15. Cloud Federation and Distribution

William Culhane[1], Patrick Eugster[1,2,1], Chamikara Jayalath[3], Kirill Kogan[4], and Julian Stephen[1]

[1]Department of Computer Science, Purdue University, USA
[2]Department of Computer Science, TU Darmstadt, Germany
[3]Google Inc., USA
[4]IMDEA Networks, Spain

Keywords: Multi-datacenter, federation, geo-distribution, partitioning, replication, migration, associativity, multicast

The cloud computing paradigm has significantly evolved beyond the simple early application scenarios such as third-party hosting of web servers. This evolution was triggered by the desire of cloud providers to serve diverse needs of customers around the globe. In particular, the term "cloud" was originally put on par with "datacenter", yet data– and compute-clouds have evolved to complex multi-datacenter infrastructures (see Figure 1). As many cloud-based solutions start to serve customers around the globe or through new media, data may be spread across multiple sites and even cloud offerings for various reasons including low latency retrieval based on geographical proximity, legal constraints, or cost considerations. Regardless of the original motivation, federation across datacenters including especially so-called "geo-distribution" leads to many challenges around (1) the location and access of *data* stored and shared between datacenters, (2) the *computation* on such distributed data, and, in general, around (3) the *communication* of data across datacenters in the context of (1) and (2). This article first motivates federation and then describes the challenges in these three areas and outlines solutions to them.

1. The Case for Federation

The term "cloud" nebulously defines a variety of services related to distributed storage and computation. There are many cloud providers, and the exact services provided vary based on the provider, even at the abstract level. For instance, a provider may provide any – or a combination of – Infrastructure as a Service (IaaS), Platform as a Service (Paas), or Software as a Service (Saas) [1]. A consumer can choose one or several of these based on their own needs and development capabilities. Sometimes multiple offerings have to work together, be it because applications are built from different existing services or components hosted in different clouds, or by design. In fact, there are instances where one cloud service uses another. This is referred to as *vertical* federation, as one cloud offering takes precedence over another in a stack. It is different from *horizontal* federation in

[1] Contact author. `p@cs.purdue.edu`

which services do not strictly rely on accessing each other in the same order. Horizontal and vertical integration may happen in tandem, as when a processing service accesses multiple storage services of different clouds. Technologies including Eucalyptus[2] and OpenNebula[3] can similarly be viewed as supporting vertical federation, building a service/infrastructure on top of horizontally federated cloud services.

Some services are interchangeable, or nearly so. The most straightforward example of this is data storage. A consumer may choose to spread data between providers for security so that the full data is not known by any third party, or any individual breach does not gain access to the full data. Alternatively, a consumer may switch providers when a contract expires or a new product becomes available. In both these cases data has to be accessible across clouds.

Sometimes the choice of which cloud offering to use is motivated by legal concerns, especially with regard to the location or accessibility of data [3]. For instance, the European Union issued a directive controlling where private data may traverse or reside. Even storing data within the EU may make that data subject to the relevant laws regardless of the origin of the data. In the United States, the Health Insurance Portability and Accountability Act and the Gramm-Leach-Bliley Act require strong protection on certain types of data, which limit the hosting options which comply with the Acts. It may also make sense to isolate such data, as it cannot be used for purposes beyond those stated when it was originally collected. Thus if there is any reason the data may be considered sensitive, it is imperative to understand which data can and does fall under various jurisdictions to discern which laws govern its storage and use. If there are any disputes with the cloud provider, it is also necessary to know which laws apply.

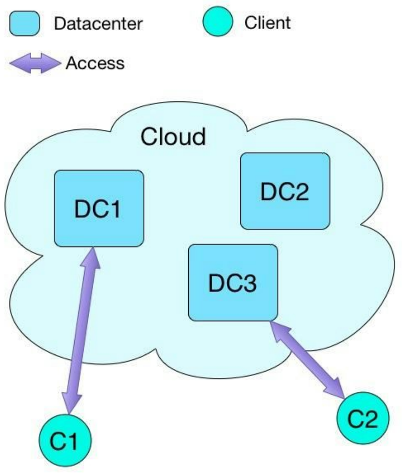


Figure Datacenters and the Cloud

[2] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5071863
[3] http://opennebula.org/

In general to support federation and to avoid a case where the consumer is locked in to a single provider ("vendor lock-in") and a single set of tools it must be possible to (1) distribute data, (2) compute on such data, and (3) communicate it across clouds and thus most commonly across datacenters [2]. Given the many different abstraction levels and even abstractions within a same level offered by different cloud vendors, we focus in the following on how to fundamentally solve these three issues efficiently; these are independent of mediation between specific technologies, APIs, and abstractions.

2. Distributed Data

There are several motivations and ways for distributing data across datacenters. For example, when global businesses offer cloud-based web services at a global scale, they naturally want to serve customers from the closest datacenter. This is illustrated in Figure 1, where client C1 communicates with datacenter DC1, whilst C2 prefers DC3. This can lead to building up customer databases in different datacenters. However, many operations – e.g. audits – have to involve logical datasets that are distributed across datacenters. Customers may not want to pay for moving the data to one place, or such an operation may be infeasible, for example due to legal issues when different laws govern the data hosted at the different sites.

Addressing

Distributed datasets used for a single application can be stored in individual files in distributed file systems such as Hadoop Distributed File System (HDFS) deployed in these datacenters, or in alternate storage systems. (For simplicity we may refer in the following to such datasets as *files*, although they may be stored in other formats.) Such files need not share common formats.

The networks within cloud datacenters usually offer significantly different guarantees than the networks connecting them. Indeed, cloud providers typically control the network within their offerings and as such ensure some practical level of Quality of Service (QoS) for communication; communication between such datacenters may, however, go through the Internet, making it harder to provide guarantees on latency or bandwidth. Consequently distributed file systems such as HDFS typically only support deployments within single datacenters and perform poorly if at all across datacenter boundaries. Each datacenter might host its own deployment of such a file system, and accesses would happen only from within a respective datacenter. However, several geo-distributed file systems and storage systems have been proposed recently, with a global perspective on data management. Two main possibilities approaches exist:

a) Distribution-*aware* addressing: consists of managing a different addressing/name space for every involved datacenter. Thus file operations, including creation and access, are explicitly parameterized by their location. This approach is similar to deploying a separate instance of a storage system in each datacenter and facilitating accesses from remote datacenters.

b) Distribution-*agnostic* addressing: here the level of abstraction is raised by addressing several datacenters as a whole. That is, users do not have to know a priori where exactly their files are located.

The resulting explicit and implicit distribution can also be combined. For instance, with a consolidated name space file creation operations can support optional arguments denoting the desired location(s) for files, or locations can be queried and explicitly modified by users independently of file names per se.

Access features and guarantees are a priori decoupled from the addressing model. For instance, transactional guarantees for multiple combined file accesses are possible with both approaches. However, such advanced features are usually more often found with single address spaces, in addition to other conveniences like automatic load distribution across datacenters, automatic (re-)location of files based on access patterns, or automatic replication for fault-tolerance. We describe these services in more detail shortly

GridFarm [4] is an early example of a file system that supports geo-distributed accesses with explicit denomination of servers which are managing files. Files are addressed and accessed individually without guarantees across accesses.

Partitioning

Sometimes individual logical files are partitioned across datacenters. This can happen for different reasons besides the security concern mentioned in Section 1:

- Human initiative: with large datasets, especially those which keep growing at a high rate, one management mechanism is a manual coarse-grained division of one dataset into several datasets or files which can be stored in a distributed manner. The resulting partitions typically follow a single partition criterion (based on the values for some primary key in the data such as "person with family names starting by A-D", or "logs from machines x-y"), or can apply several such criteria in a nested fashion with directories.
- Technical limitations: despite the tremendous scalability of existing distributed file systems and storage systems, there are always limitations in how much data can be addressed or indexed with a given number of bits. The same goes for the datasets themselves, which can be limited in size, e.g., by the size of individual hard disks. Even without considering such limits or pushing them very far in practice, file systems providing only restricted features for data access can yield bottlenecks in applications that use them. For instance, append-only semantics for files in HDFS become increasingly cumbersome when attempting to access arbitrary parts of larger files.

Figure 2 shows an example where a single dataset DS1 is partitioned into two parts DS1.a and DS2.b, which are stored in two different datacenters respectively.

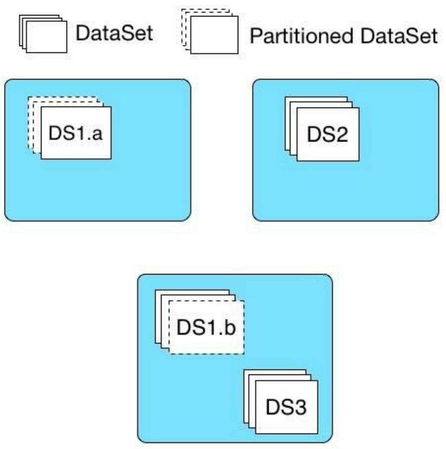


Figure 2 Partitioned Data

Migration

There are many reasons a file might be moved from one location to another, including a change in the rules governing the distribution of data or the need for low latency local accesses. Files may even be moved back and forth or across multiple sites. This is of particular interest in the case of a unique name space for files stored across datacenters. Such a global name space abstracts geo-distribution, yet access times between datacenters can hardly be abstracted away. Thus it becomes of increased importance to keep such access times minimal, which can be achieved by taking into account access patterns of individual files, and performing automatic migration of files to optimize for some objective function such as minimizing the average access latency. Optimization may further take latency and bandwidth or cost of remote communication from a given datacenter to another into account, as well as legal constraints to disable certain migrations. Correlations between accesses to different files can yield additional cues. An example of a system for automatic re-location of files is Volley [5]. Volley analyzes logs of access requests from datacenters, and uses an iterative algorithm to optimize access latencies to individual files as well as costs by taking inter-datacenter communication bandwidths into account to achieve a user-defined desired tradeoff between the cost of pro-active migration and its benefits.

For files which are only read, or for which slight deviation from up-to-date versions can be tolerated, migration can be augmented with *caching* for faster access.

Replication and redundancy

Another common technique to orchestrate distributed accesses is to replicate files. Figure 3 shows an example where two replicas of a dataset DS1 – DS1.1 and DS1.2– are stored in two datacenters. The resulting redundancy can also be used to provide tolerance towards failures of nodes hosting files, or possibly larger outages in datacenters. Inversely, such fault tolerance can motivate multi-datacenter setups.

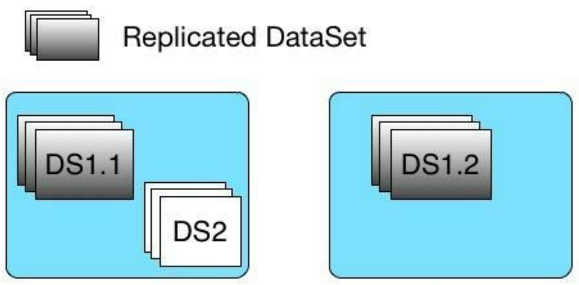


Figure 3 Replicated Data

In the case of files that can be modified, replication immediately leads to the issue of synchronizing concurrent accesses, which is straightforwardly solved when there is a single copy for a file. Several storage systems support geo-distributed replication and provide fault tolerance, differing in the guarantees that they provide.

Consistency

Consistency guarantees become of utmost important in the context of accesses on distributed shared data, especially with replication and failures. The CAP theorem states that the impossibility of providing consistency, availability, and partition tolerance [6]; only two of them can be implemented simultaneously.

Several systems thus focus on crash failures, and/or provide weaker consistency guarantees. For example, Dynamo [7] is a key-value storage, which provides only eventual consistency. Vivace [8] provides strong consistency in the presence of failures and congestion.

Security

Geo-distributed file accesses exacerbate security problems, e.g.,

- Access control: access control solutions need to be deployed across datacenters, which usually comes down to deploying such solutions in datacenters individually, and coordinating them manually.
- Data transfer: if the cloud provider is trusted (cf. private clouds), users may be willing to store sensitive data in respective datacenters. With inter-datacenter communication taking place across the Internet, most users will want to encrypt data that is thus transferred though, which further increases the performance overhead of cross-datacenter communication and motivates the consideration of distribution constraints in communication and computation.

3. Distributed Computation

With datasets being distributed across datacenters, one also needs to consider how to best perform computations on such datasets. This is of particular relevance in the context of datasets which are partitioned across multiple datacenters. While the concepts introduced and discussed in the following are generic in nature, we highlight them mostly in the light of a core application scenario for (geo-distributed) cloud-based computation, namely that of (big) data analytics.

Centralized computation

With data being geo-distributed, even with replication, there might not always be a copy of every relevant data-item available at every site where computation occurs.

The simplest approach to deal with this situation is to copy – if possible – all data to a single location before performing any computation on it. In the example scenario given in Figure 4 dataset DS1 is copied to datacenter DC2 prior to performing a computation that involves multiple datasets. This approach is simple implementation-wise, yet there are a number of associated disadvantages:

1. Performance: before executing a program, all data has to be transferred locally, whilst at least parts of the computation could possibly be performed locally to the data. Without specific support, this becomes especially inefficient when programs are run repeatedly (possibly after minor modification) on the same data, leading to repeated copying.
2. Cost: with many cloud providers charging for inter-datacenter communication (while intra-datacenter communication is typically free of charge), transfers of large datasets can become financially expensive.
3. User effort: in order to avoid that the same data is transferred several times, a straightforward approach is for the user to manually deal with fetching data and copying it close-by at need. This is however tedious, as it requires users to manually deal with updates to datasets.

While several systems have been proposed to store data in a geo-distributed manner, computation has been given only little attention thus far.

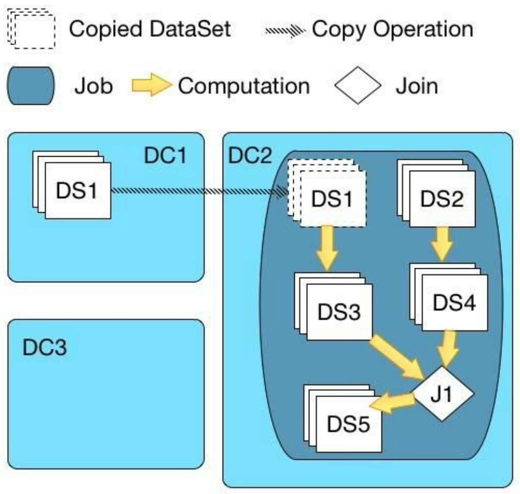


Figure 4 Centralized Computation

Decentralized computation

Performing as much computation as possible close to the data usually improves performance as well as minimizes costs. This is particularly valid for big data analysis, where the amount of data tends to decrease as the computation proceeds – typically large datasets are used as inputs to the computations, which attempt to extract specific, more concise, knowledge from those datasets. Additionally, legal constraints or security concerns can prevent the copying of data to the site of the issuer of a query. Airavat [9] thus supports remote statistical queries such as averages over large datasets via MapReduce, combined with access control and query analysis to allow precisely restricted third-party queries on protected datasets. However, Airavat still assumes that all data is located in a single datacenter.

Implementing generic decentralized computation is far from trivial, when the goal is to optimize performance and/or cost, as there are different *execution paths* according to which computational steps can be interleaved with consolidation of data from several datacenters. Consider an operation involving two datasets implemented through a single MapReduce task which, as its name suggests, consists of two phases: a first map phase which outputs *intermediate* <`key`, `value`> pairs for its input datasets, followed by a reduce phase which atomically processes <`key`, `list`<`value`>> pairs where an instance of list<value> represents all `value`s generated a same `key`. Considering that at some point in the computation all data has to be consolidated in a single datacenter, there are three straightforward execution paths, differing by when the consolidation happens:

a) Pre-map: copying one of the datasets to another datacenter – e.g., the datacenter hosting the smaller dataset – prior to computation yields an execution path similar to performing computation in a centralized fashion, and in some cases might be the only option due to the nature of the job.
b) Post-map/pre-reduce: here the first phase can occur on datasets individually, with copying occurring before the reduction phase, by aggregating <`key`, `list`<`value`>> pairs for a same `key` across datacenters.
c) Post-reduce: consolidating data at the end of the reduce phase assumes that typically no two datacenters will generate intermediate `value`s for a same `key`, or that some simple way exists to aggregate results after the reduction.

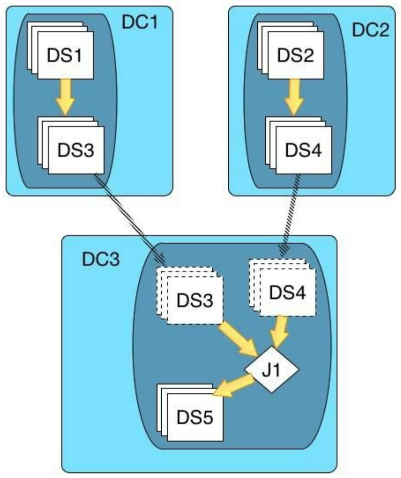
This is only a simplification: the above-mentioned approaches could be combined in many different ways especially when more than two datasets are involved, e.g., consolidating some datasets following path a), with further consolidation following b). The situation becomes yet more complicated when – as common in real-life MapReduce computations – several such tasks are performed in sequence. Relevant parameters for optimization beside latency, bandwidth, and cost of inter-datacenter communication as in data management include but are not limited to:

- Amount of data present at individual points in the computation.
- Complexity and costs of individual computational steps.

- (Momentary) resource availabilities at datacenters.

As emphasized by the possibly transient nature of resource availabilities, determining an optimal execution path can require runtime information and conflict with other jobs executed simultaneously on a shared cluster of nodes. Rout [10] is an example of a seminal system for geo-distributed big data processing based on MapReduce. Rout employs a lazy copying heuristic based on momentary resource availabilities in involved datacenters (or more precisely, clusters of nodes allocated for MapReduce tasks in respective datacenters). Computation proceeds individually at the sites of involved datasets until consolidation becomes necessary, at which point cluster resource availabilities are balanced with amounts of data to be transferred and communication bandwidths in order to choose the datacenter in which computation is to proceed. Another example of a system supporting geo-distributed computation is G-MR (Geo-distributed MapReduce) [11]: G-MR achieves more fine-grained optimization by using sampling to determine the amounts of data at respective points in the computation, yet uses an offline optimization technique which does not take resource availabilities into account at runtime, yet can support other computation models than (sequences) of MapReduce jobs. Legal policies can similarly be considered. Figure 5 shows how the computation of Figure 4 can be executed in a decentralized manner, involving three different datacenters. Note that computation may also be replicated, for instance to ensure high availability.

Figure 5 Decentralized Computation



Associativity and distributed associativity

As hinted to above, a reasonable consolidation heuristic during computation on geo-distributed and possibly partitioned datasets consists in consolidating data as late as possible, given that the amount of data typically decreases as computation proceeds. However, it is not always possible to perform computation on sub-datasets in isolation. We can distinguish several scenarios:

a) Associative computations: following the mathematical definition of associativity, some functions *f* can be performed on arbitrary subsets of data in isolation, before being applied to the outcome of some of these sub-computations, etc. Consider summing the values of some particular attribute across many records: we can create partial sums and sum these further in whatever order, as long as every partial sum is considered exactly once.
b) "Distributed associativity" [12]: certain functions *f* can be "made associative", by describing them alternatively in two steps $f = h \circ g$ where *g* be performed on sub-datasets whilst the second *h* consolidates these results in an arbitrary number of stages in an associative manner. As an example, an average can be expressed as follows: a first function which computes for a sub-dataset (1) the respective average as well as (2) *the number of elements considered*; the second function outputs the sum of all the element counts (2) for any number of inputs in addition to the respective average computed by prorating the averages of its inputs by the respective number of elements (and dividing the result by the total number of elements considered thus far).
c) As a special as for the above we can consider the scenario where the first function is the "original" one, that is, the function that we want to apply to an entire dataset ($f = h \circ f$). In this case we can refer to the function *h* as a "merge" or "aggregation" function for the function *f*. An example consists of counting the number of occurrences of some value across several datasets or a partitioned dataset: we can count the number of occurrences on each sub-dataset, before summing these counts in whatever order we like.
d) Several authors have considered *inferring* merge functions (*h*) or decompositions (*g, h*) automatically or with little programmer input. Most notably, the Third Homomorphism Theorem [13] states that if two sequential programs iterate a list from opposite sides and compute the same value, there exists a parallel program which can divide-and-conquer the program with the same results.

Programming

The impossibility to automatically infer merge functions in general naturally leads to the question of how to aid the programmer in expressing such functions.

In the case of big data analysis, several so-called "data-flow" languages have been proposed in the recent past, which represent data at individual stages by some form of *collection* or *data-structure*, and computational steps as operations performed on such data-structures. Such languages are usually compiled to MapReduce jobs. Languages differ in the data-structures that they propose, though most include some form(s) of sets or bags along with some kind(s) of associative maps. For instance, Pig Latin [14] introduces bags and maps. While early languages tried to be mostly distribution-transparent, more recent approaches for the sake of performance and advanced functionality (e.g., supporting iteration and increment computation) abandon transparency and provide richer APIs to give the programmer more explicit choices of how to perform operations. For instance, resilient distributed datasets (RDDs) [15] go as far as exposing `map` and `reduce` operations which are parameterized by functions; the signature and semantics of the application of these functions is given by the types of their respective formal arguments and the operation through which they are applied. In a similar vein, Rout thus extends Pig Latin with support for merge functions for many of the language's built-in operators, while for others (e.g., `COUNT`), the runtime has built-in knowledge on how to perform these in an associative or distributed associative manner.

4. Distributed Communication

Executing across multiple datacenters requires generic support for communicating between nodes spread across these datacenters. This leads to various challenges beyond efficiency, due to the different mechanisms and features supported by cloud providers. The space of solutions to overcome these challenges is yet very limited.

Mechanisms

Different cloud providers support different basic communication mechanisms, and to different extents. Two particular limitations are:

1. Multicast: Few cloud providers support IP Multicast or UDP Broadcast, for simple reasons. First off, different users/applications could use the same IP Multicast address. Second, such clashes could be exploited to generate denial of service attacks. Yet, since cloud host resources are often exploited in bunches, multicast addressing is a common and thus relevant scenario.
2. Host addresses: Most cloud providers do not expose hardware addresses, or propose proprietary addressing mechanisms to avoid enabling arbitrary direct access to hosts from outside of clouds. Such accesses can similarly lead to security vulnerabilities, inefficiencies, and bypass cost accounting.

To make up for these limitations, cloud providers typically offer specialized middleware services which are however not portable or interoperable across cloud providers and often times do not operate efficiently across datacenters of a same cloud provider. Examples of specialized services are Amazon EC2's CloudFront content delivery network or its Simple Notification Service.

Communication models

Several core models of cloud communication can be distinguished. Roughly:

- One-to-one (unicast): This scenario of communication between two endpoints remains the most common one. One can further distinguish here between different semantics, and uni- vs. bi-directional communication.
- One-to-many (multicast): We can further subdivide this scenario:
    - "Explicit multicast": Here, a component producing data wants to share that data with an explicitly specified set of destinations.

- “Implicit multicast”: Here the set of actual destination components is given by these components themselves, e.g., based on matching of message attributes to desired ranges of values specified by them.

- One-of-many (message queuing): This is similar to implicit multicast, yet the system selects exactly one destination for every message based on characteristics of potential destination components (e.g., current load), specific policies (e.g., for “fair” load balancing), or others.
- One-to-all (broadcast): Here, a message is sent to all “participating” components, typically of a given application instance.
- Many-to-many: Here we consider systems that allow for different messages to be aggregated within the middleware, thus allowing composite messages to be delivered directly to applications.

Figure 6 illustrates a s ... nts distributed across multiple datacenters.

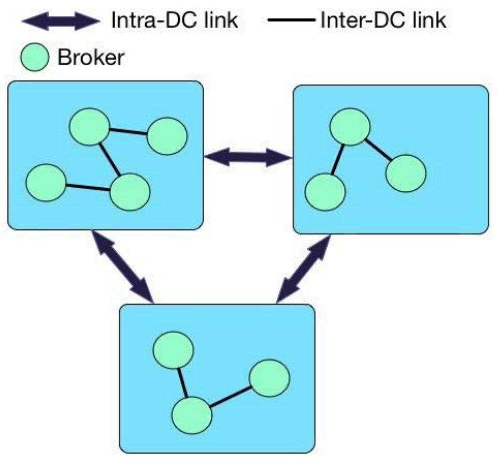


Figure Components Communicating Across Datacenters

Abstractions

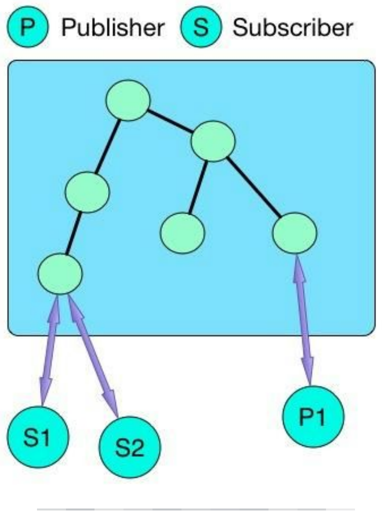
There are several traditional abstractions for programming distributed applications in general, which also play a role in cloud datacenter communication:

- Sockets: these are typical for programming unicast interaction (one-to-one) as well as simple one-to-many interaction
- Publish/subscribe: this is a common high-level abstraction for multicasting. Producers are termed *publishers* and consumers *subscribe* to specific messages. Figure 7 illustrates entities of a publish/subscribe system. Several variants for outlining messages of interest can be distinguished:
  - Topic-based publish/subscribe (TPS): here, publishers explicitly post messages under specific *topics*, whilst subscribers delineate topics of interest. Topics can exhibit a “flat” name-space or be in different relationships to each other such as hierarchical.
  - Content-based publish/subscribe (CPS): here subscriptions are expressed based on predicates on message content ("filters"). Most

systems thus support messages with some form of property-value pairs which are used for filtering.

- Message queuing: these systems are the most common implementation of one-of-many communication. Typically consumers pull messages from queues, but this can also be replaced by push-style communication.

Sockets typically are bound to IP addresses, which given the problems with such addresses outside of clouds makes them unsuited for inter-cloud communication.



Elasticity

Elasticity is a core tenet of third-party cloud computing, whose economic benefits precisely hinge on the ability to tap into resources at need, pay as one goes, and release unneeded resources instantly to avoid any amortization or constant costs. This necessitates communication mechanisms to efficiently scale up and down: a component may have to communicate with 100s of others to inform them of changes relevant to their individual sub-computations, and the next moment there may only be 10 such components necessary to proceed. Based on this requirement as well as other cloud-specific needs already mentioned, the publish/subscribe abstraction is a suitable candidate for elastic communication in and across clouds:

1. Addressing: the logical addressing introduced by publish/subscribe abstracts cloud-specific addressing, which in turn enables inter-datacenter and inter-cloud communication.
2. Scale: the publish/subscribe paradigm is able to capture both unicast (one-to-one) and multicast (one-to-many) communication scenarios.
3. Integration: especially the property-name abstraction offered by content-based publish/subscribe integrates well with other systems such as many storage systems which focus on key-value pairs.

However, typical publish/subscribe systems focus on scaling *up*, and less on small scales. TPS systems typically scale up to many topics, and many consumers per topic although many existing systems work well with small consumer numbers for given topics. CPS systems integrate better abstraction-wise with other cloud services and are more expressive than their TPS counterparts, yet do not deal well with scenarios where few consumers are listening to a given producer. Yet, many scenarios exist where a producer's messages are of relevance to a handful of consumers: a common pattern is three-fold replication of components for fault-tolerance; CPS systems *performance-wise* deal poorly with such scenarios.

Atmosphere [16] is a CPS system designed specifically for cloud communication, in a way support multi-datacenter setups as well as scaling down and up. The system uses original protocols for tracking the number and distribution of subscribers for specific publishers, and splitting the traditional overlay networks used in CPS systems in order to facilitate efficient small-scale communication: the overlay is used primarily to keep components connected, whilst "shortcuts" are created over this overlay for more efficient propagation of messages

5. Conclusions

No single cloud solution will be sufficient for all uses while still being wieldy enough to use, and some consumers will want to use different services or separate their data. Thus cloud federation is necessary. Several solutions exist to distribute and migrate data, as well as to provide services across clouds. Clouds may rely on each other, in vertical federation, or coexist to be accessed by a third service in horizontal federation. New services may be introduced to remap proprietary services to a new abstraction, or services may be tied together to create seamless interaction.

6. References

[1] A. Lenk, M. Klems, J. Nimis, S. Tai and T. Sandholm, "What's inside the Cloud? An architectural map of the Cloud landscape," in *CLOUD '09 Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, Washington, DC, USA, 2009.

[2] T. Kurze, M. Klems, D. Bermbach, A. Lenk, S. Tai and M. Kunze, "Cloud Federation," in *CLOUD COMPUTING 2011, The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, Rome, Italy, 2011.

[3] V. Winkler, Securing the Cloud: Cloud Computer Security Techniques and Tactics, Waltham, MA, USA: Elsevier, 2011, pp. 74-84.

[4] O. Tatebe, S. Sekiguchi, Y. Morita, S. Matsuoka and N. Soda, "Worldwide Fast File Replication on Grid Datafarm," in *Computing in High Energy and Nuclear Physics*, La Jolla, CA, USA, 2003.

[5] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman and H. Bhogan, "Volley: Automated Data Placement for Geo-Distributed Cloud Services," in *Proceedings of the 7th USENIX Conference on Networked System Design and Implementation*, San Jose, CA, USA, 2010.

[6] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the 19th Annual ACM Symposiem on Principles of Distributed Computing*, Portland, OR, USA, 2000.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, Stevenson, Washington, USA, 2007.

[8] B. Cho and M. K. Aguilera, "Surviving congestion in geo-distributed storage systems," in *Presented as part of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, 2012.

[9] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov and E. Witchel, "Airavat: Security and Privacy for MapReduce," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, USA, 2010.

[10] C. Jayalath and P. Eugster, "Efficient Geo-Distributed Data Processing with Rout," in *Proceedings of the 33rd International Conference on Distributed Computing Systems*, Philadelphia, PA, USA, 2013.

[11] C. Jayalath, J. Stephen and P. Eugster, "From the Cloud to the Atmosphere: Running MapReduce across Datacenters," *IEEE Transactions on Computers,* pp. 74-87, 27 May 2013.

[12] Y. Yu, P. K. Gunda and M. Isard, "Distributed aggregation for data-parallel computing: interfaces and implementations," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, Big Sky, MT, USA, 2009.

[13] A. Morihata, K. Matsuzaki, Z. Hu and M. Takeichi, "The Third Homomorphism Theorem on Trees: Downward \& Upward Lead to Divide-and-conquer," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Savannah, GA, USA, 2009.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins, "Pig Latin: A Not-so-foreign Language for Data Processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2008.

[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker and I. Stoica, "Resilient Distributed Datasets: a Fault-Tolerant

Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, CA, USA, 2012.

[16] C. Jayalath, J. Stephen and P. Eugster, "Atmosphere: A Universal Cross-Cloud Communication Infrastructure," in *Proceedings of the 14th ACM/IFIP/USENIX Middleware Conference*, Beijing, China, 2013.