HAL open science

Atmosphere: A Universal Cross-Cloud Communication Infrastructure

Chamikara Jayalath, Julian James Stephen, Patrick Eugster

To cite this version:

Chamikara Jayalath, Julian James Stephen, Patrick Eugster. Atmosphere: A Universal Cross-Cloud Communication Infrastructure. 14th International Middleware Conference (Middleware), Dec 2013, Beijing, China. pp.163-182, 10.1007/978-3-642-45065-5_9 . hal-01480796

HAL Id: hal-01480796

https://inria.hal.science/hal-01480796v1

Submitted on 1 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Atmosphere: A Universal Cross-Cloud Communication Infrastructure*

Chamikara Jayalath, Julian Stephen, and Patrick Eugster

Purdue University, Department of Computer Science,
305 N. University Street, West Lafayette, IN 47907.
{cjayalat,stephe22,peugster}@cs.purdue.edu

Abstract. As demonstrated by the emergence of paradigms like *fog computing* [1] or *cloud-of-clouds* [2], the landscape of third-party computation is moving beyond straightforward single datacenter-based cloud computing. However, building applications that execute efficiently across data-centers and clouds is tedious due to the variety of communication abstractions provided, and variations in latencies within and between datacenters.
The *publish/subscribe* paradigm seems like an adequate abstraction for supporting "cross-cloud" communication as it abstracts low-level communication and addressing and supports many-to-many communication between publishers and subscribers, of which one-to-one or one-to-many addressing can be viewed as special cases. In particular, *content-based publish/subscribe* (CPS) provides an expressive abstraction that matches well with the key-value pair model of many established cloud storage and computing systems, and decentralized overlay-based CPS implementations scale up well. On the flip side, such CPS systems perform poorly at small scale. This holds especially for multi-send scenarios which we refer to as entourages that range from a channel between a publisher and a single subscriber to a broadcast between a publisher and a handful of subscribers. These scenarios are common in datacenter computing, where cheap hardware is exploited for parallelism (efficiency) and redundancy (fault-tolerance).
In this paper, we present Atmosphere, a CPS system for cross-cloud communication that can dynamically identify entourages of publishers and corresponding subscribers, taking geographical constraints into account. Atmosphere connects publishers with their entourages through *überlays*, enabling low latency communication. We describe three case studies of systems that employ Atmosphere as communication framework, illustrating that Atmosphere can be utilized to considerably improve cross-cloud communication efficiency.

Keywords: cloud, publish/subscribe, unicast, multicast, multi-send

1 Introduction

Consider recent paradigm shifts such as the advent of *cloud brokers* [3] for mediating between different cloud providers, the *cloud-of-clouds* [2] paradigm denoting the integration of different clouds, or *fog computing* [1] which similarly signals a departure

* Supported by DARPA grant # N11AP20014, PRF grant # 204533, Google Research Award "Geo-Distributed Big Data Processing", Cisco Research Award "A Fog Architecture".

from straightforward third-party computing in a single datacenter. However, building *cross-cloud* applications — applications that execute across datacenters and clouds — is tedious due to the variety of abstractions provided (e.g., Infrastructure as a Service vs. Platform as a Service).

Cross-cloud communication. One particularly tedious aspect of cross-cloud integration, addressed herein, is communication. Providing a communication middleware solution which supports efficient cross-cloud deployment goes through addressing a number of challenges. A candidate solution should namely

R1. support a variety of *communication patterns* (e.g., communication rate, number of interacting entities) effectively. Given the variety of target applications (e.g., social networking, web servers), the system must be able to cope with one-to-one communication as well as different forms of multicast (one-to-many, many-to-many). In particular, the system must be able to scale up as well as down ("elasticity") based on current needs [4] such as number of communicating endpoints.
R2. run on standard "low-level" network layers and abstractions without relying on any specific protocols such as IP Multicast [5] that may be deployed in certain clouds but not supported in others or across them [6].
R3. provide an interface which hides cloud-specific hardware addresses and integrates well with abstractions of widespread cloud storage and computing systems in order to support a wide variety of applications.
R4. operate efficiently despite varying network latencies within/across datacenters.

Publish/subscribe for the cloud. One candidate abstraction is *publish/subscribe*. Components act as *publishers* of messages, and dually as *subscribers* by delineating messages of interest. Examples of publish/subscribe services designed for and/or deployed in the cloud include Amazon's Simple Notification Service (SNS) [7], Apache Hedwig [8], LinkedIn's Kafka [9], or Blue Dove [4]. Intuitively, publish/subscribe is an adequate abstraction because it supports generic many-to-many interaction, shields applications from specific lower-level communication — in particular hardware addresses — thus supporting application interoperability and portability. In particular, *content-based publish/subscribe* (CPS) [10,11,12,13,14] promotes an addressing model based on message *properties* and corresponding values (subscribers delineate values of interest for relevant properties) which matches well the key-value pair abstractions used by many cloud storage (e.g., [15,16]) and computing (e.g., [17]) systems.

Limitations. However, existing publish/subscribe systems for the cloud are not designed to operate beyond single datacenters, and CPS systems focus on scaling *up* to large numbers of subscribers: to "mediate" between published messages and subscriptions, CPS systems typically employ an overlay network of brokers, with filtering happening downstream from publishers to subscribers based on upstream aggregation of subscriptions. When messages from a publisher are only of interest to one or few subscribers, such overlay-based multi-hop routing (and filtering) will impose increased

latency compared to a direct *multi-send* via UDP or TCP from the publisher to its subscribers. Yet such scenarios are particularly wide-spread in third-party computing models, where many cheap resources are exploited for parallelism (efficiency) or redundancy (fault-tolerance). A particular example are distributed file-systems, which store data in a redundant manner to deal with crash failures [18], thus leading to frequent communication between an updating component and (typically 3) replicas. Another example for multi-sends are (group) chat sessions in social networks.

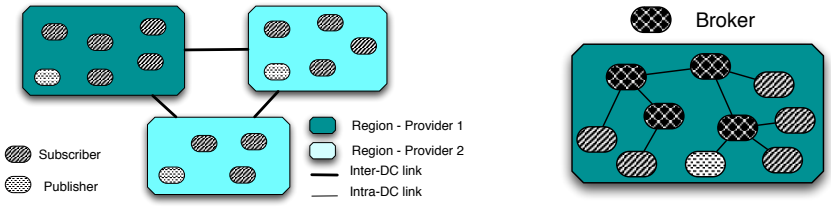
Existing approaches to adapting interaction and communication between participants based on *actual* communication patterns (e.g., [19,4,20]) are agnostic to deployment constraints such as network topology. *Topic-based publish/subscribe* (TPS) [21,22] — where messages are published to *topics* and delivered to consumers based on topics they subscribed to — is typically implemented by assigning topics to nodes. This limits the communication hops in multi-send scenarios, but also the number of subscribers.

In short, CPS is an appealing, generic, communication abstraction (R2, R3), but existing implementations are not efficient at small scale (R1), or, when adapting to application characteristics, do not consider deployment constraints in the network (R4); inversely, TPS is less expressive than CPS, and existing systems do not scale up as well.

Atmosphere. This paper describes Atmosphere, a middleware solution that aims at supporting the expressive CPS abstraction across datacenters and clouds in a way which is effective for a wide range of communication patterns. Specifically, our goal is to support the extreme cases of communication between individual pairs of publishers and subscribers (unicast) and large scale CPS, and to elastically scale both up *and* down between these cases, whilst providing performance which is comparable to more specialized solutions for individual communication patterns. This allows applications to focus on the logical content of communication rather than on peer addresses even in the unicast case: application components need not contain hardcoded addresses or use corresponding deployment parameters as the middleware automatically determines associations between publishers and subscribers based on advertisements and subscriptions.

Our approach relies on a CPS-like peer-based overlay network which is used primarily for "membership" purposes, i.e., to keep participants in an application connected, and as a fallback for content-based message routing. The system dynamically identifies clusters of publishers and their corresponding subscribers, termed *entourages* while taking network topology into account. Members of such entourages are connected directly via individual "over-overlays" termed *überlays*, so that they can communicate with low latency. The überlay may only involve publishers and subscribers or may involve one or many brokers depending on entourage characteristics and resource availabilities of involved publishers, subscribers, brokers, and network links. In any case, these *direct connections* which are gradually established based on resource availabilities, will effectively reduce the latency of message transfers from publishers to subscribers.

Contributions. Atmosphere adopts several concepts proposed in earlier CPS systems. In the present paper, we focus on the following novel contributions of Atmosphere:



(a) Regions (b) Brokers

Fig. 1. Bird's-eye View

1. a technique to dynamically identify topic-like entourages of publishers in a CPS system. Our technique hinges on precise advertisements. To not compromise on flexibility, advertisements can be updated at runtime;
2. a technique to efficiently and dynamically construct überlays interconnecting members of entourages with low latency based on resource availabilities;
3. the implementation of a scalable fault tolerant CPS system for geo-distributed deployments named Atmosphere that utilizes our entourage identification and überlay construction techniques;
4. an evaluation of Atmosphere using real-life applications, including social networking, news feeds, and the ZooKeeper [23] distributed lock service, demonstrating the efficiency and versatility of Atmosphere through performance improvements over more straightforward approaches.

Roadmap. Section 2 provides background information and related work. Section 3 presents our protocols. Section 4 introduces Atmosphere. Section 5 evaluates our solution. Section 6 draws conclusions.

2 Background and Related Work

This section presents background information and work related to our research.

2.1 System Model

We assume a system of processes communicating via unicast channels spanning g cloud datacenters or more generally *regions*. Regions may be operated by different cloud providers. Each region contains a number of components that produce messages and/or that are interested in consuming messages produced. Figure 1(a) shows an example system with three regions from two different providers where each region hosts a single producing and multiple consuming components.

2.2 CPS Communication

With *content-based publish/subscribe* (CPS), a message produced by a publisher con-

tains a set of *property-value* pairs; inversely, components engage into consumption of messages by issuing subscriptions which consist in ranges of values – typically defined indirectly through operators such as $\leq$ or $\geq$ and corresponding threshold values.

A *broker overlay network* typically mediates the message distribution between publishers and subscribers. A broker, when receiving a message, analyzes the set of property-value pairs, and forwards the message to its neighbors accordingly. (For alignment with the terminology used in clouds we may refer to properties henceforth as *keys*.) Siena [24] is a seminal CPS framework for distributed wide-area networks that spearheaded the above-mentioned CPS overlay model. Siena's routing layer consists of broker nodes that maintain the interests of sub-brokers and end hosts connected to them in a *partially ordered set* (poset) structure. The root of the poset is sent to the *parent broker* to which the given broker is subscribed to. CPS systems like Siena employ *subscription summarization* [10,25] for brokers to construct a summary of the interests of the subscribers and brokers connected to it. This summary is sent to neighboring brokers. A broker that receives a published message determines the set of neighbors to which the message has to be forwarded by analyzing the corresponding subscription summaries. Summaries are continuously updated to reflect the changes to the routing network, occurring for instance through joins, leaves, and failures of subscribers or brokers.

2.3 Existing CPS System Limitations

When deployed naïvely, i.e., without considering topology, in the considered multi-region model (see Figure 1(a)) CPS overlays will perform poorly especially if following a DAG as is commonly the case, due to the differences in latencies between intra- and inter-region links. To cater for such differences, a broker network deployed *across regions* could be set up such that (1) brokers in *individual regions* are *hierarchically* arranged and each subscriber/publisher is connected to exactly one broker (see Figure 1(b)), and (2) root brokers of individual regions are connected (no DAG). The techniques that we propose shortly are tailored to this setup.

However, the problem with such a deployment is still that — no matter how well the broker graph matches the network topology — routing will happen in most cases over multiple hops which is ineffective for multi-send scenarios where few subscribers only are interested in messages of a given publisher. In the extreme case where messages produced by a publisher are consumed by a single subscriber there will be a huge overhead from application-level routing and filtering over multiple hops compared to a direct use of UDP or TCP. The same holds with multiple subscribers as long as the publisher has ample local resources to serve all subscribers over respective direct channels.

While several authors have proposed ways to identify and more effectively interconnect matching subscribers and publishers, these approaches are deployment-agnostic in that they do not consider network topology (or resource availabilities). Thus they trade *logical proximity* (in the message space) for *topological proximity*.

Majumder et al. [26] for instance show that using a single minimum spanning or a Steiner tree will not be optimal for subscriptions with differing interests. They propose a multiple tree-based approach and introduce an approximation algorithm for finding the optimum tree for a given type of publications. But unlike in our approach these trees are location agnostic hence when applied to our model a given tree may contain

brokers/subscribers from multiple regions and a given message may get transmitted across region boundaries multiple times unnecessarily increasing the transmission latency. Sub-2-Sub [19] uses gossip-based protocols to identify subscribers with similar subscriptions and interconnect them in an effective manner along with their publishers. In this process, network topology is not taken into account, which is paramount in a multi-region setup with varying latencies. Similarly, Tariq et al. [20] employ spectral graph theory to efficiently regroup and connect components with matching interests, but do not take network topology or latencies into account. Thus these systems can not be readily deployed across regions. Publiy+ [27] introduces a publish/subscribe framework optimized for bulk data dissemination. Similar to our approach, brokers of Publiy+ identify publishers and their interested subscribers and instruct them to directly communicate for disseminating large bulk data. Publiy+ uses a secondary content-based publish/subscribe network only to connect publishers and interested subscribers in different regions. Publiy+ is not designed for dissemination of large amounts of small messages since the data dissemination between publishers and subscribers is always direct and the publish/subscribe network is only used to form these direct connections.

2.4 Other Solutions for Cloud Communication

Cloud service providers such as Microsoft and Amazon have introduced *content delivery networks* (CDNs) for communication between their datacenters. Microsoft Azure CDN caches Azure blob content at strategic locations to make them available around the globe. Amazon's CloudFront is a CDN service that can be used to transfer data across Amazon's datacenters. CloudFront can be used to transfer both static and streamed content using a global network of *edge locations*. CDNs focus on stored large multimedia data rather than on live communication. Also, both above-mentioned CDN networks can be used only within their respective service provider boundaries and regions.

Volley [28] strategically partitions geo-distributed *stored* data so that the individual data items are placed close to the global "centroid" of the past accesses.

Use of IP Multicast has been restricted in some regions and across the Internet due to difficulties arising with multicast storms or multicast DOS attacks. Dr. Multicast [6] is a protocol that can be implemented to mitigate these issues. The idea is to introduce a new logical group addressing layer on top of IP Multicast so that access to physical multicast groups and data rates can be controlled with a acceptable user policy. This way system administrators can place caps on the amount of data exchanged in groups and the members that can participate on a group. Dr. Multicast specializes on intra-datacenter communication and does not consider inter-datacenter communication.

3 Entourage Communication

In this section, we introduce our solution for efficient communication between publishers and "small" sets of subscribers on a two-level geo-distributed CPS network of brokers with hierarchical deployments within individual regions as outlined in Figure 2 for two regions. This solution can be adapted to existing overlay-based CPS systems characterized in Section 2.2.

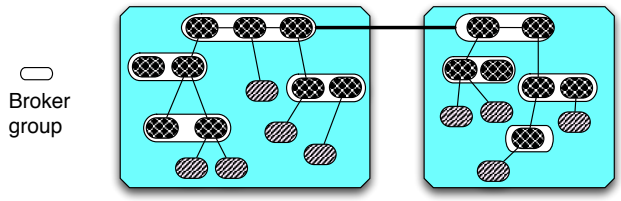


Fig. 2. Broker Hierarchies

3.1 Definition of Entourages

The range of messages published by a publisher p are identified by advertisements τ_p, which, as is customary in CPS, include the keys and the value ranges for each key. Analogously, the interest range of each subscriber or broker n is denoted by τ_n. $\tau_p \cap \tau_n$ denotes the common interest between a publisher p and a subscriber or broker n.

We define the interest match between a publisher p and a subscriber/broker n as a numerical value that represents the fraction of the publisher's messages that the subscriber/broker is interested in assuming the publisher to have an equal probability of publishing a message with any given value within its range. If the range τ_p of p is denoted by $\langle key_1, range_1\rangle, \langle key_2, range_2\rangle, ..., \langle key_x, range_x\rangle$ and $\langle key_1, range'_1\rangle, \langle key_2, range'_2\rangle, ..., \langle key_x, range'_x\rangle$ denotes the range τ_n of n, then the interest match is given by:

$$\Pi_{i=1}^{x} \frac{|range_i \cap range'_i|}{|range_i|}$$

So, the interest match is defined to be the product of the intersection of the value ranges that corresponds to the same key. If ranges that correspond to a given key have an empty intersection, then n is not interested in messages with the publishers value range for that key, hence there is zero interest match.

A publisher p and a set Φ_p of subscribers/brokers form a ψ-*close entourage* if each member of Φ_p has at least a ψ interest match with p where $0 \leq \psi \leq 1$. ψ is a parameter that defines how close the cluster is to a topic. If $\psi = 1$, each member of the cluster is interested in every message published by p, hence the cluster can be viewed as a topic.

3.2 Solution Overview

Next we describe our solution to efficient cross-cloud communication in entourages. The solution consists of three main parts which we describe in turn.

1. A decentralized protocol that can be used to identify entourages in a CPS system.
2. A mechanism to determine the maximum number K_p of direct connections a given publisher p can maintain without adversely affecting message transmission.
3. A mechanism to efficiently establish auxiliary networks termed *überlay* between publishers and their respective subscribers using information from above two.

```
1: id                                                        {ID of the broker}
2: super                                                     {ID of the parent broker}
3: subbrokers                                                {Sub-brokers}
4: subscribers                                               {Subscribers directly connected to the broker}
5: wait ← 0                                                  {# of records to be received by sub-brokers}
6: results ← ∅                                               {Results to be sent to the parent broker}
7: when RECEIVE(COUNT, p, τ_p) from id'
8:     end ← false                                           {Whether will be forwarding COUNT}
9:     for all node ∈ subbrokers ∪ subscribers do
10:        if interestMatch(τ_node, τ_p) ≥ ψ then            {Sufficient interest}
11:            if node ∈ subbrokers then
12:                SEND(COUNT, p, τ_p) to subbroker          {Forward COUNT}
13:                wait ← wait + 1                           {# of results to wait for}
14:            else
15:                results ← results ∪ {⟨node, 1⟩}           {Add node}
16:        else
17:            end ← true
18:    if |wait| + |results| = 0 then                        {No matching nodes found}
19:        end ← true
20:    if end = true then
21:        results ← ∅                                       {Resetting records; any responses discarded}
22:    reply ← false
23:    if end = true or (|results| > 1 and wait = 0) then
24:        reply ← true                                      {Send the COUNTREPLY to parent broker}
25:    if reply = true or |results| + wait > 1 then
26:        results ← results ∪ {⟨id, 0⟩}                     {Adding current broker}
27:    if reply = true then
28:        SEND(COUNTREPLY, p, results) to super             {Sending COUNTREPLY}
29: when RECEIVE(COUNTREPLY, p, results') from id'
30:    for all ⟨id'', depth⟩ ∈ results' do
31:        results ← results ∪ {⟨id'', depth + 1⟩}           {Depth + 1}
32:    wait ← wait − 1                                       {Have to wait for 1 less record}
33:    if wait = 0 then
34:        SEND(COUNTREPLY, p, results) to super             {Got all responses}
```

Fig. 3. DCI Protocol

3.3 Entourage Identification

We describe the *DCI (dynamic entourage identification) protocol* that can be used to identify entourages in a CPS-based application. The protocol assumes the brokers in region i to form a hierarchy, starting from one or more root brokers. An abstract version of the protocol is given in the Figure 3.

The protocol works by disseminating a message named COUNT along the message dissemination path of publishers. A message initiated by a publisher p contains τ_p and ψ values. Once the message reaches a root node of the publishers region, it is forwarded to each of the remote regions.

The brokers implement two main event handlers, (1) to handle COUNT messages (line 7) and (2) to handle replies to COUNT messages – COUNTREPLY messages (line 29.)

COUNT messages are embedded into advertisements and carry the keys and value ranges of the publisher. When a broker receives a COUNT message via event handler (1), it first determines the subscribers/brokers directly attached to it that have an interest match of at least ψ with the publisher p. If there is at least one subscriber/broker with a non-zero interest match that is smaller than ψ then the count message is not forwarded to any child. Otherwise the COUNT message is forwarded to all interested children. This is because children with less than ψ interest match are not considered to be direct

members of the p's entourage and yet messages published by p have to be transmitted to all interested subscribers including those with less than ψ interest match. In such a situation, instead of creating direct connections with an ancestor node and some of the descendants, we choose to only establish direct connections with the ancestor node since establishing direct connections with both an ancestor and a descendent will result in duplicate message delivery and unfair latency advantages to a portion of subscribers. A subscriber or a broker that does not forward a COUNT message immediately creates a COUNTREPLY message with its own information and sends it back to the parent.

A broker does not add its own information to the reply sent to the parent broker if the broker forwarded the COUNT message to exactly one child. This is because a broker that is only used to transfer traffic between two other brokers or a broker and a subscriber has a child that has the same interest match with p but is hop-wise closer to the subscribers. This child is a better match when establishing an entourage.

In the latter event handler (2), a broker aggregates COUNTREPLY messages from its children that have at least a ψ interest match with p, and send this information to its respective parent broker through a new COUNTREPLY message. Aggregated COUNTREPLY messages are ultimately sent to p. To stop the COUNTREPLY messages from growing indefinitely, a broker may truncate COUNTREPLY messages that are larger than a predefined size M. When truncating, entries from the lowest levels of the hierarchy are removed first. When removing an entry, entries of all its siblings (i.e., entries that have the same parent) are also removed. This is because as mentioned before, our entourage establishment protocol does not create direct connections with both a ancestor node and one of its descendants.

A subscriber or a broker may decide to respond to its parent with a COUNTREJECT message instead of a COUNTREPLY, either due to policy decisions or local resource limitations. A broker that receives a COUNTREJECT from at least one of its children will discard COUNTREPLY messages for the same publisher from the rest of its children.

As a publisher's range of values in published messages evolves, it will have to send new advertisements with COUNT messages to keep its entourage up to date. This is supported in our system Atmosphere presented in the next section by exposing an advertisement update feature in the client API.

3.4 Entourage Size

We devise a heuristic to determine the maximum number of direct connections a given publisher can maintain to its entourage without adversely affecting the performance of transmission of messages.

Factors and challenges. Capabilities of any node connected to a broker network are limited by a number of factors. A node obviously has to spend processor and memory resources to process and publish a stream of messages. The bandwidth between the node and the rest of the network could also become a bottleneck if messages are significantly large, or transmitted at a significantly high rate. This is particularly valid in a multi-tenant cloud environment. The transport protocols used by the publisher and latencies between it and the receivers could limit the rate at which the messages are transmitted.

If the implementation is done in a smart enough way, the increase in memory footprint and the increase in latency due to transport deficiencies can be minimized. The additional memory required for creating data-structures for new connections is much smaller compared to the RAM available in todays computers (note that we do not consider embedded agents with significantly low memory footprints). The latencies could become a significant factor if the transport protocol is implemented in a naïve manner, e.g., with a single thread that sends messages via TCP directly to many nodes, one by one. The effect could be minimized by using smarter implementation techniques, e.g., by using features such as multi-threaded transport layers, custom built asynchronous transport protocols, and message aggregation.

Conversely, the processor and bandwidth consumption could significantly increase with the number of unicast channels maintained by a publisher as every message has to be repeatedly transmitted over each connection and every transmission requires CPU cycles and network bandwidth.

Number of connections. First we determine the increase in processor usage of a given publisher due to establishing direct connections with subscribers or brokers. With each new direct connection, a publisher has to repeatedly send its messages along a new transport channel. So a safe worst case assumption is to assume that the amount of processing power needs to be proportional to the number of connections over which messages are transmitted.

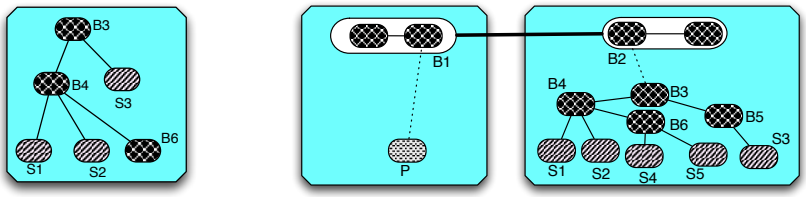
Additionally, as mentioned previously, a given publisher p will have a bandwidth quota of W_p when communicating with *remote* regions. Considering both these factors, the number of direct connections K_p which publisher p can establish can be approximated by the expression $min(\frac{1}{U_p}, \frac{W_p}{r_p \times s_p})$.

This requires the publishers to keep track of their processor utilization; in most of the operating systems, processor utilization can be determined by using system services (e.g., the `top` command in Unix). The above bound on the number of directly connected nodes is not an absolute bound, but rather a initial measure used by any publisher to prevent itself from creating an unbounded number of connections. A publisher that establishes K_p connections and needs more connections will reevaluate its processor and bandwidth usage and will create further direct connections using the same heuristic, i.e., assuming the required processor and bandwidth usage to be proportional to the number of connections established.

3.5 Überlay Establishment

We use information obtained through the techniques described above to dynamically form "over-overlays" termed *überlays* between members of identified entourages so that they can communicate efficiently and with low latency.

Graph construction. A publisher first constructs a graph data structure with the information received from the DCI protocol. This graph gives the publisher an abstract view



(a) G1 (b) G2

Fig. 4. Graph vs Overlay

of how its subscribers are connected to the brokers. There are three important differences between the graph constructed by the publisher ($G1$) and a graph constructed by globally observing the way subscribers are actually networked with the brokers ($G2$).

a. $G1$ only shows brokers that distribute the publisher's traffic to two or more sub-brokers while $G2$ will also show any broker that simply forwards traffic between two other brokers or a broker and a subscriber.
b. $G1$ may have been truncated to show only a number of levels starting from the first broker that distribute the publisher's traffic into two children while $G2$ will show all the brokers and subscribers that receive the publisher's traffic.
c. $G1$ will only show brokers/subscribers that have at least a ψ interest match with the publisher while $G2$ will show all brokers/subscribers that show interest in some of the publisher's messages.

Figures 4(a) and 4(b) show an example graph constructed by a publisher and an actual network of brokers and subscribers that will result in the graph respectively. The broker $B5$ was not included in the former due to a. above and subscribers $S4$ and $S5$ may not have been included either due to b. or c. (i.e., either because the graph was truncated after three levels or because subscriber $S4$ or $S5$ did not have at least ψ interest match with the publisher p) or simply because $S4$ or $S5$ decided to reject the COUNT message from its parent due to one of many reasons given previously.

Connection establishment. Once the graphs are established for each remote region publisher can go ahead and establish überlays. The publisher determine the number of direct connections it can establish with each remote region r (K_p^r) by dividing K_p among regions proportional to the sizes (number of nodes) of respective $G1$ graphs.

For each region r the publisher tries to decide if it should create direct connections with brokers/subscribers in one of the levels of the graph, and if so with which level. The former question is answered based on the existence of a non-empty graph. If the graph is empty, this means that none of the brokers/subscribers had at least ψ interest match with the publisher and hence forming an entourage for distributing messages of p is not viable. To answer the latter question, i.e., the level with which direct connections should be created, we compare two properties of the graph.

ad – the average distance to the subscribers.
cv – the portion of the total overlay of the region that will be covered by the selection.

By creating direct connections closer to the subscribers, the entourage will be able to deliver messages with low latency. By creating direct connections at higher levels, the direct connections will cover a larger portion of the region's broker network, hence reducing the likelihood of having to recreate the direct connections due to new subscriber joins. This is especially important in the presence high levels of churn (ch^r for region r). Additionally the publisher can create direct connections which are also bounded by the value of K_p^r for the considered region. The publisher proceeds by selecting the level to which it will establish direct connections (L_p) based on the following heuristic.

$$\frac{cv_{L_p} \times ch^r + 1}{ad_{L_p} + 1} \geq \frac{cv_l \times ch^r + 1}{ad_l + 1} \;\; \forall \; l \in \{1 ... \lfloor \log K_p^r \rfloor\}$$

Basically the heuristic determines the level which gives the best balance between the coverage and the average distance to subscribers. The importance of coverage depends on the churn of the system. Each factor of the heuristic is incremented by one so that the heuristic gives a non-zero and deterministic value when either churn or distance is zero. To measure the churn, each broker keeps track of the rate at which subscribers join/leave it. This information is aggregated and sent upwards towards the roots where the total churn of the region is determined.

If there are more than K_p^r nodes at the selected level then the publisher will first establish connections with K_p^r randomly selected nodes there. The publisher will keep sending messages through its parent so that the rest of the nodes receive the published messages. Any node that already establishes direct connections with the publisher will discard any message from the publisher received through the node's parent. Once these connections are established the publisher as mentioned previously reevaluates its resource usage and creates further direct connections as necessary.

If a new subscriber that is interested in messages from the publisher joins the system, initially it will get messages routed via the CPS overlay. The new subscriber will be identified, and a direct connection may be established in the next execution of the DCI protocol. If a node that is directly connected to the publisher needs to discard the connection, it can do so by sending a COUNTREJECT message directly to the publisher. A publisher upon seeing such a message will discard the direct connection established with the corresponding node.

4 Atmosphere

In this section, we describe Atmosphere, our CPS framework for multi-region deployments which employs the DCI protocol introduced previously. The core implementation of Atmosphere in Java has approximately 3200 lines of code.

4.1 Overlay Structure

Atmosphere uses a two-level overlay structure based on *broker* nodes. Every application node that wishes to communicate with other nodes has to initially connect to one of the

brokers which will be identified as the node's *parent*. A set of peer brokers form a *broker group*. Each broker in a group is aware of other brokers in that group. Broker-groups are arranged to form *broker-hierarchies*. Broker-hierarchies are illustrated in Figure 2. As the figure depicts, a broker-hierarchy is established in each considered region. A region can typically represent a LAN, a datacenter, or a zone within a datacenter. At the top (root) level broker-groups of hierarchies are connected to each other. The administrator has to decide on the number of broker-groups to be formed in each region and the placement of broker-groups.

Atmosphere employs subscription summarization to route messages. Each broker summarizes the interests of its subordinates and sends the summaries to its parent broker. Root-level brokers of a broker-hierarchy share their subscription summaries with each other. At initiation, the administrator has to provide each root-level group the identifier of at least one root-level broker from each of the remote regions.

4.2 Fault Tolerance and Scalability

Atmosphere employs common mechanisms for fault tolerance and scalability. Each broker group maintains a strongly consistent membership, so that each broker is aware of the live brokers within its group. A node that needs to connect to a broker-group has to be initially aware of at least one live broker (which will become the node's parent). Once connected, the parent broker provides the node with a list of live brokers within its broker-group and keeps the node updated about membership changes. Each broker, from time to time, sends heartbeat messages to its *children*.

If a node does not receive a heartbeat from its parent for a predefined amount of time, the parent is presumed to have failed, and the node connects to a different broker of the same group according to the last membership update from the failed parent. A node that wishes to leave, sends an *unsubscription* message to its parent broker. The parent removes the node from its records and updates the peer brokers as necessary.

Atmosphere can be scaled both horizontally and vertically. Horizontal scaling can be performed by adding more brokers to groups. Additionally, Atmosphere can be vertically scaled by increasing the number of levels of the broker-hierarchy. Nodes may subscribe to a broker in any level.

4.3 Flexible Communication

Atmosphere implements the DCI protocol of Section 3. To this end, each publisher sends COUNT messages to its broker. These messages are propagated up the hierarchy and once the root brokers are reached, distributed to the remote regions to identify entourages. Once suitable entourages are identified, überlays are established which are used to disseminate messages to interested subscribers with low latency.

When changes in subscriptions (e.g., joining/leaving of subscribers) arrive at brokers these may propagate corresponding notifications upstream even if subscriptions are covered by existing summaries; when arriving at brokers involved in direct connections these can notify publishers directly of changes, prompting these to re-trigger counts.

4.4 Advertisements

By wrapping it with the client library of Atmosphere the DCI protocol for publishers/-subscribers is transparent to application components, at the exception of *advertisements* which publishers can optionally issue to make effective use of direct connections.

Advertisements are supported in many overlay-based CPS systems, albeit not strictly required. Similarly, publishers in Atmosphere are not forced to issue such advertisements as Atmosphere, although effective direct connection establishment hinges on accurate knowledge of publication spaces. Atmosphere can employ runtime monitoring of published messages if necessary. For such inference, the client library of Atmosphere compares messages published by a given publisher against the currently stored advertisement and adapts the advertisement if required. When witnessing significant changes, the new advertisement is stored and the DCI protocol is re-triggered.

Note that messages beyond the scope of a current advertisement are nonetheless propagated over the direct connections in addition to the overlay. The latter is necessary to deal with joining subscribers in general as mentioned, while the former is done for performance reasons – the directly connected nodes might be interested in the message since the publisher's range of publications announced earlier can be a subset of the ranges covered by any subscriptions.

The obvious downside of obtaining advertisements only by inference is that überlay creation is delayed and thus latency is increased until the ideal connections are established. To avoid constraining publishers indefinitely to previously issued advertisements, the Atmosphere client library offers API calls to issue explicit advertisement updates. Such updates can be viewed as the publisher-side counterpart of *parametric subscriptions* [29] whose native support in a CPS overlay network have been shown to not only have benefits in the presence of changing subscriptions, but also to improve upstream propagation of changes in subscription summaries engendered via unsubscriptions and new subscriptions.

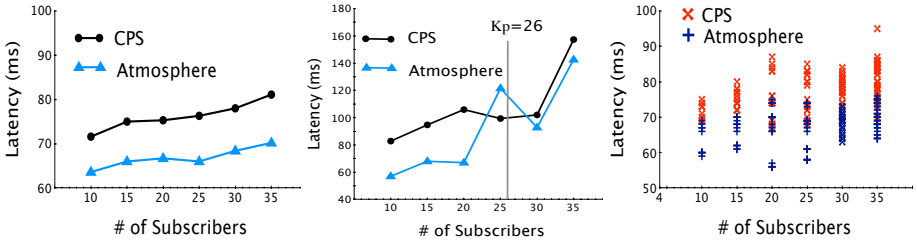
5 Evaluation

We demonstrate the efficiency and versatility of Atmosphere via several microbenchmarks and real-life applications.

5.1 Setup

We use two datacenters for our experiments, both from Amazon EC2. The datacenters are located in US east coast and US west coast respectively. From each of these datacenters we lease 10 *small* EC2 instances with 1.7GB of memory and 1 virtual core and 10 *medium* EC2 instances with 3.7GB of memory and 2 virtual cores each.

Our experiments are conducted using three publish/subscribe systems: (1) Atmosphere with DCI protocol disabled, representing a pure CPS system (referred to as CPS in the following); (2) Atmosphere with DCI protocol enabled (Atmosphere); (3) Apache ActiveMQ topic-based messaging system [21] (TPS). ActiveMQ is configured for fair comparison to use TCP just like Atmosphere and to not persist messages. All code is implemented in Java.



(a) 50 msgs/s (b) 200 msgs/s (c) All Subscribers - 50 msgs/s

Fig. 5. Latency and all Subscribers for 50 msgs/s

5.2 Microbenchmarks

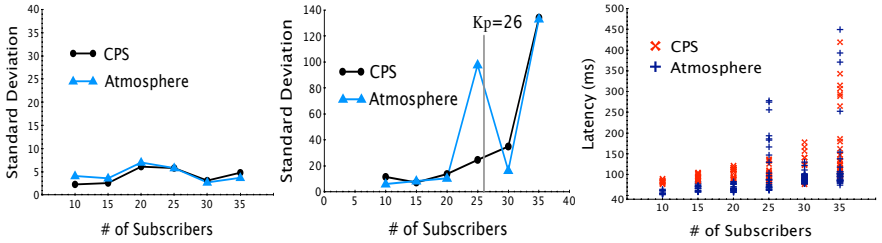
We first assess the performance benefits of Atmosphere via micro-benchmarks.

Latency. We conduct experiments to observe the message transmission latency of Atmosphere with and without DCI protocol enabled. The experiment is conducted across two datacenters and use *small* EC2 instances. A single publisher is deployed in the first datacenter, while between 10 and 35 subscribers are deployed in the second datacenter. Each datacenter maintain three root brokers.

Figures 5(a) and 5(b) show the latency for message rates 50 msgs/s and 200 msgs/s while Figures 6(a) and 6(b) show the standard latency deviations for the same rates. We separate latency from its standard deviation for clarity. Figures 5(c) and 6(c) show the average message transmission latency to individual subscribers for message rates 50 msgs/s and 200 msgs/s respectively.

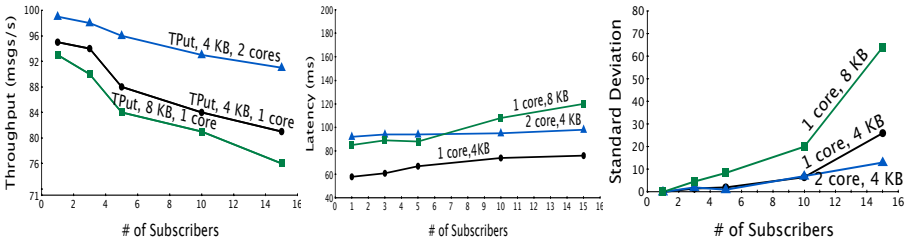
As the graphs clearly show, when the number of interested subscribers is small, maintaining unicast channels between the publisher and the subscribers pays off, even considering that the relatively slow connection to the remote datacenter is always involved, and only local hops are avoided. This helps to dramatically reduce both the average message transmission latency and the variance of latency across subscribers. For message rates 50 and 200, when the number of subscribers is 10, maintaining direct connections reduce the latency by 11% and 31% respectively.

For message rates 50 and 200, the value of K_p is determined to be 50 and 26 respectively. The Figure 5(b) and 6(b) show that both the message transmission latency and the variation of it considerably increase when the publisher reaches this limit. Also the figures show the benefit of not using the überlay after the number of subscribers exceed K_p. For example, as shown in Figure 5(b) when publisher move from maintaining a überlay with its entourage to communicating using CPS (25 to 30 subscribers) the average message transmission latency reduce by 24%. The increase in latency at 35 subscribers is due to brokers being overloaded, which can be avoided in practical systems by adding more brokers to the overlay and distributing the subscribers among them. Also note that the broker overlay used for this experiment consist of only two levels which is the case where entourage überlays exhibit least benefits.



(a) 50 msgs/s (b) 200 msgs/s (c) All Subscribers - 200 msgs/s

Fig. 6. Standard Deviation of Latency and all Subscribers for 200 msgs/s



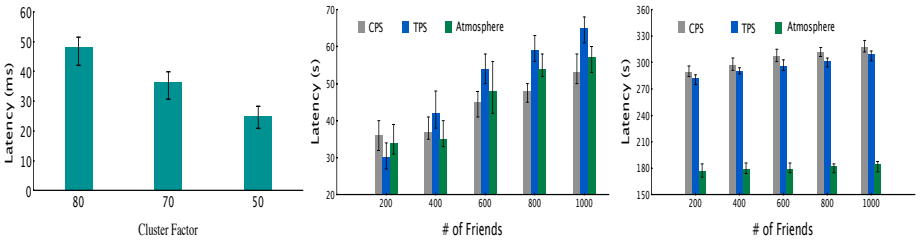
(a) Throughput (b) Latency (c) Standard Latency Deviation

Fig. 7. Effects of Resource Usage

Number of subscribers in an entourage. We conduct experiments using three publisher setups: (1) a publisher uses a *small* EC2 instance (1 core) and sends messages of size 4KB (p_1); (2) a publisher uses a *medium* EC2 instance (2 cores) and sends messages of size 4KB (p_2); (3) a publisher uses a *small* EC2 instance and sends messages of size 8KB (p_3). Subscribers and publishers are placed in two different datacenters as previously. Publishers produce at the *highest possible rate* here. Figures 7(a), 7(b), and 7(c) show how message latency, throughput, and standard latency deviation, respectively, vary for these setups as the number of subscribers changes.

The throughput of p_2 is significantly higher than that of p_1. This is expected since the rate at which messages can be transmitted increases with the processing power within the relevant confines. Interestingly though, the average message transmission latency for p_2 is higher than the average transmission latency of messages published by p_1. This suggests that the latency depends on the throughput and not directly on the processing power; the throughput itself of course depends on processing power.

The size of the transmitted messages has a substantial effect on both throughput and latency. The latter effect becomes significant as the number of subscribers increases. Additionally, Figure 7(c) shows that the variation in transmission latency can be significantly reduced by increasing the processing power of the publisher or by decreasing the size of the transmitted messages (e.g., by using techniques such as compression).



(a) ψ (b) Status (c) Friend

Fig. 8. Effect of ψ and Evaluation of our Social Network App

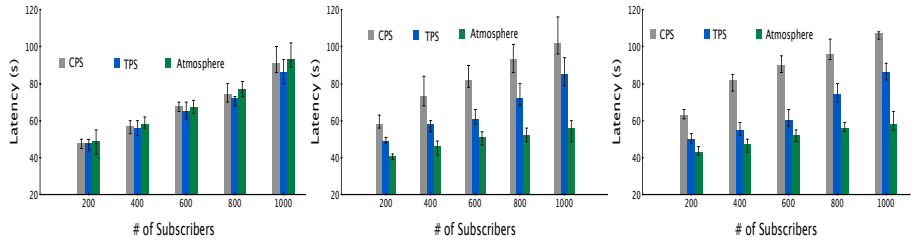
Effect of ψ. To study the effects of the clustering factor (ψ) on latency, we deploy a system of one publisher and multiple subscribers. We generated subscribers with interest ranges (of size 20) starting randomly from a fixed set of 200 interests. The publisher publishes a message to one random interest at specific intervals. Brokers are organized into a fully complete binary tree with 3 levels and 40 subscribers are connected to leaf level brokers. On this setup, latency measurements are taken with different ψ values. Figure 8(a) shows the results. When ψ is high, entourages are not created because no broker has an interest match as high as ψ. This means messages get delivered to root-level brokers which causes higher delays as the messages need to travel through all the levels in the broker network. For a lower value of ψ, an entourage is established, reducing latency.

5.3 Case Studies

We developed three test applications to show how Atmosphere can be used to make real world applications operate efficiently.

Social network. Typical IM clients attached to social networking sites support the following two operations: (1) *status* updates, in which the current status (Busy/Active/Idle or a custom message) of a user is propagated to all users in his/her friend list; (2) the ability to start a conversation with another user in the friend list. Even when explicit status updates are infrequent, IM clients automatically update user status to Idle/Active generating a high number of status updates. We developed an instant messaging service that implements this functionality either on top of Atmosphere or ActiveMQ.

Figure 8(b) shows latency measurements for status updates. Figures 8(c) shows latency measurements for a randomly selected friend. For conversations, we use actual conversation logs posted by users of Cleverbot [30]. We evaluate this type of communication on Atmosphere, pure CPS with Atmosphere, and ActiveMQ. The results show that our system is 40% faster than pure CPS and 39% faster than ActiveMQ in delivering instant messages. For delivering status messages, in the worst case, Atmosphere is on par with both systems because our system distinguishes between the communication types required for status updates and instant message exchange and dynamically forms entourage überlays for delivering instant messages only.



(a) Headlines (b) Sports (c) Weather

Fig. 9. News App Evaluation

News service. We developed an Atmosphere-based news feed application that delivers news to subscribed clients. Our news service generates two types of messages: (1) messages containing news headlines categorized according to the type of news (e.g., sports, politics, weather); (2) messages containing detailed news items of a given category. This service can also operate on top of either Atmosphere or ActiveMQ.

In Figures 9(a), 9(b), and 9(c) we explore latency of the news application for three different communication patterns. The total number of subscribers varies from 200 to 1000 with a subset of 30 subscribers interested in sports-based news and a subset of 20 subscribers interested in weather reports. We measure the average latency for delivering sports news and weather reports to these 30 and 20 subscribers. Other subscribers receive all news. Here again our system delivers sports and weather reports 35% faster than a pure CPS system and around 25% faster than ActiveMQ. This is because Atmosphere automatically creates entourages for delivering these posts.

Geo-distributed lock service. We implemented a geo-distributed lock service that can be used to store system configuration information in a consistently replicated manner for fault tolerance. The service is based on Apache ZooKeeper [23], a system for maintaining distributed configuration and lock services. ZooKeeper guarantees scalability and strong consistency by replicating data across a set of nodes called an *ensemble* and by executing consensus protocols among these nodes.

We maintain a ZooKeeper ensemble per datacenter and interconnect the ensembles (i.e., handle the application requests over the ensembles) using Atmosphere. We compare the Atmosphere-based lock service with a naïve distributed deployment of ZooKeeper (*Distributed*) where all ZooKeeper nodes participated in a single geo-distributed ensemble. This experiment uses three datacenters. For each run, a constant number of ZooKeeper servers are started at each datacenter. Our system provides the same guarantees as naïve ZooKeeper except the rare scenario of datacenter failure (in this case *Atmosphere* deployment may loose a part of the stored data).

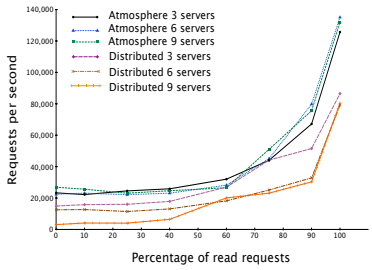


Fig. 10. Lock Service

We vary the percentage of read requests and observed the maximum load the systems could handle with 3, 6, and 9 total nodes forming ensembles. Figure 10 shows the results of the experiment for Atmosphere-based (*Atmosphere*) deployment and a distributed deployment of ZooKeeper where all ZooKeeper nodes participated in a single geo-distributed ensemble (*Distributed*). The figure shows that by establishing überlays, *Atmosphere* deployment can handle a larger load.

These case studies illustrate the general applicability of Atmosphere.

6 Conclusions

Developing and composing applications executing in the cloud-of-clouds requires generic communication mechanisms. Existing CPS frameworks — though providing generic communication abstractions — do not operate efficiently across communication patterns and regions, exhibiting large performance gaps to more specific solutions. In contrast, existing simpler TPS solutions cover fewer communication patterns but more effectively – in particular scenarios with few publishers and many subscribers which are wide-spread in cloud-based computing.

We introduced the DCI protocol, a mechanism that can be used to adapt existing solutions to efficiently support different patterns, and presented its implementation in Atmosphere, a scalable and fault-tolerant CPS framework suitable for multi-region-based deployments such as cross-cloud scenarios. We illustrated the benefits of our approach through different experiments evaluating multi-region deployments of Atmosphere.

We are currently working on complementary techniques that will further broaden the range of efficiently supported communication patterns, for example the migration of subscribers between brokers guided by resource usage on these brokers. Additionally we are exploring the use of Atmosphere as the communication backbone for other systems including our Rout [31] framework for efficiently executing Pig/PigLatin workflows in geo-distributed cloud setups and our G-MR [32] system for efficiently executing sequences of MapReduce jobs on geo-distributed datasets. More information about Atmosphere can be found at `http://atmosphere.cs.purdue.edu`.

References

1. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog Computing and its Role in the Internet of Things. In: MCC. (2012)
2. Bessani, A., Correia, M., Quaresma, B., André, F., Sousa, P.: DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. In: EuroSys. (2011)
3. Grivas, S., Uttam, K., Wache, H.: Cloud Broker: Bringing Intelligence into the Cloud. In: CLOUD. (2010)
4. Li, M., Ye, F., Kim, M., Chen, H., Lei, H.: A Scalable and Elastic Publish/Subscribe Service. In: IPDPS. (2011)
5. Deering, S., Cheriton, D.: Multicast Routing in Datagram Internetworks and Extended LANs. ACM TOCS **8**(2) (May 1990) 85–110
6. Vigfusson, Y., Abu-Libdeh, H., Balakrishnan, M., Birman, K., Burgess, R., Chockler, G., Li, H., Tock, Y.: Dr. Multicast: Rx for Data Center Communication Scalability. In: EuroSys. (2010)

7. Amazon Inc.: Amazon SNS. http://aws.amazon.com/sns/ (2012)
8. Apache Software Foundation: Apache BookKeeper: Hedwig. http://zookeeper.apache.org/bookkeeper/
9. Kreps, J., Narkhede, N., Rao, J.: Kafka: a Distributed Messaging System for Log Processing. In: NetDB. (2011)
10. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Achieving Scalability and Expressiveness in an Internet-scale Event Notification Service. In: PODC. (2000)
11. Pietzuch, P., Bacon, J.: Hermes: A Distributed Event-Based Middleware Architecture. In: ICDCSW. (2002)
12. Fiege, L., Gärtner, F.C., Kasten, O., Zeidler, A.: Supporting Mobility in Content-Based Publish/Subscribe Middleware. In: Middleware. (2003)
13. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching Events in a Content-based Subscription System. In: PODC. (1999)
14. Li, G., Hou, S., Jacobsen, H.A.: A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems Based on Modified Binary Decision Diagrams. In: ICDCS. (2005)
15. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-Value Store. In: SOSP. (2007)
16. Das, S., Agrawal, D., Abbadi, A.E.: G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In: SOCC. (2010)
17. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. CACM **51**(1) (January 2008) 107–113
18. Apache Software Foundation: Apache HDFS. http://hadoop.apache.org
19. Voulgaris, S., Riviere, E., Kermarrec, A.M., van Steen, M.: Sub-2-Sub: Self-Organizing Content-Based Publish Subscribe for Dynamic Large Scale Collaborative Networks. In: IPTPS. (2006)
20. Tariq, M., Koldehofe, B., Koch, G., Rothermel, K.: Distributed Spectral Cluster Management: A Method for Building Dynamic Publish/Subscribe Systems. In: DEBS. (2012)
21. Apache Software Foundation: Active MQ. http://activemq.apache.org/
22. IBM Inc.: Websphere MQ. http://www-01.ibm.com/software/integration/wmq/
23. Apache Software Foundation: Apache ZooKeeper. http://hadoop.apache.org/zookeeper/
24. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and Evaluation of a Wide-Area Event Notification Service. ACM TOCS **19**(3) (August 2001) 332–383
25. P. Triantafillou and A. A. Economides: Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems. In: ICDCS. (2004)
26. Majumder, A., Shrivastava, N., Rastogi, R., Srinivasan, A.: Scalable Content-Based Routing in Pub/Sub Systems. In: INFOCOM. (2009)
27. Kazemzadeh, R.S., Jacobsen, H.A.: Publiy+: A Peer-Assisted Publish/Subscribe Service for Timely Dissemination of Bulk Content. In: ICDCS. (2012)
28. Agarwal, S., Dunagan, J., Jain, N., Saroiu, S., Wolman, A., Bhogan, H.: Volley: Automated Data Placement for Geo-Distributed Cloud Services. In: NSDI. (2010)
29. Jayaram, K.R., Eugster, P., Jayalath, C.: Parametric Content-Based Publish/Subscribe. ACM TOCS **31**(2) (May 2013) 4:1–4:52
30. Carpenter, R.: Cleverbot. http://cleverbot.com/
31. Jayalath, C., Eugster, P.: Efficient Geo-Distributed Data Processing with Rout. In: ICDCS. (2013)
32. Jayalath, C., Stephen, J., Eugster, P.: From the Cloud to the Atmosphere: Running MapReduce across Datacenters. IEEE TC - Special Issue on Cloud of Clouds, to appear