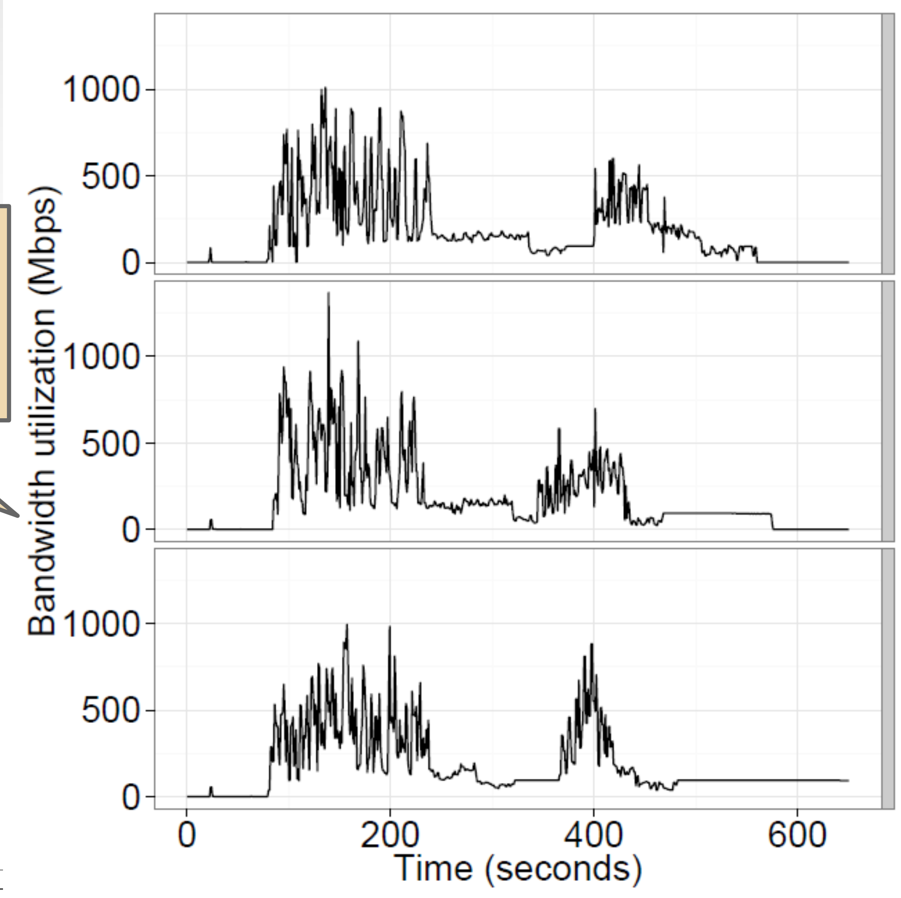
Completion times of jobs in the presence of **speculative execution** (*left*) and the number of speculated tasks (*right*)

… often hard to predict!

- ❑ ***Tempora*** hard to p

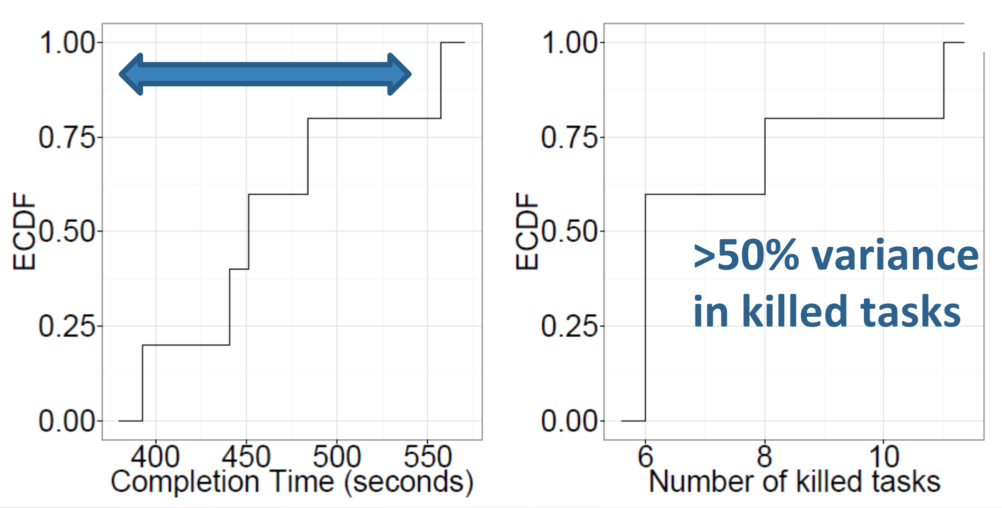
EC2 likely to be much more noisy!

- ❑ Resource allocations must be changed ***online***



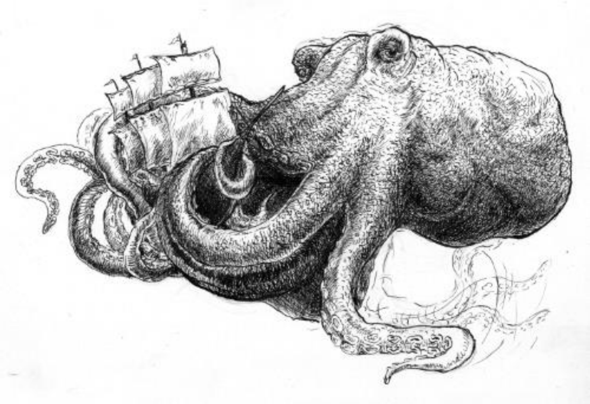
Bandwidth utilization of 3 different runs of the same **TeraSort workload** (**without interference**)

>20% variance



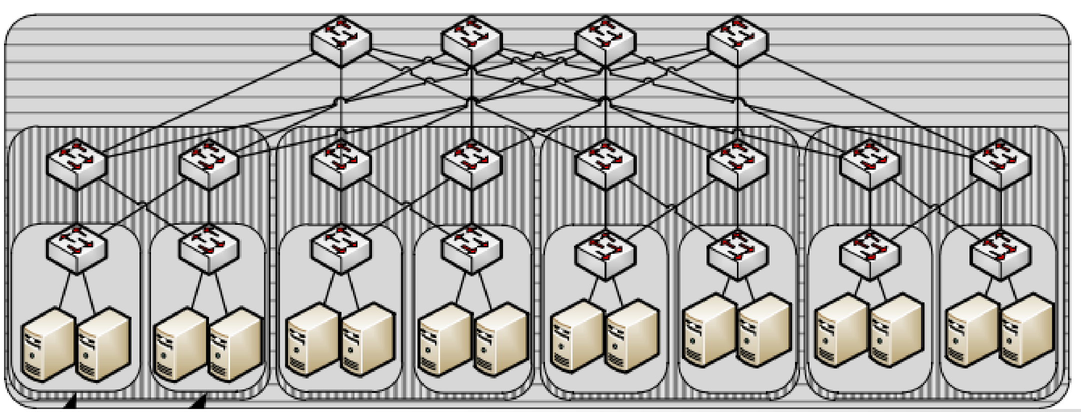
Completion times of jobs in the presence of **speculative execution** (*left*) and the number of speculated tasks (*right*)

Kraken: Elastic Reconfigurations



monitor
progress

unpredictable

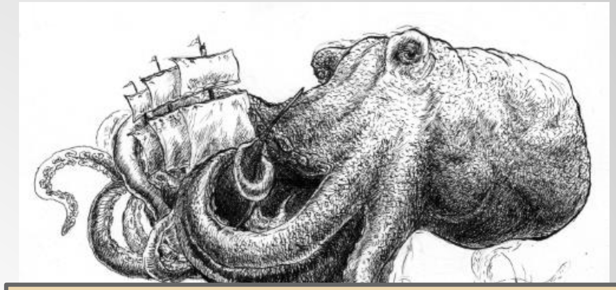


up/down BW
up/down CUs

interference

speculative
execution

Kraken: Elastic Reconfigurations

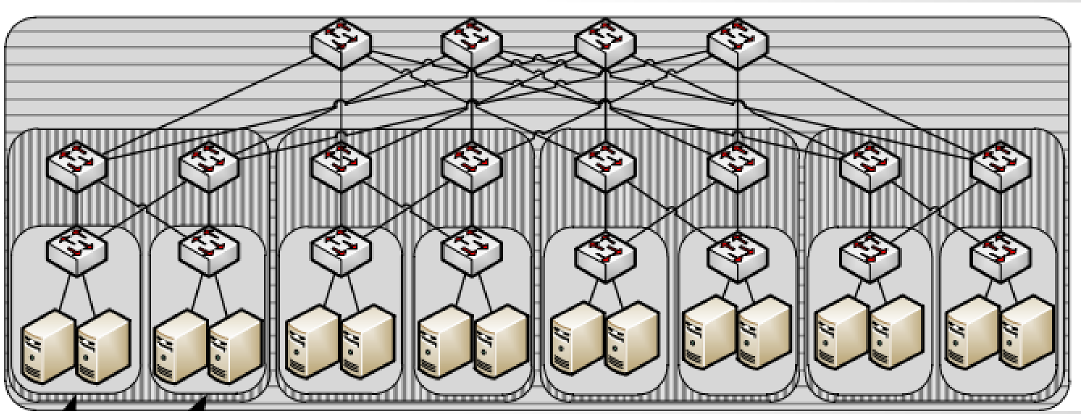


Rational: User knows requirements best.

Rational: Provider knows substrate best.

monitor progress

unpredictable



up/down BW
up/down CUs

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

Allows to adjust performance and compensate for stragglers and interference!

- Efficient virtu
- 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

Local migrations only: no need for re-embeddings!

al cluster embedding

Scale up a
cluster siz

Can also help to improve acceptance ratio!

Support task migrations

Provable performance guarantees:
Tradeoff embedding quality vs migration cost

Kraken: Elastic Reconfigurations

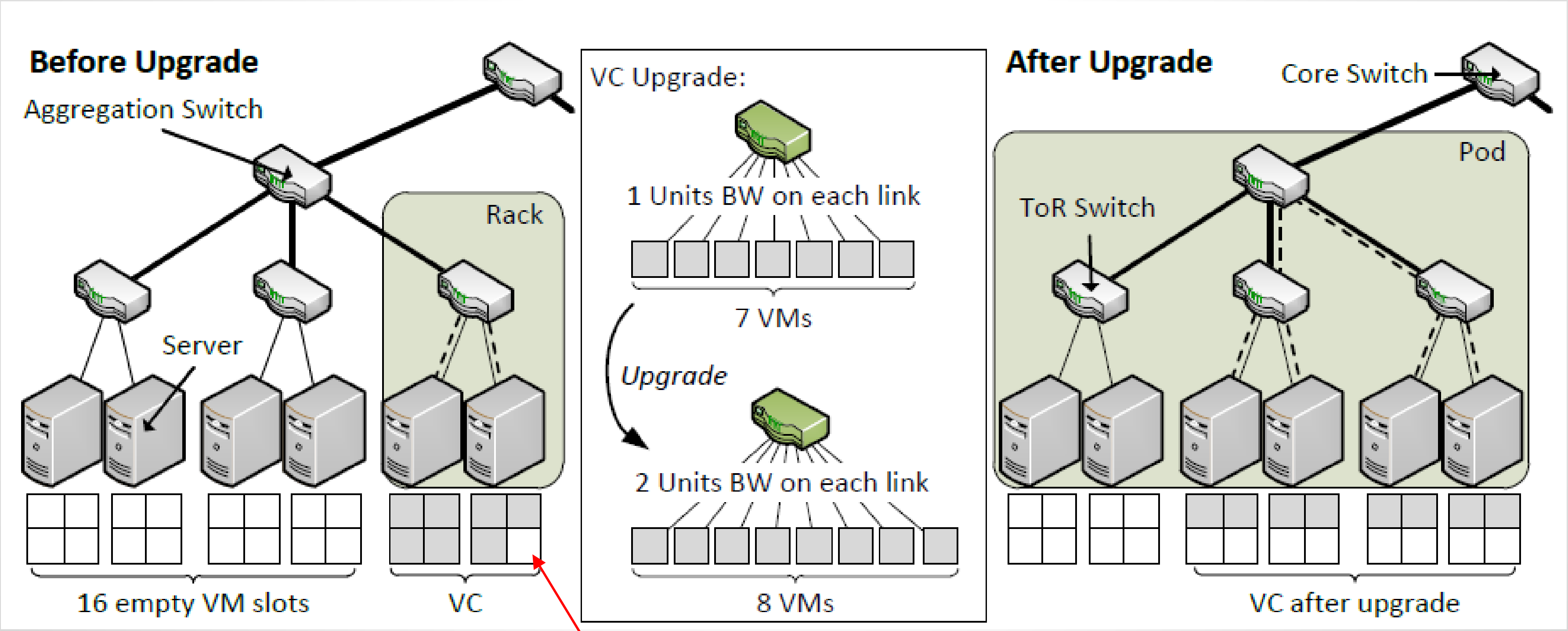
- ✓ Efficient virtual cluster embedding
- ✓ Scale up and down network resources and

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

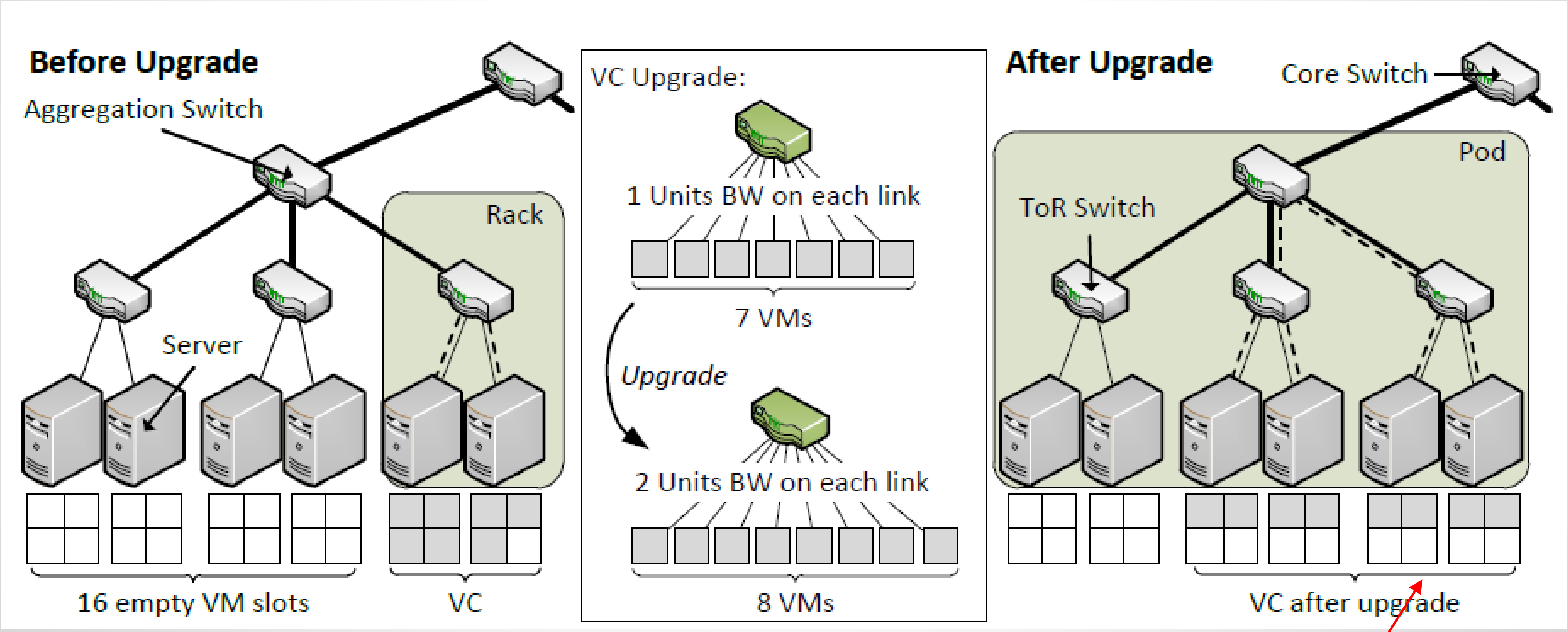


cannot add here:

bandwidth insufficient!

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

Kraken

Algorithm 1 Algorithm upgrade(VC,x,δ)

Output: success or *failure*

for all nodes v in the fat-tree: compute $slotCount(v)$ values
$m^* \leftarrow \infty$; $F^* \leftarrow \infty$; $cog^* \leftarrow \perp$;
for all v in substrate **do**
    $M \leftarrow$ minMig(v)
    if $|M| \leq m^*$ **then**
        $F \leftarrow$ footprint(v, $|M|$)
        if $F < \infty \wedge (|M| < m^* \vee F < F^*)$ **then**
            $cog^* \leftarrow v$
            $m^* \leftarrow |M|$
            $F^* \leftarrow F$
        end if
    end if
end for
if $m^* = \infty$ **then**
    return *failure*
end if
$\mu \leftarrow$ computeEmbedding(VC, cog^*)
return *success*

highest priority:
satisfy change request
by trying all options
(linear number)

Kraken

Algorithm 1 Algorithm upgrade(VC,x,δ)

Output: success or *failure*

for all nodes v in the fat-tree: compute $slotCount(v)$ values
$m^* \leftarrow \infty$; $F^* \leftarrow \infty$; $cog^* \leftarrow \perp$;
for all v in substrate **do**
    $M \leftarrow$ minMig(v)
    if $|M| \leq m^*$ **then**
        $F \leftarrow$ footprint(v, $|M|$)
        if $F < \infty \wedge (|M| < m^* \vee F < F^*)$ **then**
            $cog^* \leftarrow v$
            $m^* \leftarrow |M|$
            $F^* \leftarrow F$
        end if
    end if
end for
if $m^* = \infty$ **then**
    return *failure*
end if
$\mu \leftarrow$ computeEmbedding(VC, cog^*)
return *success*

compute minimal reconfiguation cost

(dynamically, and given current configuration)

Kraken

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

Output: success or *failure*

for all nodes v in the fat-tree: compute $slotCount(v)$ values
$m^* \leftarrow \infty$; $F^* \leftarrow \infty$; $cog^* \leftarrow \perp$;
for all v in substrate **do**
  $M \leftarrow$ `minMig`(v)
  if $|M| \leq m^*$ **then**
    $F \leftarrow$ `footprint`$(v, |M|)$
    if $F < \infty \wedge (|M| < m^* \vee F < F^*)$ **then**
      $cog^* \leftarrow v$
      $m^* \leftarrow |M|$
      $F^* \leftarrow F$
    end if
  end if
end for
if $m^* = \infty$ **then**
  return *failure*
end if
$\mu \leftarrow$ `computeEmbedding`(VC,cog^*)
return *success*

among all solutions with min migration costs, minimize footprint

Kraken

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

Output: success or *failure*

for all nodes v in the fat-tree: compute $slotCount(v)$ values
$m^* \leftarrow \infty$; $F^* \leftarrow \infty$; $cog^* \leftarrow \perp$;
for all v in substrate **do**
    $M \leftarrow$ `minMig`(v)
    if $|M| \leq m^*$ **then**
        $F \leftarrow$ `footprint`$(v, |M|)$
        if $F < \infty \wedge (|M| < m^* \vee F < F^*)$ **then**
            $cog^* \leftarrow v$
            $m^* \leftarrow |M|$
            $F^* \leftarrow F$
        end if
    end if
end for
if $m^* = \infty$ **then**
    return *failure*
end if
$\mu \leftarrow$ `computeEmbedding`(VC,cog^*) **dynamic program**
return *success*

Kraken

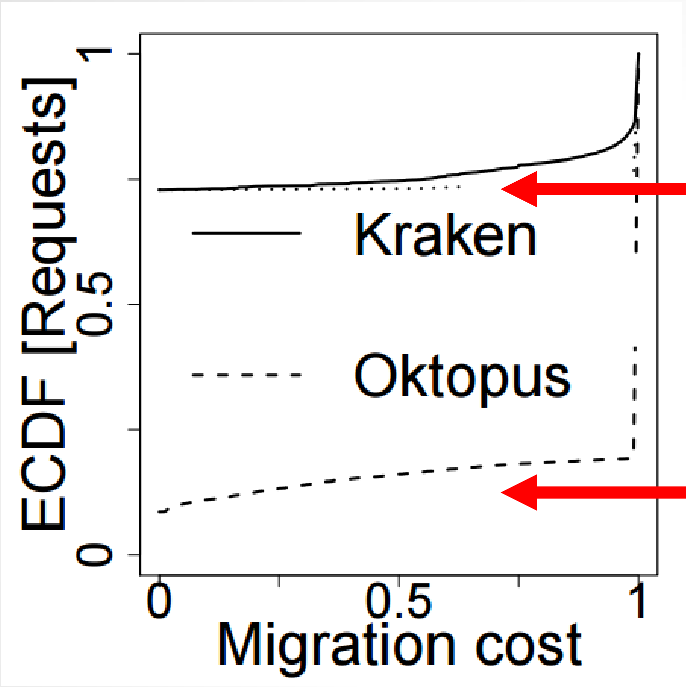
other orders possible

linear time

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

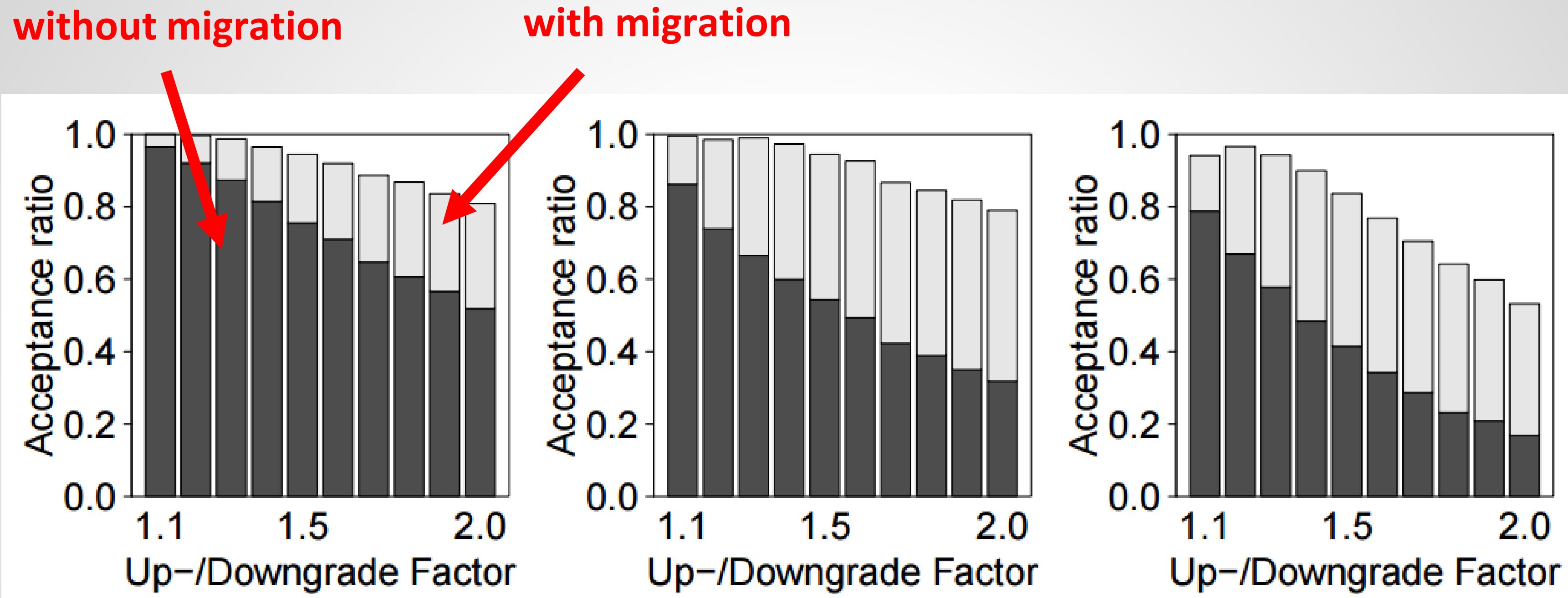


Kraken can satisfy most reconfiguration requests locally.

Oktpus: resort to rembedding costly.

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

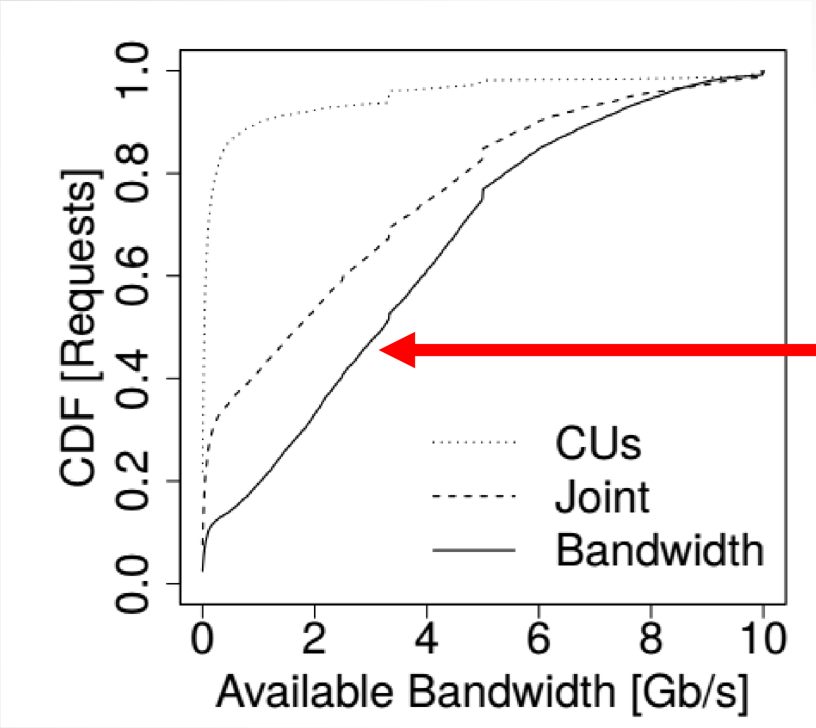
Benefit 2: Improved Acceptance Ratio with Reconfigurations



cluster size upgrade | bandwidth upgrade | joint upgrade

Migrations allow to accept additional requests!

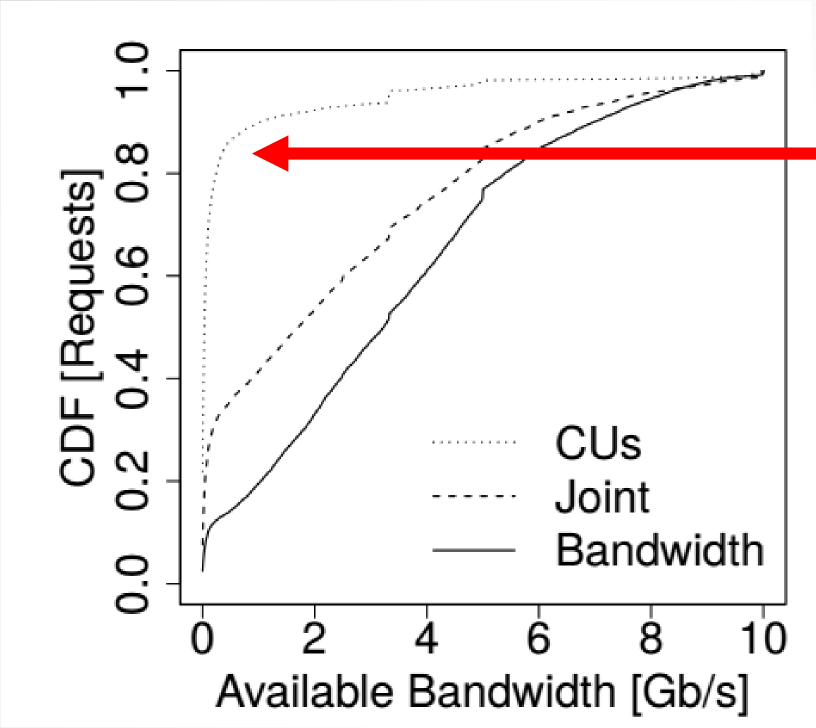
Migrations are Feasible: Available Bandwidth



Sufficient available bandwidth for flow reconfigurations / migrations.

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

Migrations are Feasible: Available Bandwidth



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

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

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

Questions?

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

Output: *success* or *failure*

for all nodes v in the fat-tree: compute $slotCount(v)$ values
$m^* \leftarrow \infty$; $F^* \leftarrow \infty$; $cog^* \leftarrow \perp$;
for all v in substrate **do**
  $M \leftarrow$ `minMig`(v)
  if $|M| \leq m^*$ **then**
    $F \leftarrow$ `footprint`(v,$|M|$)
    if $F < \infty \wedge (|M| < m^* \vee F < F^*)$ **then**
      $cog^* \leftarrow v$
      $m^* \leftarrow |M|$
      $F^* \leftarrow F$
    end if
  end if
end for
if $m^* = \infty$ **then**
  return *failure*
end if
$\mu \leftarrow$ `computeEmbedding`(VC,cog^*)
return *success*

Algorithm 2 `minMig`(substrate node v)

Output: set of CUs

$M \leftarrow \emptyset$
$L \leftarrow$ `computeConflictLinks`(v)
sort L with decreasing distance from v
for all links $\ell \in L$ **do**
  while ℓ oversubscribed **do**
    let c be an arbitrary CU below ℓ
    $M \leftarrow M \cup \{c\}$
  end while
end for
$M \leftarrow M \cup$ `extraCUs`(v)
return M

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

Output: cost value

$done \leftarrow 0$
for all children v' of v in the fat-tree **do**
  $done \leftarrow done + slotCount(v')$
end for
return `ST`(v) $+ height(v) \cdot n +$ `costsAbove`(v,$m - done$)