CloudTalk: Enabling Distributed Application Optimisations in Public Clouds

Alexandru Agache
University Politehnica of Bucharest

Mihai Ionescu
University Politehnica of Bucharest

Costin Raiciu
University Politehnica of Bucharest

Abstract

Clouds offer an opaque I/O API to their customers: details of the underlying resources (network topology, disk drives) or their current load are kept hidden. Tenants can profile the I/O performance in their VMs and optimise accordingly, but the side effect is increased load. Certain cloud providers try to discourage profiling by enforcing strict I/O isolation, at the cost of reduced utilisation in the average case. In this paper we challenge this status quo and propose CloudTalk, an API that allows tenants to communicate with the cloud provider and receive hints used to optimise their workloads.

We have built a distributed implementation of CloudTalk that scales to hundreds of machines and provides significant performance benefits in many cases. Further, we have implemented changes to Hadoop and HDFS that use CloudTalk to decide which machines to use for task placement and replica selection. Our experiments in a local cluster and on Amazon EC2 show that CloudTalk helps improve performance by as much as two times for a wide range of scenarios.

1. Introduction

Data center applications can be optimised by using network topology, link or server load knowledge to choose the best endpoints to place tasks [**??**], or to harmonize multiple computing frameworks running on the same cluster [**???**]. When sizes are known, one can approximate shortest-job first scheduling [**??**] for short flows, and load balance long flows across multiple paths to avoid collisions [**??**]. These optimisations and others have been designed for, and used in, private clouds where the data center owner runs both the apps and the infrastructure, bringing performance improvements up to orders of magnitude better.

EuroSys '17 April 23 - 26, 2017, Belgrade, Serbia

ISBN ACM 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064185

Unfortunately, some optimisations are almost impossible to implement in public clouds today. That is because clouds offer a rather opaque API to their customers: clients do not know the underlying network and storage topology and its current load. Providers have no insight into tenant workloads, and no control over the stack they use.

The standard techniques by which clients can glean infrastructure information are probing and topology discovery, but they both come with major issues. Probing is both costly and unreliable when performed independently by multiple tenants. Topology discovery only works for a handful of applications, and limits the cloud provider's flexibility in matching virtual machines to physical servers. Cloud users are seemingly stuck: optimising their distributed applications requires in-depth infrastructure knowledge and real-time I/O information that providers are understandably reluctant to give away. Cloud providers are not in great shape either: protocols such as Multipath TCP benefit long flows, and PFC (Priority Flow Control) [**?**] is great for short scatter-gather communication. The providers could enable them for their customers, but they don't know the traffic type.

In this paper we propose CloudTalk, a novel cloud-tenant API that enables application and infrastructure optimisations while preserving provider flexibility in VM placement. Tenant apps use CloudTalk to express the traffic pattern they wish to run, together with possible endpoints for the different flows. The cloud provider runs a CloudTalk server on each hypervisor that uses topology and live traffic information to answer client queries with near-optimal endpoint placement. Additionally, the CloudTalk server can configure lower-layer protocols such as PFC to optimise client traffic. The main contributions of our paper include:

- A measurement analysis of Amazon EC2 to check whether profiling is feasible and data center applications can be optimised using the derived information.
- The design of the CloudTalk language (see §4.1).
- A fast and fully distributed CloudTalk prototype that scales to hundreds of servers or more (§4).
- Implementing CloudTalk support in apps such as distributed filesystems (HDFS), and map/reduce (Hadoop).

We have run our solution both in a local cluster containing 20 servers, as well as on 100 Amazon EC2 servers. The results show that CloudTalk improves performance by 1.15-2 times for tested applications.

2. Motivation and requirements

Our goal is to enable joint app/network optimisation of tenant distributed applications running over public clouds. Tenants run apps without network information, while cloud providers run their networks without explicit information from apps regarding demands, and no control over tenant software, including network stacks.

Many possible optimisations have been proposed that do not work across the VM API, including [**?????????**] to name a few. To implement any of these we need cloud providers and tenants to share information. Consider the optimisation proposed by Chowdary et al. [**?**] to select the placement of HDFS replicas based on congestion information in the network. The experiments show performance improvements of up to 2x compared to the baseline. Implementing this in a public cloud is near impossible today: the tenant runs the distributed filesystem and has the ultimate say in where data resides, but the provider alone knows the load of its links.

Optimising reducer placement. Another example is the popular map/reduce paradigm. Where should the relatively few reducers be placed? Our experiments show that choosing an unloaded server can double performance.

Optimising web-search. We ran Apache Solr on 100 Amazon EC2 instances to implement web search on a snapshot of the .uk domain. By using topology information when placing aggregators, we can improve query response times by an order of magnitude (see §5.4).

At the other end of the cloud API, providers have few options to optimise their infrastructure without tenant support. As we show in §3, EC2 appears to have a fully provisioned core network. To fully utilise this capacity, there are a number of challenges to address.

Utilising full bisection networks is no easy task: hashing flows to random paths can lead to wasting 60% of capacity because of collisions [**?**]. The best solutions involve changing the end-host stacks to spread high-throughput elephant connections over multiple paths [**??**], but they adversely affect short flow completion times. If cloud providers *knew* which flows are elephants and would benefit from redirection, they could deploy optimised stacks in the hypervisor and proxy the traffic.

Enabling network features selectively. Even when bandwidth is plentiful, tenant apps might still suffer from peculiar traffic patterns, such as incast. The provider could enable PFC, a layer two mechanism that uses pause messages to prevent loss and completely eliminate incast-related problems. PFC cannot be enabled for all tenants, though, because it reduces throughput for elephant flows.

Requirements. We need a deployable mechanism that has the following properties:

- **Flexibility** to support a wide range of application or infrastructure optimisations.
- **Scalability** to large public clouds and many tenants.
- **Fast response times**: the new API should respond to queries as quickly as possible, to enable benefits for both batch and real-time applications.
- **Security**. The API should not enable tenants to launch DoS attacks against the cloud or other tenants.

3. Probing and optimizing in the cloud

Cloud providers such as Amazon EC2 or Microsoft Azure do not disclose any network topology details. Nevertheless, a great deal of information can still be inferred by running measurements on cloud instances. Is this sufficient to optimise tenant applications? We discuss active probing and its limitations in the context of Amazon EC2. There are two types of data that can be collected: topology information, which is relatively static, and performance-related data, such as the available network or local storage bandwidth.

3.1 Probing the EC2 network

We reverse-engineered the topology of Amazon EC2's us-east-1d data center by acquiring many instances and using well known network diagnosis tools such as ping, traceroute, and the iperf bandwidth measurement tool. To understand the evolution of the EC2 network, we reran measurements on the same data center periodically, starting from 2011 to October 2015.

Figure 1 shows the topology as inferred from our 2011 measurements. By using the traceroute number of hops and reported IP addresses, we were able to cluster VMs on the same physical host (1 hop in traceroute, via the hypervisor), and in the same rack (two hops via two hypervisors). To make sense of the topology at higher levels in the hierarchy we used `ping -R` (record route) to generate the possible paths between any two VMs. Next, we used IP aliasing [**?**] to figure out which addresses are on the same routers. Finally, we used iperf measurements to infer the (likely) maximum capacity of the links. While we cannot be absolutely sure that Figure 1 ever was 100% accurate, it correctly explains the behavior we noticed during our tests, as well as anecdotal information gathered from Amazon employees. The topology is similar to VL2 [**?**].

We also measured the bandwidth available to each VM and found it varies across instances, with extremes between 100Mbps and 1Gbps. It appears no shaping was performed, and all VMs on the same physical host shared its gigabit link. Other EC2 measurements around that time found similar performance characteristics [**??**].

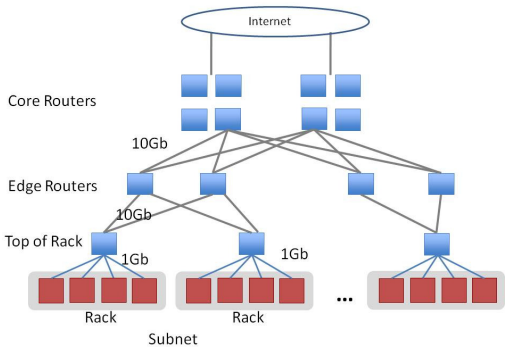


Figure 1: EC2 US-EAST Availability Zone D Topology as inferred by measurements in December 2011.

More recent measurements (2015) reveal an effort from Amazon to hide its network details, as follows:

- The IP aliasing techniques stopped functioning straight after we shared with Amazon a copy of our preliminary work in 2012.
- Ping with record route is no longer allowed.
- Traceroute reports addresses with no obvious structure. Most likely Amazon has moved to a tunneled architecture where tenant traffic is encapsulated and decapsulated in Dom0, similar to VL2. This allows EC2 to offer the illusion of a virtual LAN to all tenants by tunneling L2 traffic encapsulated at L3, as well as to hide its network configuration details.
- The throughput experienced by VMs is very stable regardless of time of day, and only depends on instance type (e.g. 500Mbps for c3.large).

However, we can still get information from traceroute and ping data. In particular, ping times are correlated with the number of traceroute hops, indicating whether two machines are on the same host, rack or subnet.

Optimising applications. The EC2 network was fairly open in 2011, and tenants could use both capacity probing and topology information to improve their apps; a number of works proposed to use profiling to optimise VM placement [**???**].

Unfortunately, capacity probing is inefficient when used at large scale. Continuous, frequent network and storage measurements constitute pure overhead from the cloud provider's viewpoint, and can negatively influence the performance of tenants not doing probing. Further, probing costs increase linearly with the number of tenants wishing to optimise their applications, and *will not scale to a large cloud*. Finally, probes from different tenants could overlap in time leading to *incorrect inferences* about the available capacity. While we can't know for sure, these considerations are likely to have played a role in Amazon's decision to perform strict isolation of bandwidth between tenants on most of its VMs, sacrificing aggregate utilisation in the process.

Topology information is very difficult to completely hide, and can be used to optimise delay-sensitive applications such as web-search, as we show in our evaluation (see §5.4). Static topology information is, however, insufficient to optimise apps like HDFS or Hadoop: in full-bisection networks there are no bottlenecks in the core, and all hotspots form at the server access links.

3.2 Towards a solution

Applications should collaborate with the cloud to reap mutual benefits: the status quo is suboptimal for all parties. The strawman solution is to have the data center operator reveal the real network topology, together with accurate load information. This violates our security requirement, and as such has no chance of adoption.

We could adopt ideas from IETF ALTO (application-layer traffic optimisation) Working Group [**?**]. ALTO aims to help network operators and endpoints communicate to enable provider-friendly peer-to-peer replica selection. ALTO servers run by the operator provide requesting applications with a network map and a cost map. The network map is a clustering of IP addresses performed by the operator according to its own routing policy, and the cost map provides routing costs between clusters. The applications use this information to select the most desirable peers. The ALTO model offers better confidentiality than the strawman, but it fails to capture many-to-one or many-to-many traffic patterns, and does not include dynamic load information.

Cloud providers will never disseminate load information to tenants for fear of compromising security. Instead, we need a mechanism that can guide apps to make the right scheduling decisions without giving them actual load measurements. For this system to work, apps must announce their intended operations via an API.

4. CloudTalk Architecture

CloudTalk enables optimisations by shifting to the provider a part of the client-side decision making process involved in distributed scheduling. A tenant does not ask specific questions about the environment, but rather describes a scenario where multiple options are available, and expects the cloud provider to suggest the best one. Consider the example in Figure 2 where virtual machine 1 wants to read file `f` that is replicated on VMs 2 and 3. The client application will formulate a query which describes this communication pattern to the cloud, and the reply will suggest the best replica to read from.

Queries are written in the CloudTalk language (§4.1) and answered using a distributed architecture. CloudTalk runs two services on every physical machine in the data center, as shown in Figure 2. The client facing *CloudTalk server* receives queries via local TCP connections from tenants,

processing them and returning an answer. The *status server* gathers information about disk and network interface usage and relays it to the CloudTalk server upon request.

After receiving and parsing a query, the CloudTalk server builds a reply which contains the best endpoint placement with respect to the problem instance. This has two phases: first, the CloudTalk server identifies all the VM IP addresses that appear in the query and locates the respective status servers, which are then interrogated (step (2) in Figure 2). UDP is used as transport, to minimize incast related problems. During step (3), the CloudTalk server aggregates all the status information pieces, by waiting for a predefined amount of time, or until all responses arrive. If nothing is received from a status server, we assume that a particular address is under heavy I/O load.

The CloudTalk server has now all the information needed to generate an answer. Next, it solves an optimisation problem where *possible answers* are *evaluated* to find the one that minimizes the client's job completion time. Trying out all possible answers becomes intractable when the client queries are moderately complex. We have designed a scalable heuristic algorithm that offers near-optimal results for many popular queries (see §4.2).

To estimate flow completion times, CloudTalk offers two options to its clients: a packet level simulator and a flow level estimator. The first is very accurate and captures packet-level effects such as incast, but it is also quite slow when many packets are involved, because it can accurately model transfers at the network and transport layers. It cannot capture effects caused by the operating system, or the applications themselves, but this usually does not lead to significant inaccuracies. The flow-level estimator arithmetically allocates a rate to each flow using the assumption that bottleneck links are shared equally (while also taking any restrictions into account), similar to [**?**]. The algorithm iteratively computes flow rates until they stabilize. It is accurate for large transfers and much faster than the packet level simulator, but doesn't work very well for short flows. When clients submit a query, they are also able to select which evaluation method should be used to determine the best placement.

Moreover, the client can specify whether the query should be evaluated while taking dynamic load information (data provided by status servers) into account. This is usually the case, but there are also situations where static information is sufficient. For example, Section 5.4 presents the optimisation of a web-search application deployment, which uses the packet-level simulator and static information to find out the best server placement, before the entire framework is started.

The CloudTalk architecture can, in principle, scale arbitrarily, but there are two possible practical limitations:

- When a query targets hundreds or thousands VMs, the CloudTalk server will ask all the associated status servers, resulting in a large overhead. We show, via analytical simulation and EC2 experiments, that it is enough to randomly ask a fixed subset of servers while still achieving near-optimal answers (§ 4.3).
- Is local monitoring of the I/O usage enough to ensure a globally-optimal answer? For full-bisection topologies, such as Amazon's, the answer is yes: the topology core is provisioned in such a way that bottlenecks can only appear at the end-hosts links.

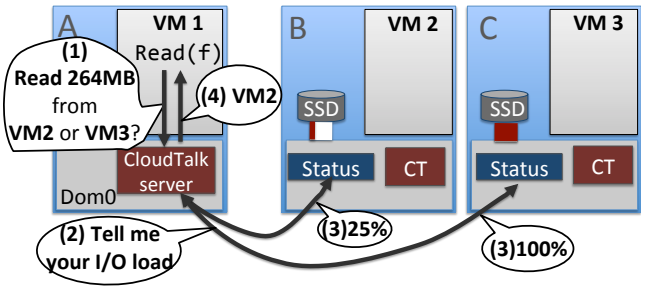


Figure 2: CloudTalk architecture: tenant apps query CloudTalk for the best way to perform their work.

4.1 Language

There are many dataflow programming languages already available, but none of those we considered present all the characteristics we desire for CloudTalk. Our language is specifically tailored towards describing flows generated during common I/O tasks, such as disk and network transfers. We employ a restricted but expressive set of flow properties, which allows us to model a wide range of application communication scenarios. This is enabled by the presence of restrictions and flow interdependencies, that can be expressed in a concise manner using attribute reference semantics.

Distributed applications rely on data flows—disk access and network transfers— to perform their work. An application task typically consists of multiple such flows, perhaps with dependencies among them, and the task finishes only when the last flow does. A CloudTalk query contains the representation of an entire set of flows, called a *problem instance*. In the CloudTalk language, each *flow* abstracts a data transfer. For network transfers, a flow represents an unidirectional transport connection, and each endpoint stands for a server (physical or virtual machine) which is identified by an IPv4 address. For disk transfers, a flow represents the process of reading or writing data to a local disk. Below we show a query for the example in Figure 2.

A = (vm2 vm3)
f_1 A -> vm_1 size $256M$

A CloudTalk query consists of one or more statements, separated by a semicolon when they appear on the same line. The exact syntax of the language is given in Table 1. Each flow has an optional name (f_1 in this case), a source, a destination as well as a description composed of static or dynamic attributes. The endpoints of a network connection can be literal IP addresses (which refer to a particular server such as

vm_1), or variables. In the example above, the variable A can take as values the IP addresses vm_2 or vm_3. We employ the convention of using lower case for literals and upper case for variables. There is also the special *disk* keyword, that can be used as endpoint, and identifies flows which describe disk I/O instead of network transfers. It represents local disk access, assuming each VM has a single local disk.

Variables embody the actual client request: each one has an associated set of values (IP addresses), and signifies that a particular endpoint can be placed on any of those servers. The CloudTalk server will respond with a binding of variables to values that represents the best recommendation in terms of endpoint placement.

Flow size is a static attribute and is set to 256M in our example. A static attribute is one that never changes over the lifetime of a flow, once its value is set or becomes apparent. This category contains the start and end times, and the size of a flow (expressed in bytes). A static attribute might remain unknown (for example, an endless flow has no size), but if it can be determined at some point, it will keep that value forever. A typical example is the end time, which is not usually known in advance, but will be set as the flow finishes. A dynamic attribute can change its value during a flow's lifetime. There are two such attributes: the instantaneous transmission *rate* and *transfer*, a monotonously increasing function that tells how much data has been successfully transmitted up to the respective point in time.

Attributes can specify a wide range of flow properties when restricted. If a flow description does not mention one attribute, then it is unrestricted. Restrictions depend on each particular attribute: for start time and size, restrictions are used as the attribute's value, i.e. the flow will start at that particular time. If there is nothing said about this attribute, then the flow is assumed to start right away (as in our example above). A restriction on size determines how much data can be sent until the flow ends. If the size attribute is not restricted in any way, and neither is the end time, the flow is assumed infinite (and will not count towards overall termination). Consider below a refinement of our example where we take the disk into account too, and capture the fact that the flow may be bottlenecked by the disk or the network:

A = (vm_1 vm_2 ... vm_n)
f_1 disk -> A size $100M$ rate r(f_2)
f_2 A -> vm_1 size sz(f_1) rate r(f_1)

There are n possible data sources and server vm_1 is the reader. We split the communication pattern in two flows, both of which start immediately: flow f_1 reads data from disk and f_2 sends it over the network.

A flow depends on another flow if one or more of its attributes are restricted by an expression that contains references to the other flow's attributes. In our example, flows f_1 and f_2 are interdependent, because they must have roughly the same rate: data cannot be sent faster that it is read from disk, and it will not be read at a higher speed than the network send rate. The presence of a buffer at the sender can be modeled with transfer restrictions, but this is more concise and paints a similar picture in most cases. A rate restriction means that a flow can never transmit faster than the specified value. Even if the rate is not explicitly constrained, it is still limited by network conditions. Our two restrictions mandate that the rates of the two flows will be the same. The references that can be used to establish flow dependencies are represented by the REF placeholder from Table 1. Reference can be made to all five flow attributes: *st* (start time), *e* (end time), *sz* (flow size), *r* (instantaneous rate), and *t* (amount of data transferred so far).

Variable	A variable has a set of possible values (IP addresses or disk), and the job of the CloudTalk server is to bind each variable to the value that minimizes task completion time.
var = (val_1 val_2 $\cdots$ val_n)	
Flow	Defines a flow between two endpoints. The flow has an optional name, a source, destination and a description.
[$name$] $src->$ dst $description$	
Description	Defines a flow's attributes.
[start val] [end val] [size val] [rate val] [transfer val]	
Values	In the context of a flow description, values can be numeric literals, references to an attribute of another flow (specified by name or identifier) or the result of an operation which involves other values.
val := LITERAL | REF | val OP val | (VAL)	
REF := st(f) | e(f) | sz(f) | r(f) | t(f)	
OP := + | - | * | /	
Attributes	*start* and *end* are given in seconds relative to current time. *size* and *transfer* are given in bytes. *rate* is given in Bps and specifies the maximum instantaneous rate for this flow.

Table 1: The CloudTalk language elements.

Now consider a pattern known as daisy-chaining. It is used, for example, when writing data to an HDFS file system. We have an initial sender (server a) and three other servers (B, C and D). Server a sends some data to B, which is stored locally but also forwarded to C. In turn, C does the same thing to D, which is just a receiver and doesn't send anything further. We assume that local storage is much faster than the network, so the disk related operations are not described in the query.

B = C = D = (s_1 s_2 ... s_n)
f_1 a -> B size $100M$
f_2 B -> C size sz(f_1) transfer t(f_1)
f_3 C -> D size sz(f_1) transfer t(f_2)

We could use the rate attribute to specify the dependencies between these flows, but it would miss local buffering effects: server A might send to B at a high rate initially, but then its rate could drop due to congestion. At the same time, B could start by slowly forwarding data towards C, and then ramp up to line rate when its link frees up. We use the *transfer* attribute in this case, which restricts the number of transferred bytes to always be lower than, or equal to the specified threshold, regardless of the instantaneous flow rate.

It is noteworthy that B, C and D will be bound by CloudTalk to different servers despite sharing the same pool of possible values. This is the default behaviour, because many of the multi-variable use cases we encountered required identifying different endpoints for replica placement, but can be overridden by the client when necessary.

4.2 Query evaluation algorithm

A CloudTalk query contains both variable declarations and flow definitions, which define the entire set of possible task placement scenarios. How can we find the best placement? It is easy to prove that query evaluation is NP-hard in the general case, but we omit the proof for space reasons.

To answer CloudTalk queries we could evaluate all possible bindings and select the best result, but the solution space is often intractably large. One heuristic that works very well in practice is to simply pick the n-best servers for each query, where n is the number of variables. This heuristic proved to work very well for the use cases we considered, so it became the query evaluation method of choice for all our experiments, except for the web search optimisation presented in Section **??** (which uses the packet-level simulator, together with static information).

The algorithm examines the type of operation each variable is involved in (for example, sending data over the network), and picks the server whose I/O availability is best suited for that scenario. Listing 1 contains the pseudo-code for the relevant parts of the algorithm. Execution begins with *evaluateQuery*: every variable has two associated sets, `to` and `from`, where we record the set of endpoints that send to, or receive data from it. By removing the *disk* endpoint (if present) from these, we build the network-only `tx` and `rx` sets. Does the order in which we bind variables matter? The following scenario sheds some light onto this issue:

X = Y = Z = (a b c)
f_1 X -> Y size $100M$
f_2 Z -> a size $100M$

Here, any optimal value selection must include the binding $Z \leftarrow a$, because this way f_2 runs locally on a server and doesn't use network resources at all. If variables are bound to values in a random order, we risk using a for a different endpoint, which prevents the aforementioned optimal binding. At the same time, if disk I/O were used, the choice is not obvious; server a's local disk could be so slow that it is better to read data over the network instead. Here, f_1 and f_2 have the same size; for the task to finish quickly both flows must be placed on a high capacity link.

Our heuristic always assumes all flows are equally important, so it is highly desirable to bind a variable to an endpoint that it directly communicates with. This applies only when the variable communicates with at most one endpoint, and that endpoint is one of its possible values. In the pseudocode, line 8 tests this condition for data reception, while line 9 does it for transmission. If at least one condition is met, we

```
1  evaluateQuery
2    for f in flows
3      to[f.dst].insert(f.src)
4      from[f.src].insert(f.dst)
5    for v in vars
6      tx[v] = to[v] \ {'disk'}
7      rx[v] = from[v] \ {'disk'}
8      if size(rx[v]) == 1 && rx[v] ⊆ values[v]
9      || size(tx[v]) == 1 && tx[v] ⊆ values[v]
10       assignValue(v, values(v) \ used_values)
11       vars.remove(v)
12   for v in vars
13     assignValue(v, values(v) \ used_values)
14
15 assignValue(var, possible_values)
16   bestScore = 0
17   for v in possible_values
18     score = min(netRx(var, v), netTx(var, v),
19             diskRead(var, v), diskWrite(var, v))
20     if(score >= bestScore)
21       best = v
22       bestScore = score
23   bind(var, best)
24   used_values.insert(var)
25
26 netRx(var, ip)
27   if to[var] == {ip} || from[var] == {ip}
28     return MAX
29   if size(rx[var]) == 0
30     return MAX
31   return evalRx(ip)
32
33 diskRead(var, ip)
34   if 'disk' not in to[var]
35     return MAX
36   return evalDiskRead(ip)
37
38 evalRx(ip)
39   d = data[ip]
40   return d.net.rxCap - d.net.rxUse / W
```

Listing 1: Query evaluation algorithm pseudocode.

prioritize the variable for value assignment. Using priorities ensures that we can still check for disk bottlenecks and value availability. After high priority variables are bound, the assignment method is applied to the remaining ones.

The *assignValues* function is responsible of picking a value for a given variable, out of a predefined set of possibilities. A score is assigned to each candidate value, and the best is selected at the end. The overall score represents the least available resource used by the flow which involves that particular variable. The individual score calculation procedures are quite similar, so we only present the *netRx* and *diskRead* functions. The former estimates the network receive fitness of an endpoint, with respect to the given variable. On line 27 we check if the previously described single local endpoint condition is met, and return a maximum score if that is the case. The same score value is used when the variable is not involved in any network traffic reception (line 29). If neither of the previous conditions are met, the score is evaluated based on available status information (line 31). Things are similar for *netTx*, *diskRead*, and *diskWrite*, but the last two no longer include any check for a local endpoint.

All the status information evaluation functions are similar; the data parameter is the only difference. The *evalRx* method only cares about network receive capacity and network receive usage. The $data[ip]$ element contains informa-

tion received by the CloudTalk server from that particular machine. In its simplest form, the result of *evalRx* is the difference between maximum capacity and usage. However, there is also the selectable weight W (implicitly 2), which can be used to change the relative importance of maximum resource capacity versus contention.

In order to get a better picture of how the heuristic operates, let us return to the previous example. There are three variables (X, Y, and Z), and three common possible values (a, b, and c). We have `tx[X]` = $\emptyset$, `rx[X]`={`Y`}, `tx[Y]`={`X`}, `rx[Y]` = $\emptyset$, `tx[Z]` = $\emptyset$, and `rx[Z]`={`a`}. Z is the only variable that fulfills at least one of the conditions from lines 8 and 9, so it will be assigned a value first. The score of a (for Z) is equal to the maximum value, for both *tx* and *rx*. Moreover, disk operations and network receive status are not relevant for Z, so the overall score will be the maximum value, meaning Z will be bound to a. The other two variables, X and Y, will be bound last, in that order. For X, the heuristic will pick from b and c the endpoint with the most outgoing capacity, and Y will be bound to the remaining server.

The execution of the heuristic takes place after all endpoint related information has been gathered from the status servers. Let m be the number of flows, n the number of variables, and p the maximum number of possible values for any given variable. All score related calculations run in asymptotically constant time, so the *assignValue* function finishes in $O(p)$ steps. In *evaluateQuery*, lines 3 and 4 are executed m times, and *assignValue* is called n times, so the entire algorithm terminates after $O(\max(m,\ np))$ steps.

This heuristic can be seen as a hybrid of classical first-fit and best-fit strategies. In our context, a pure first-fit approach would not work well, because a variable can be bound to any of its possible values (it "fits" anywhere), which amounts to random placement. At the same time, the heuristic is not a best-fit either, as this would entail binding variables to endpoints that can both accommodate their communication requirements, and overshoot the necessary capacity by as little as possible. We did not fully pursue this strategy because it increases computational complexity; chained variables can't be assigned in isolation anymore, as the strongest restriction propagates via dependencies. A significant disadvantage of the previous heuristic is the loss of time as a dimension. It ignores all start restrictions, and does not consider that some flows finish sooner than others. The solution is to add separate heuristics for different classes of queries, and allow the user to select which one is appropriate.

4.3 Using sampling to improve scalability

Tenants may have large collections of virtual machines, perhaps thousands in some extreme cases. When such a tenant issues a query, CloudTalk may need to collect status data from thousands of status servers. The scatter-gather communication pattern used to retrieve load information risks losing replies for a large number of queried servers. Our experiments show that querying one hundred servers gives low packet loss with our UDP-based solution, while for a thousand servers, there is high packet loss. Dividing the requests into multiple rounds leads to increased query times and inaccurate information, so it is not desirable either.

To avoid these issues, CloudTalk uses sampling when N, the total number of tenant VMs, is larger than one hundred. CloudTalk only asks n randomly selected servers where $n << N$. How good is the answer computed via sampling, compared to having full knowledge? Our evaluation in § 5.2 shows via simulation and EC2 deployment that we only need to sample relatively few machines for near-optimal results: the number of samples needed depends on network load and the required number of servers d, but does not depend on N. In many situations, sampling achieves near-optimal results irrespective of the number of VMs owned by tenants.

5. Evaluation

We have implemented CloudTalk in approximately 3000 lines of C++. We rely on the flex/bison suite to generate a lexer and parser for the CloudTalk language. We have selected three representative application classes that can benefit from CloudTalk: MapReduce computation (Hadoop), a distributed filesystem (HDFS) and web-search (Solr). We have changed these applications to generate CloudTalk queries and use the replies whenever they have a choice (100-300 LOC per app).

We begin by examining the performance of the CloudTalk server implementation and its scalability. The rest of our evaluation is dedicated to measuring HDFS, Hadoop and Solr performance with and without CloudTalk optimisations. The results show that CloudTalk boosts application performance by 15% to 100%.

We rely both on a local cluster of twenty machines and Amazon EC2 for testing. Our EC2 experiments use a slightly modified setup to that in § 4: instead of running the CloudTalk and status servers in the hypervisor, we run them as processes inside our virtual machine. The status server estimates the remaining available capacity by subtracting NIC current usage from the per tenant bandwidth limits EC2 imposes these days (500Mbps for our VM type). In summary, we use EC2 to emulate a CloudTalk deployment at scale not available in our cluster, with the caveat that the network speeds are fairly low compared to real data center networks.

5.1 CloudTalk server evaluation

Our heuristic query evaluation algorithm is designed to be fast, but how much accuracy do we lose by favouring quick response times? It can be shown that our algorithm is optimal for single variable queries, and for daisy-chaining queries where the first endpoint is a fixed address. Surprisingly, these simple traffic patterns cover many use cases, but it's easy to find a query where our algorithm is suboptimal.

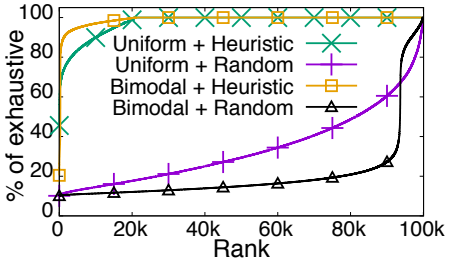


Figure 3: How close is our query evaluation algorithm to optimal?

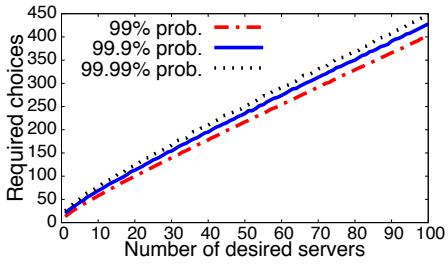


Figure 4: Evaluating the accuracy of distributed sampling.

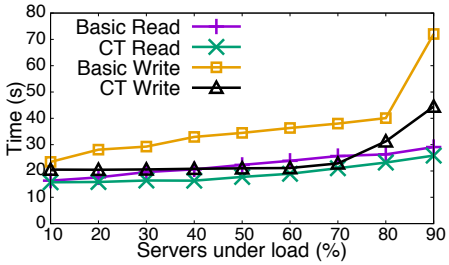


Figure 5: HDFS running over SSDs.

A simple example is daisy-chaining with every endpoint being a variable: $x_1 = x_2 = x_3 = (s_1 \dots s_n)$; $f_1\ x_1 > x_2$ size $100M$; $x_2 > x_3$ size $sz(f_1)$ transferred $t(f_1)$. There are three variables with n possible values here. We contrast the results of our algorithm against an exhaustive evaluation of all possible solutions. The comparison is made for 100k artificially generated network states involving 20 servers. Every state reflects network interfaces of the same capacity, but with variable outgoing and incoming transfer rates which vary between 0 and 90% of link capacity, and are independently selected for each direction. The background traffic is inelastic, meaning that flows described in the query can only use remaining capacity. We ran one batch of experiments where the rates follow a uniform distribution, and then another where they follow a bimodal distribution, with peaks at 0% and 90% utilisation.

First, we measured the query response time at the server, finding that it takes CloudTalk around 0.45ms on average to answer one query: of these, 0.32ms are spent in parsing the query and 0.13ms running our query evaluation algorithm. In comparison, the brute-force evaluation algorithm takes 130ms on the same query.

To measure how good the server selections are, we invoke the flow-level estimator to get an estimate of write throughput. We compare our solution and random server choice against exhaustive search, and plot the results in Figure 3. The vertical axis shows the achieved throughput as percentage of optimal, and the horizontal axis shows the number of experiments where the throughput was at least that much. The results show our heuristic is within 2x of the optimal in the worst case, and is optimal in 80% of the experiments. As expected, picking a binding at random proves to be a much worse solution. However, it's important to consider that random selection is generally the only possible option when no information is available, and our experiments show there is considerable room for improvement.

Our heuristic evaluation algorithm presented has a running time proportional to both the number of servers being considered (n) and the number of variables (d): Table 2 contains the average running time (in microseconds) of multiple heuristic evaluations for different values of n and d. It grows linearly with n, and is reasonably small when only a couple

n \ d	3	5	10	20	30
100	231	310	477	886	1307
200	427	550	878	1524	2209
300	607	782	1237	2163	3196
500	987	1245	1963	3420	4918
1000	2014	2586	4065	7146	10244
2000	3910	5057	7923	13674	19379

Table 2: Heuristic evaluator running times (μs)

hundred servers or less are involved. It actually turns out that parsing the query takes slightly more time than the heuristic evaluation. However, the parser is not optimised (a first step would be to write a custom parser, instead of using a tool-generated one), and the language itself could use constructs which help reduce the verbosity of some queries.

5.2 Sampling accuracy

A number of measurement works [**??**] show that in many data centers, at any given time, there is a large number of idle links, but there are some links that are hot and should be avoided. In the context of a full-bisection network, links could be split in two categories: low utilisation and high utilisation (near 100% utilised).

Query evaluation only cares for the top ranking servers in terms of available capacity. If we need d servers to answer the query, our goal is to select all d servers from the ones with idle uplinks. For concreteness, let's assume an average network utilisation of 70%.

Assume we want to contact n servers. How large must n be to ensure that the best d of them are in the 30% idle target group with some predefined probability, say 99%? We turn to simulation to answer this question and ran the following experiment: we generate a total of $N = 100.000$ servers with load 0% (30% probability) or 100% (70% probability). Then, we chose d, the desired number of servers for our query, and keep increasing the number n of sampled servers until all d servers were idle, with high probability (the confidence threshold). In Figure 4, we plot the dependence between d (on the X axis) and n (on the Y axis), for three different confidence thresholds.

First, consider the practical case where d, the number of servers needed, is small (between one to five in most of our optimisations): it suffices to query only 10-25 servers out of the total 100.000 to ensure the evaluation is optimal. n grows sub-linearly with d: one needs to enquire 4 servers for every server needed in an answer. With our current experience, this implies we can answer optimally all queries that involve 25 servers or less, even in a 100.000 node deployment.

Results depend on the fraction of idle servers: if 70% of servers are idle, we only need to ask 1.6 servers for each server we use; if only 10% of are idle, we need to ask as many as 20 servers per used server. In practice, we expect average network utilisation to be fairly low (30%-50% at most), thus our sampling solution can efficiently answer complex queries.

It's worth noting that server load does not necessarily have to follow a bimodal distribution, in order for sampling to make sense. In the general case, we can assume each server has a different load level. However, they can still be seen as part of a sequence which is sorted according to the fitness of each element. At this point, we only have to choose a threshold such that every server above it is considered desirable, while all others are not. This leads to a situation which is fundamentally similar to only having fully loaded or idle servers. Providers have access to actual load information, and can adjust the target interval for sampling accordingly.

Amazon validation. We also wanted to validate these results on Amazon EC2. The use case under consideration is writing a file to an HDFS cluster which consists of 301 machines. With the default replication factor of 3, one replica will be placed locally (as there is no storage bottleneck), and two more on remote servers. This means that we are looking to select two servers from the other 300 in the cluster. To make this choice actually matter, we make 70% of the servers (the writer not included) transfer data among themselves, using the iperf bandwidth measurement tool as a long running traffic generator, at line rate. Poor replica placement will dramatically increase the transfer time in this scenario.

If CloudTalk is not enabled, the average write time increases to 40s, from under 4s in an idle cluster. We enabled CloudTalk and limited the number of remote servers being interrogated each time to 19 (this is the number predicted in our previous analysis for a target interval of 30% with 99% confidence). After running transfers for a couple of hours, out of 2675 measurements, we got the following results: 2649 finished in under 4 seconds, 3 more finished in under 6 seconds, and the rest in under 30s. The number of unfortunate choices is less than the 1% predicted by the theory.

5.3 Hadoop evaluation

Our modified version of Hadoop was deployed both on a small local cluster consisting of 20 machines, and on 101 Amazon EC2 c3.large instances. The local computers have access to both gigabit and 10Gbps connections, that go directly into a switch. The EC2 c3.large instances offer transfer rates of around 500Mbps. Their local storage was considerably faster, so bottlenecks were always network related. In the local cluster, the 10Gbps interconnect can be used to overwhelm any of our disks.

HDFS Read. Inside HDFS, files are split into blocks and then replicated; the usual replication factor is three. Before reading any data, a client will request the location of the current block, and receive a list of machines hosting the replicas in return. It then uses CloudTalk to inquire which server is most suitable to read from:

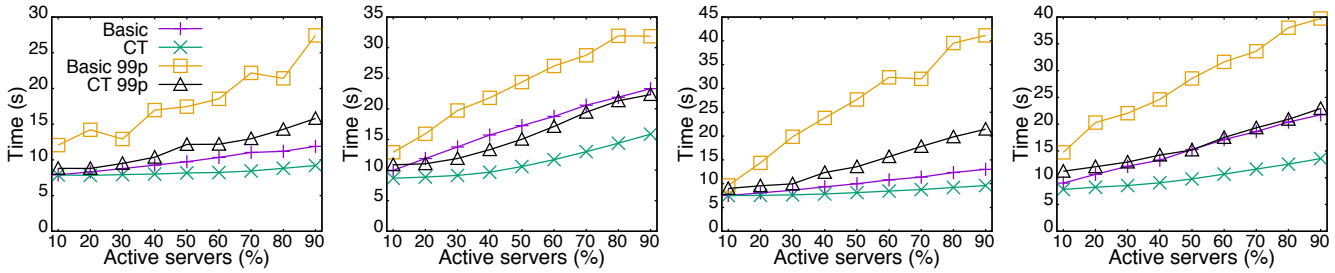
$$
\begin{aligned}
&src \texttt{ = (}replica_1\ replica_2\ \texttt{...}\ replica_n\texttt{)}\\
&f_1 \texttt{ disk -> } src \texttt{ size } 256M \texttt{ rate r(}f_2\texttt{)}\\
&f_2 \texttt{ } src \texttt{ -> } client \texttt{ size } 256M \texttt{ rate r(}f_1\texttt{)}
\end{aligned}
$$

The 256MB size restriction corresponds to the HDFS block size in use. The first set of experiments ran over the gigabit network. First, each node copies a 768MB file from local storage to HDFS. Then, at each step, a percentage of servers become active. In this state, a server will attempt to copy three files, chosen at random, from HDFS to local storage. There is an idle period of up to three seconds (also random) between copy operations. We did the same on EC2, with the only exception being file size, which is now 512MB. Every experiment is repeated 20 times and we measure the duration of each individual transfer. The local results can be found in Fig. 6a and the Amazon ones in Fig. 6c. The horizontal axis contains the percentage of active servers, while the vertical axis shows time, in seconds. We plot both the average value and the 99th percentile of read time.

CloudTalk does improve average performance but only by 10%-30% both on our cluster and on Amazon. The improvements depend on how busy the servers are: when few servers are active, basic HDFS can pick idle sources purely by chance; when many servers are busy, it is harder to find idle replicas. CloudTalk provides a factor of two reductions for the 99% completion times on both our local cluster and the Amazon EC2 deployment.

HDFS Write. The process of writing a file to HDFS is a bit more complicated than reading, as it relies on daisy chaining. When data is written to a new block, the NameNode selects a primary destination together with a group of secondary nodes, up to the number of replicas. As the first server receives and stores data from the client, it also starts sending to the next node, and so on, until a chain is established to the last replica location.

The write experiments are similar to those previously described for read, just that each active server writes files to HDFS instead of retrieving them. In this scenario, the presence of a slow transfer anywhere in the chain will decrease the overall completion time. There are usually at least three replicas required, and this leads to poor performance when replica selection is random.



(a) HDFS Read in Cluster (b) HDFS Write in Cluster (c) HDFS Read in EC2 (d) HDFS Write in EC2

Figure 6: HDFS evaluation on local cluster of 20 machines and 100 EC2 virtual machines.

We have changed the HDFS NameNode to use the following query whenever a block is allocated:

r_1 = r_2 = r_3 = ($dataNode_1$ ... $dataNode_n$)
f_1 *client*->r_1 size 256M rate r(f_2)
f_2 r_1->disk size 256M rate r(f_1)
f_3 r_1->r_2 size 256M rate r(f_4) transfer t(f_2)
f_4 r_2->disk size 256M rate r(f_3)
f_5 r_2->r_3 size 256M rate r(f_6) transfer t(f_4)
f_6 r_3->disk size 256M rate r(f_5)

Figure 6b shows the local results, while the EC2 measurements can be found in Figure 6d. We can see that CloudTalk fares significantly better in terms of average duration now. The basic HDFS often attempts to write on a node already used by another replication process, so it's performance degrades rather quickly. With CloudTalk, it is possible to discover and select less utilised servers and the benefits are obvious: both the average and 99% flow completion times are improved by a factor of 1.5 to 2.

SSD HDFS. As previously mentioned, both on EC2 and in our gigabit network, storage is seldom a bottleneck. We used the 10Gbps local network to create scenarios where contention happens primarily at the disk level. For both read and write, there is a single client, but a variable percentage of servers also run a local process that causes considerable disk utilisation. It continuously reads a very large file (for HDFS read experiments), or repeatedly writes (for HDFS writes). The experiments evaluates how many idle block locations we can find. Files read or written by the client are 4GB large.

The results are shown in Figure 5. An important thing to mention is that even with 10Gbps connectivity, our single client was not able to fully utilise a disk in read scenarios, because it became CPU bound first. However, there still was a noticeable difference when reading from an idle disk. This and the recurrent issue of not being able to find an idle replica when a lot of servers exhibit high disk utilisation, reduce read completion times up to 1.2x. The writes, on the other hand, finish 1.5 to 2 times faster with CloudTalk.

Reduce. Reducer placement is important for a MapReduce job, because large amounts of data are received during the shuffle phase. If a reduce task ends up running on a busy machine, the overall job finish time can increase significantly. The next experiments study the interaction between individual reduce tasks and incoming connections that do not respond to congestion, such as UDP traffic. The MapReduce scheduler tends to spread tasks as much as possible if there are enough nodes and available slots, but does not take into account the existence of other transfers. We evaluate these effects by having UDP iperf connections from outside the Hadoop cluster arrive at a subset of the machines within. The cluster contains 10 servers locally, and 58 instances on EC2. All other machines run iperf senders. The number of connections varies from 10 to 70% of cluster size.

Reduce tasks are not assigned all at once, but rather at most one for each individual heartbeat message received by the scheduler. We do not rely on a single query to find the best servers for the job. Instead, whenever a node request a task to run, its fitness is evaluated after receiving a response to the following query:

x_1 = ... = x_m = ($node_1$ $node_2$ ... $node_n$)
f_1 0.0.0.0 -> x_1 size 1G rate r(f_2)
f_2 x_1 -> disk size 1G rate r(f_1)
...
f_{2m-1} 0.0.0.0 -> x_m size 1G rate r(f_{2m})
f_{2m} x_m -> disk size 1G rate r(f_{2m-1})

This query does not describe exactly what the application will do, but presents a similar scenario that is easier to describe. There are m variables (one for each pending reduce task), and n possible values (one for each node). If there are less nodes than reduce tasks, then everyone receives at least one reduce task. The odd numbered flows specify we are interested in servers that will receive significant incoming traffic. The literal 0.0.0.0 meaning "unknown source" is used instead of faithfully representing the data sent from each mapper, and the size of transfers is set to a constant value. What's important for query evaluation is not the exact byte count, but the fact that all servers receive the same amount of data. Even-numbered flows capture writing data to disk.

The reply will contain the set S of m best servers to place reduce tasks on. A task is given to the current node x only if $x \in S$, and a mechanism that prevents endlessly waiting for the best node in certain situations is in place. We compare the running time of a *sort* job on our modified variant of Hadoop, with the original version. We use the *randomwriter*

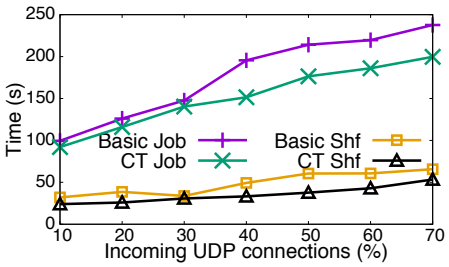


Figure 7: Reducer placement in cluster

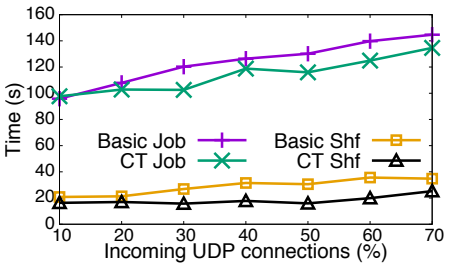


Figure 8: Reducer placement on EC2

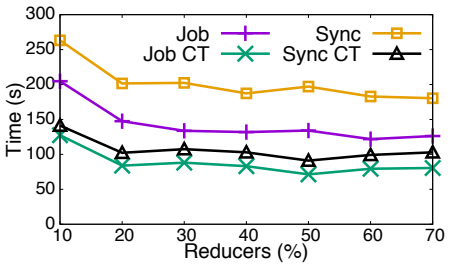


Figure 9: Map-reduce optimisations

example application to generate 512MB (256MB for EC2) of data per cluster node. The number of reducers is set to half the cluster size.

The local results are shown in Figure 7. We measure job completion times and the shuffle duration for successful reduce tasks. While it may appear that we should have an easy time selecting servers that are not on the receiving end of UDP traffic, this is not actually the case. As reduce tasks start, there are still ongoing maps, some of which are not data-local. This means that iperf targets are just some of the nodes which appear to receive a lot of traffic. Also, the overall job duration is subject to delays caused by writing results to HDFS (which is not optimised during these experiments). Still, jobs finish faster when CloudTalk is used because the shuffle times are shorter and it's less likely that one or more reduces will require speculative execution. The EC2 results are presented in Figure 8, and show that shuffle duration is reduced by a factor of 1.1 to 2x.

Map/reduce. We now enable all our optimisations and measure their effect on a map/reduce job. We further implement a simple change in the map scheduler to select the best map assignment for the current node during heartbeat processing. We use the following CloudTalk query:

X = ($node_1$ $node_2$ ... $node_n$)
f_1 disk -> X size $128M$ rate r(f_2)
f_2 X -> currentNode size $128M$ rate r(f_1)

The possible values for variable X are nodes which store a data split that must be processed by a pending map task. After receiving the answer, we select any map task that has input at that particular location for the current node. This and the previous reduce optimisation are simple ways of improving task placement that never skip assigning work. A more complex scenario involves ignoring slow nodes (as long as they appear that way) entirely, even if this means piling multiple tasks on top of several fast workers. However, it requires more intricate queries using a suitable evaluation procedure, and is covered to a certain degree by the use of speculative execution.

Four out of 20 local servers have their SSDs replaced with HDDs, which are 5 to 10 times slower. We run multiple instances of the *sort* job over the gigabit network, with 512MB of data per node. Optimisations are disabled during input generation, otherwise nothing would be written to the HDDs. The number of reducers varies from 10 to 70% of cluster size. Before starting a job, servers drop the buffer cache, to ensure data is read from disk during each run.

The are two interesting metrics: job finish time and job sync time. The latter measures the time elapsed between starting the job and syncing results to disk by running *sync* on all servers on completion. In Figure 9 we see that CloudTalk enabled Hadoop reduces job completion time by a factor of two in all experiments because it avoids (as much as possible) interacting with the slow drives. Mappers prefer to copy data over the network instead of accessing the slow local disks. CloudTalk also picks replica locations that are least contended for both reading and writing. Results show that even a few slow disks can greatly impact performance and that great benefits can be had from using load and capacity information.

5.4 Optimising Web Search

Web search is a classical distributed application which uses a "scatter-gather" workflow. Servers are organized in a hierarchical structure: the query is sent by the frontend towards the leaves, while the results go in the opposite direction. The architecture we used for our experiments has two levels of aggregators, and is shown in Figure 10. Each aggregator has around 50 nodes underneath. We deployed Apache Solr on top of the Apache Tomcat web server on 100 VMs in EC2. Each machine hosts 4GB of data from roughly $5{\cdot}10^6$ indexed URLs (taken from a 2.5TB snapshot of the .uk domain).

We measured the performance of the system in multiple configurations and present the results in Figure 11. First, we measured the raw performance of one machine searching its part of the index — this is the baseline for the distributed measurements. Second, we measured with only one aggregator contacting all 100 servers. When traffic was low, the system was behaving correctly, but with obviously higher delays compared to the single-machine baseline. Once the traffic exceeds 35 queries per second (qps), the aggregator software started crashing, and low-level packet information showed that the effect is due to TCP incast[**?**].

The last tests use the configuration in Figure 10, keeping the location of the frontend and the servers constant while changing the location of the aggregators based on the in-

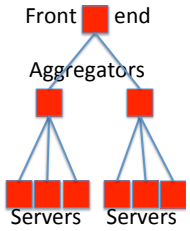


Figure 10: Web Search architecture

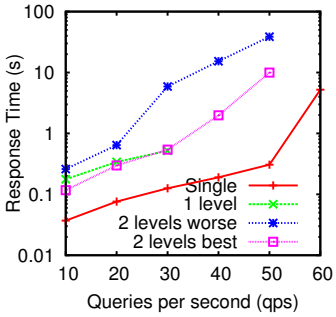


Figure 11: Web search performance

ferred EC2 topology from §3. We tested two cases: "worst" is when we placed the aggregators to be in different subnets to its corresponding nodes, and "best" where each aggregator was placed to maximize the number of servers close to the aggregator: the two aggregators had 14 or 20 machines in the same rack, 15 or 10 in the same subnet and 18 in different subnets. The results show that the effects of incast are smaller compared to the single aggregator scenario. When load increases, the performance for the "best" scenario is substantially better than the "worst" case, even by a factor of more than ten.

Can we use CloudTalk to guide the placement of aggregators? Given that performance is dominated by incast, our flow level evaluator will not work for web-search, so the client can tell CloudTalk to use a packet-level simulator for evaluation instead. CloudTalk uses the *htsim* simulator to simulate a modified VL2 topology containing 1200 servers that mirrors the EC2 topology. We placed the 100 servers in appropriate racks and subnets in the simulated topology and used this query to find aggregator locations:

AGG_1 = AGG_2 = (srv_1 srv_2 ... srv_k)
f_{1a} srv_1 -> AGG_1 size $10KB$
f_{1b} AGG_1 -> frontend transfer t(f_{1a})
...
f_{51a} srv_{51} > AGG_2 size $10KB$
f_{1b} AGG_2 -> frontend transfer t(f_{51a})

Servers addresses are sorted according to proximity. The first 50 servers go to the first aggregator, and the other 50 to the second aggregator. The aggregators can be placed on 10 servers chosen to be in different racks. The aggregators send the data to the frontend, and the query finishes when all the data has been transferred. We evaluated all possible aggregator placements (100), and for each placement we simulate the desired flows in an idle network. With 50-packet buffers per switch port, the predicted query delay when using a single aggregator is 1.04s, 0.55s for the "worst" two-level aggregator setup and 0.4s for the "best" setup, hence our simulator does indeed capture incast effects and could be used for CloudTalk optimisations of web search. Evaluating the entire query takes around 100s. Simulation cannot be used for realtime queries, but is acceptable for web-search since it only runs once, at deployment time.

Another way to handle the web-search query is for the provider to suggest aggregator placement in racks with switches that have larger per-port buffers or to enable priority flow control (PFC) for selected tenant traffic.

5.5 Scaling CloudTalk

Can something like CloudTalk really work at scale, in networks which consist of thousands, tens of thousands, or even more servers? We believe the answer is yes based on the results presented in the previous two sections: the number of servers being asked for status information does not depend on the size of the network and is reasonably small (as it tends to be for most queries we considered), and query evaluation should take a few milliseconds even in the worst case. These results are contingent on having a bimodal-like load distribution, but various studies give us confidence in being able to meet this condition most of the time. We now examine other obstacles to scalability that may arise in practice.

Network overhead. One potential barrier to scaling is the overhead caused by CloudTalk related packets which include queries to status servers (64B) and the associated responses (78B). The CloudTalk overhead of a HDFS read is 1.3KB (0.002% if an entire 64MB block is read). The overhead of an HDFS write in a deployment of 100 nodes is 45KB; again, for large writes this is negligible. Our reduce optimization running on a 100 node cluster with 50 reducers sends 43KB of status messages.

In the examples above, sampling is not used, and our CloudTalk server contacts all 100 nodes. When dealing with much larger clusters, sampling will be enabled, and the network overhead will be similar to the above. In summary, we consider that CloudTalk overhead can be considered negligible for regular applications.

However, malicious users could craft queries at a very fast pace, and the resulting overhead might affect the accuracy of results and increase response times. The cloud provider implicitly pays for all resources associated with the operation of CloudTalk, because this can both make the platform more attractive to clients, and enable the discovery of important information. A conceptually simple solution to avoid issues related to malicious customers is to limit both the arrival rate, and computation time dedicated to incoming queries, depending on the amount and type of acquired VMs. Also, while query answers can be used to infer more information than the explicit recommendation they provide, we consider this does not reveal anything which cannot be already determined by tenants via probing.

Preventing oscillatory behaviour. When a large number of queries are received, the CloudTalk server may inadvertently cause oscillations in terms of resource usage by recommending the same apparently idle servers to multiple clients, until feedback finally arrives from the status servers and indicates that those servers are, in fact, overloaded. Afterwards, another small group of servers may be in the same position,

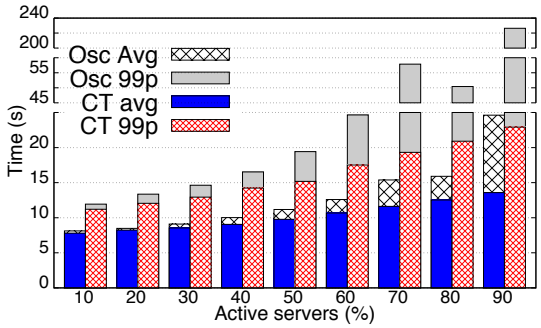


Figure 12: Using hysteresis in resource allocation reduces the tail flow completion times.

and so forth. This behaviour happens because there is an unavoidable delay between the reception of status data by the CloudTalk server, and the measurable effects caused by the application which receives the answer to a query. The delay can be significant, especially when the server must deal with high request rates. Even if finding the answer for a query is fast (under 1 ms), the overall response time perceived by the application can be as high as tens of ms, if one or more status servers fail to provide a reply. This greatly increases the chances of recommending the same set of endpoints multiple times in quick succession.

Figure 12 shows what happens when this behaviour is left unchecked in the EC2 HDFS write scenario described in Section 5.3. During each experiment, an active server copies three files in succession to the DFS. Each server will wait between 0 and 3 seconds (uniformly chosen at random) before each operation. The vertical bars labeled with "Osc" are of particular interest. They represent write completion times when CloudTalk takes no special action to prevent multiple recommendations of the same endpoint in response to multiple queries that arrive in close succession. Results are shown both on average (labeled "avg"), and for the 99th percentile (labeled "99p"). All vertical bars start from zero; those not labeled "Osc" are superimposed over the former, because their value is always smaller. As more servers become active, the tail 99 percentile write time increases to around 4 minutes (ten times the average). This happens because the loaded state of previously recommended servers only becomes apparent after a delay which depends on both the requesting application, and the measurement frequency. During this time, the server appears to be idle, so endpoints from other requests will also be bound to it.

One solution is for CloudTalk to manage resource reservations, instead of just making recommendations, but this greatly complicates things as the abstract state held by the CloudTalk server must be kept in sync with the real state of the network. While this is an interesting future research direction, we have adopted a more pragmatic approach. When an answer is provided in response to a query, the server will consider the machines it has recommended to be in use for a time t, chosen sufficiently large to allow the relevant feedback to arrive from status servers. During the Hadoop experiments, t was set to 300ms; in general, having t less than the minimum possible flow completion time ensures that it doesn't negatively impact network utilisation. This solution is very effective: in Figure 12, the 99% completion time drops to 20s, just double the average. It's important to stress that, at this point, CloudTalk does not employ actual resource reservations, but rather applies a best-effort approach where, presumably, applications always follow placement recommendations. If they do not, then performance is as good as that of random placement.

Queries from multiple users are processed in parallel, and synchronization is required before doing every individual variable assignment, to ensure the status of each endpoint has not been influenced by the response to a different query. When this happens, we attempt to bind the current variable to previously considered endpoints, in decreasing order of their evaluated fitness.

Usage patterns. CloudTalk servers are completely distributed and there is no central coordination needed. However, the way applications use CloudTalkmay result in a single CloudTalk server having knowledge of the whole network and making centralized allocation decisions in certain scenarios. For example, in HDFS write operations are handled by the NameNode that decides replica placement. The NameNode will issue queries to and get answers from the local CloudTalk server which will slowly gather information from all HDFS nodes. Such centralization enabled the oscillatory behaviour above. HDFS reads, on the other hand, are handled in a distributed manner: the clients must choose between available replicas and they query their local CloudTalk server. There were no oscillation-related issues during the read experiments, even without pseudo-reservations. This is not unexpected; even when most servers are active, each server has only three replicas to choose from, so it's a lot less likely to have idle server being recommended to multiple clients querying CloudTalk at the same time.

6. Related Work

Cloud providers offer distributed applications as a service, such as Amazon's MapReduce, which they can optimise appropriately. The downside is that tenants have no control over these frameworks and often choose to deploy their own applications instead. Providers also offer a wide variety of instance types, as well as spot instances that are priced dynamically based on availability. Tenants have some control over resources at instantiation time; for instance, [**?**] and [**?**] use spot requests in addition to on-demand instances to reduce tenant costs.

CloudTalk bears a conceptual resemblance to Mesos [**?**]; both solutions provide, to different extents, a communication medium between a cloud infrastructure and its users. Mesos enables the coexistence of multiple computing frame-

work within the same cluster by brokering resource allocations, while CloudTalk enables indirect tenant access to provider information in a public cloud. Omega [**?**] is a parallel scheduler architecture, that uses shared state to run multiple concurrent schedulers, without resource partitioning. When used in a distributed manner, CloudTalk could rely on similar techniques to achieve a global synchronized state.

Instantiation-time optimisations have been proposed to guide the placement of VMs based on expected traffic patterns [**???**]. Bazaar [**?**] suggests changing the VM instantiation API to allow tenants to express the jobs they intend to run and their desired finish times. However, all these optimisations are coarse grained, only look at network utilisation, and do not account for changing usage patterns for VMs during their lifetime or load variations on the underlying machines. CloudTalk is a complementary approach: it enables *runtime optimisations* that *allow better resource usage at short timescales*.

Sinbad [**?**] shows that modifying distributed programs to improve the outcome of scheduling with outside information is feasible. We want to enable this in a generic setting, without any assumption regarding the nature of the applications.

Coflow [**?**] can express the communication requirements of cluster computing frameworks. It exposes an intent-driven API that focuses on application semantic information, but is limited to a predefined set of communication patterns and only supports network information. CloudTalk is more general as it can express arbitrary communication patterns, as well as disk constraints.

7. Future Work

Our prototype CloudTalk implementation shows that significant benefits can be achieved when tenants and providers cooperate instead of working in isolation. The results presented in this paper represent a couple of important first steps in this direction. There are other possibilities which we didn't get the chance to explore yet.

For example, it's interesting to consider what happens when third party services also exist in the cloud. If the infrastructure is CloudTalk enabled, all parties can use it to individually optimise their applications, but there's no way so far of using CloudTalk between tenants and services which do not belong to the cloud provider. In this situation, a third party service could implement CloudTalk capabilities which can be discovered by clients. For example a third party storage service could optimise reads and writes depending on the resources owned by the requesting client.

CloudTalk can also enable new billing possibilities. Cloud providers can offer lower rates to incentivise clients to describe their workloads (potentially in advance) using queries; this information can be used for better resource planning. Clients could also use CloudTalk queries to describe a particular workload, and then request a price quota from the provider, given the communication will terminate with respect to the specified parameters.

Finally, we also consider extending the set of resources that can be used in CloudTalk queries; CPU and memory immediately come to mind. However, unlike network and disk I/O, their use is more difficult to reason about in a detailed fashion, especially in the presence of multiple tenants. One way to introduce these resources without too much added complexity is to consider both as scalar values: an endpoint may require some number of CPU cores, and a certain amount of memory. Together with the other CloudTalk features, this could enable a more precise offline description of workload requirements, which can guide the VM acquisition process. At the other extreme, we could add more detail to the model (such as CPU cycles, or memory operations), but it's unclear at this point if this is feasible or helpful.

8. Conclusions

Cloud providers want to keep their network topology and load information a trade secret, while the users need it to optimise their applications. Infrastructure cannot be completely hidden from the users, and it is possible to improve app performance significantly even with limited knowledge of the underlying topology.

We proposed CloudTalk, a novel cloud-tenant API, that enables a wide range of distributed application optimisations. CloudTalk users describe their tasks to the cloud and replies help them make the most appropriate choices for work placement. We built a scalable, fully distributed, CloudTalk implementation, that can quickly answer tenant queries in 10ms or less, and that enables a wide range of application optimisations. CloudTalk helps to significantly improve Hadoop, HDFS and Solr performance in all tests.

Acknowledgements

This work was supported by SSICLOPS, a project funded by the European Commission, under its Horizon 2020 programme (contract number 644866).

References

M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. Usenix NSDI 2010*.

M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.

P. Barford, A. Bestavros, J. Byers, and M. Crovella. On the marginal utility of network topology measurements. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, pages 5–17, New York, NY, USA, 2001. ACM.

T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th*

ACM SIGCOMM Conference on Internet Measurement, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 231–242, New York, NY, USA, 2013. ACM.

M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 31–36, New York, NY, USA, 2012. ACM.

M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 98–109, New York, NY, USA, 2011. ACM.

M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 443–454, New York, NY, USA, 2014. ACM.

C. DeSanti. 802.1qbb - priority-based flow control, 2011. http://www.ieee802.org/1/pages/802.1bb.html.

B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 20:1–20:14, New York, NY, USA, 2012. ACM.

A. Greenberg el al. VL2: a scalable and flexible data center network. In *Proc. ACM Sigcomm 2009*.

B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 127–138, New York, NY, USA, 2012. ACM.

A. Iosup, N. Yigitbasi, and D. Epema. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 104–113, May 2011.

V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 10:1–10:14, New York, NY, USA, 2012. ACM.

A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 149–160, New York, NY, USA, 2014. ACM.

K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan. Choreo: Network-aware task placement for cloud applications. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 191–204, New York, NY, USA, 2013. ACM.

I. Menache, O. Shamir, and N. Jain. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. In *Proceedings of the 11th Conference on Autonomic Computing*, 2014.

X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 1154–1162, Piscataway, NJ, USA, 2010. IEEE Press.

B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 58:1–58:11, New York, NY, USA, 2011. ACM.

C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *Proc. ACM SIGCOMM 2011*.

M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM.

J. Seedorf and E. Burger. RFC 5693: Application-layer traffic optimization (ALTO) problem statement, Oct. 2009. http://tools.ietf.org/html/rfc5693.

P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 16:1–16:15, New York, NY, USA, 2015. ACM.

M. Wang, X. Meng, and L. Zhang. Consolidating virtual machines with dynamic bandwidth demand in data centers. In *INFOCOM, 2011 Proceedings IEEE*, pages 71–75, April 2011.

H. Wu, Z. Feng, C. Guo, and Y. Zhang. Ictcp: Incast congestion control for tcp in data center networks. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 13:1–13:12, New York, NY, USA, 2010. ACM.