Developing an Elastic and Dependable Cloud-Based Content Storage System

K.VENKATA RAMANA
Professor
Author1 email

KRISHNA SWATHI
Assistant Professor
swathi.1992h@gmail.com

VUTTI NARAHARI

Assistant Professor
narahariv.alts@gmail.com

Abstract: Publish/subscribe systems applied as a service inside a cloud computing architecture provide flexibility and simplicity in the composition of distributed applications. The delivery of appropriate services in a distributed computing infrastructure is an urgent challenge. Existing publish/subscribe systems face a hurdle as a result of the dynamic variations in the pace of live content arrival for large-scale subscriptions. This article introduces the ESCC (Elastic and Scalable Content based Cloud Pub/Sub System) approach, which provides a design framework for an elastic and scalable Content-based publish/subscribe system that uses a one hop lookup overlay to minimize latency in a cloud computing environment. ESCC adjusts the server scale dynamically based on the churn demands. ESCC achieves a high throughput rate relative to other types of workloads.

Key words: Publish / Subscribe, Cloud storage, Scalable, content based retrieval, subscription

I. Introduction

Cloud computing is a new paradigm with shifting definitions, but for the purposes of this research, we can use the term "virtual infrastructure" to describe a system that will use the Internet or "large-scale private networks" to deliver shared information and communication technology services to "multiple external users" via a "cloud" on the internet. [1] There is no need for a computer user to understand technology or to be the owner of the infrastructure in order to use information technology services, such as apps, servers, and data storage. To understand cloud computing, an electrical computing grid analogy would be helpful. Consumers just utilize the resources without taking ownership or operational responsibility of them; a power company maintains and owns the infrastructure; a distribution firm distributes the energy. [2]. Networked storage space and computing resources may be gained via this subscription-based service. Consider our experience with email while thinking about cloud computing. Emergency applications have drawn more attention as the need for real-time data transmission grows more important in various industries, such as stock quotation distribution, earthquake monitoring [1], emergency weather warning [2], smart transportation systems [3], and social networks. Two patterns may be seen recently in the growth of emergency apps. One is the abrupt adjustment in the arrival rate of live material. As an example, consider ANSS [1], whose goal is to provide emergency responders accurate and real-time seismic information. In many emergency applications, the publish/subscribe (pub/sub) paradigm is a crucial technique for asynchronous data distribution. It dissociates emergency application senders and receivers from one another in place, time, and synchronization [5], allowing a pub/sub system to scale up to enormous sizes without any noticeable hiccups. Traditional pub/sub systems, however, have several issues. Prior to scaling out to extremely big sizes, the system must first ensure real-time event matching capacity. Consider Facebook, where there are billions of users and 684,478 new pieces of material are posted there on average every minute [6]. To obtain a high performance-price ratio, the system must, secondly, be elastic to the abrupt change in incoming event rate. This is due to the fact that if a certain number of servers are deployed in response to a rapid change in the pace of incoming events, many servers will be in the idle states and will transmit few messages the majority of the time. The service must, thirdly, be resilient to server outages. In emergency applications, a significant number of computers and connections may become instantly inaccessible as a result of hardware malfunctions or operator errors, resulting in the loss of events and subscriptions. When there is an earthquake, sensors produce millions of signals in a short period of time, but just a small number of events if there isn't. The large-scale subscriptions' skewness is the other. In other words, a significant portion of subscribers exhibit comparable interests. As an example, the dataset [4] of 297 K Facebook users reveals that the top 100 topics together have more than 1.1 million members, whereas 71% of subjects have no more than 16 subscribers. The majority of P2P-based systems, in contrast, [13] do not provide specialised brokers. The P2P-based overlay [6] that all nodes are a part of serves as both a publisher and a subscriber. Multihop routing is used to forward all events and subscriptions. In a multicast group or in a rendezvous node, the subscribers belonging to the same subspace of the whole content space are then grouped. The multi-hop routing may result in excessive latency and traffic overhead as the arrival event rate increases. Scalability is restricted by the possibility of imbalanced demand on the multicast groups or rendezvous nodes caused by a significant number of skewed subscriptions. Additionally, since node behavior is unpredictable, it is challenging to offer elastic service in P2P-based systems. Current pub/sub systems can't effectively handle all of the aforementioned issues. All publishers and subscribers in broker-based pub/sub systems [7–14] are directly linked to a collection of servers known as brokers. It is usual practice to duplicate subscriptions to all brokers or a subset of brokers so that each broker may match events and transmit them to the interested subscribers. However, duplicating subscriptions necessitates that each event be matched many times against the same subscriptions, which results in high matching latency and poor scalability when a huge volume of events and subscriptions arrive. Additionally, offering elastic service to accommodate shifting demands is challenging. This is due to the fact that these systems often over-provision brokers in an effort to lower their loads and lack the financial incentives to dial down the number of brokers during non-peak hours.

2. Related Work

This section presents some elasticity solutions implemented in IaaS clouds. In general, most public cloud providers offer some elasticity feature, from the most basic, to more elaborate automatic solutions. In turn, the solutions developed by academy are similar to those provided by commercial providers, but include new techniques and methodologies for elastic provisioning of resources. Amazon Web Services [14], one of the most traditional cloud providers, offers a replication mechanism called AutoScaling, as part of the EC2 service. The solution is based on the concept of Auto Scaling Group (ASG), which consists of a set of instances that can be used for an application. Amazon Auto-Scaling uses an automatic-reactive approach, in which, for each ASG there is a set of rules that defines the instances number to be added or released. The metric values are provided by CloudWatch monitoring service, and include CPU usage, network traffic, disk reads and writes. The solution also includes load balancers that are used to distribute the workload among the active instances.

GoGrid [10] and Rackspace [11] also implement replication mechanisms, but unlike Amazon, does not have native automatic elasticity services. Both providers offer API to control the amount of virtual machines instantiated, leaving to the user the implementation of more elaborate automated mechanisms. To overcome the lack of automated mechanisms, tools such as RightScale [6] and Scalr [7] have been developed. RightScale is a management platform that provides control and elasticity capabilities for different public cloud providers (Amazon, Rackspace, GoGrid, and others) and also for private cloud solutions (CloudStack, Eucalyptus and OpenStack). The solution provides automatic-reactive mechanisms based on an Elasticity Daemon whose function is to monitor the input queues, and to launch worker instances to process jobs in the queue. Different scaling metrics (from hardware and applications) can be used to determine the number of worker instances to launch and when to launch these instances. Scalr is an open-source project whose goal is to offer elasticity solutions for web applications that supports several clouds, such as, Amazon, Rackspace, Eucalyptus and Cloudstack. Currently supports Apache and ngnix, MySQL database, PostgreSQL, Redis and MongoDB. Likewise RightScale, the operations use hardware and software monitoring metrics to trigger actions. A more comprehensive elasticity solution is provided by OnApp Cloud [5], a software package for IaaS cloud providers. According to its documentation, it is possible implement replication and redimensioning of VM's, allowing changes in virtual environments manually or automatically, using user-defined rules and metrics obtained by the monitoring system.In order to take full advantage of the elasticity provided by clouds, it is necessary more than an elastic infrastructure. It is also necessary that the applications have the ability to dynamically adapt itself according to changes in its requirements. In general, applications developed in PaaS clouds have implicit elasticity. These clouds provide execution environments, called containers, in which users can execute their applications without having to worry about which resources will be used. In this case, the cloud manages automatically the resource allocation, so developers do not have to constantly monitor the service status or interact to request more resources [8], [9]. An example of PaaS platform with elasticity support is Aneka [10]. In Aneka, when an application needs more resources, new container instances are executed to handle the demand, using local or public cloud resources. There are exceptions, such as Microsoft Azure [12] in which the user must define the resources used by applications. Some academic works have presented elasticity mechanisms for applications, with the main objective of enabling the development of flexible and adaptable applications for cloud environments.

Neamtiu [13] described Elastin, a framework that comprises a compiler and a runtime environment, whose goal is to convert inelastic programs into elastic applications. The idea behind Elastin is the use of a compiler that combines diverse program versions into a single application that can switch between configurations at runtime, without shutting down the application. The executable binary file stores several configurations, each one for a given scenario. The choice of which configuration should be used is defined by the user and can be changed at runtime. Vijayakumar et al. [9] presented an elasticity mechanism for streaming applications. The proposal consists in to adapt the CPU resources of the virtual machine in accordance with the data streams. The streaming application consists of a pipeline of several stages, each one allocated individually in a virtual machine. The elasticity mechanism compares the input and output flow at each stage, and if there is a bottleneck, increases the percentage of physical CPU allocated to the virtual machine that hosts the affected stage. Knauth&Fetzer [4] also addressed streaming applications, but focusing energy consumption reduction. The proposed solution employs virtual machine migration and consolidation to provide elasticity. The basic idea is to start each application stage inside a virtual machine. When load is minimal, all virtual machines are consolidated into a minimal set of physical machines. When the load increases, virtual machines are migrated to other servers, until each physical server hosts a single virtual machine. Rajan et al. [14] presented Work Queue, a framework for the development of master-slave elastic applications. Applications developed using Work Queue allow adding slave replicas at runtime. The slaves are implemented as executable files that can be instantiated by the user on different machines on demand. When executed, the slaves communicate with the master that coordinates task execution and the data exchange.

3. PROPOSED SCHEME:

3.1 ESCC Technique: Elastic and Scalable Content based Cloud Pub/Sub System

We initially provide a distributed two-layer pub/sub framework built on the cloud computing environment in order to achieve scalable and elastic total order in content-based pub/sub systems. The matching layer and the delivery layer are the two main layers of the framework.

The delivery layer is in charge of sorting events in accordance with the total order semantics and distributing them to interested subscribers. The matching layer is in charge of matching events against subscriptions and forwarding events with matched subscribers to the delivery layer. The size of subscribers, subscription and event distributions, data complexity, and event arrival rate are the major variables limiting the scaling of the matching service. The arrival rate of events and the likelihood of ordering conflicts are the key variables limiting the scalability of the complete ordering service. Therefore, if both services are detached, we may dynamically increase the capacity of either the event matching service or the entire ordering service depending on different workload characteristics.

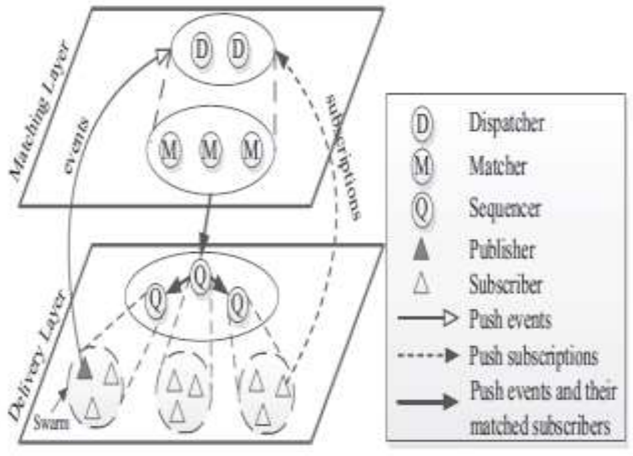

Figure 1.ESCC Architecture

3.2 Matching Layer

The delivery layer receives events with their matched subscribers once the matching layer has matched events against subscriptions. We use the SREM approach at this layer to provide high matching throughput, as illustrated in Figure 1. In SREM, the content space is partitioned into many hypercubes that are each under the control of a single server using a hierarchical multi-attribute space partitioning mechanism known as HPartition. The matching process pairs subscriptions and events that belong to the same hypercube, considerably reducing the matching latency. Additionally, ESCC proposes a performance-aware detection method (named PDetection) to adaptively change the size of servers depending on the workload churn.

ISSN: 2278-4632
Vol-12 Issue-01 No.01 January 2022

3.2 Delivery Layer

The delivery layer is in charge of providing all ordering events to interested subscribers. The primary innovation of this layer is a performance-aware provisioning method and a prior graph construction mechanism (PGBuilder) (called PProvision). The first tries to scalably lower the overall order latency. Subscribers in PGBuilder are split up into many groups, each of which is run by a different server. In other words, complete order conflicts between events may be detected concurrently by all PGBuilder servers, which significantly lowers delivery delay. In order to swiftly identify non-conflicting events and distribute them in parallel, each PGBuilder server creates preceding graphs among arrival events. Additionally, PGBuilder offers a number of dynamical maintenance tools to guarantee dependable delivery.

Algorithm 1 :PGBuilder

```
Input: New Arrival event 'e'
Q: The global queue of sequencer
C: Cluster ,CPL : Cluster preceding list
e: Direct, DPL: Direct Preceding List

    1.  Initialize cluster C with Q
    2.  if C==0 then initialize C as new
        Cluster
    3.  Process the list L as CPL and if size
        (CPL) != 0 the
    4.  addDPL.get(Q).
    5.  inti =0
    6.  while (i<tmpList.Size()) do
    7.  e = tmpList.get(i)
    8.  for(int k =0; k<e.DPL.size(); K++)
        do
    9.  tempE = e.DPL.get(K);
    10. if( ! tmpList.contains(tempE) ) then
    11. tmpList.add(tempE)
    12. i++
```

First, each sequencer dispatches arrival events into various, distinct clusters, and a global queue oversees all of these clusters (GQ). The tail cluster is then given e. The GQ naturally creates conflicts between clusters, and each cluster may be seen as as a sliding window. Therefore, just the head cluster is processed by each sequencer. The head cluster is removed from the global queue when the events of the head cluster have been sent to each of their associated subscribers, and the sequencer then processes the next head cluster. As a result, each new arrival event only has to look for conflicts with GQ events in the tail cluster, not with all GQ events. It significantly lowers the conflict detection latency independent of the event arrival rate by grouping events into several clusters.

3.3 Delivery Strategy

There are primarily two responsibilities in the ESCC technique: subscribers and sequencers. We will examine how to maintain continuous and effective total ordering under dynamic networks as the joining or departure of both roles may significantly degrade total ordering performance.

i. Subscription:

Remember that in our system, as seen in Fig. 1, each subscriber transmits their subscriptions to one of the dispatchers. According to the consistent hashing method, when a new subscriber enters the system, it is sent to the sequencer whose hash value is closest to its own value. The sequencer waits until it has received all acknowledgements from subscribers of the previous event before sending the subsequent event to guarantee dependable delivery. Sequencers must thus get the most recent perspective of its local subscribers. Otherwise, excessive delivery delay might result from waiting for acknowledgements from unsubscribed subscribers.

Algorithm 2: Event Delivery

```
Input:  e: Delivering event
/* C.CPL: the cluster preceding list of cluster C*/
/* e.DPL: the  direct predecessor list of e */
    1. foreach (subscriber s in Dest(e)) do
    2. send(s,e);
    3. C.CPL. remove(e);
    4. if C.CPL.isEmpty() then
    5. ConflictResolution()
    6. else
    7. foreach(event e in e,DSL) do
    8. e.DPL.remove(e);
    9. Delivery(e)
```

ii. Sequencer:

The root sequencer appoints each new sequencer, one by one, to be a child of the existing sequencers when a number of new sequencers enter the system. Accordingly, depending on the consistent hashing approach, each retiring sequencer should be able to determine if every local subscriber has to be moved to one of the new sequencers.

4. Performance Analysis

The design and execution of the ESCC prototype as well as a performance study of the suggested framework are presented in this part. We used object-oriented middleware to build the prototype in a modular and portable way, letting users concentrate on the application logic rather than interacting with low-level network programming interfaces. Each matcher assesses its waiting time every 200 ms to assess the flexibility and scalability. We set the timeout intervals Tout and T′ for adding and deleting matchers, respectively, to 10 seconds. In other words, ESCC gathers all fresh matchers and failed matchers up to the continuous arrival or failure interval of two matchers surpasses 10 s. Along with that, Ds hypothesizes that a matcher will not succeed if the continuous arrival gap between two heartbeat signals is more than 10 s. In order to adapt to both linear and instantaneously rising event arrival rates, the ESCC must first modify the number of matchers, as illustrated in Figure 2. Following that, the procedure demonstrates how the ESCC's elasticity may adjust to both linear and instantaneous decreasing event arrival rates.

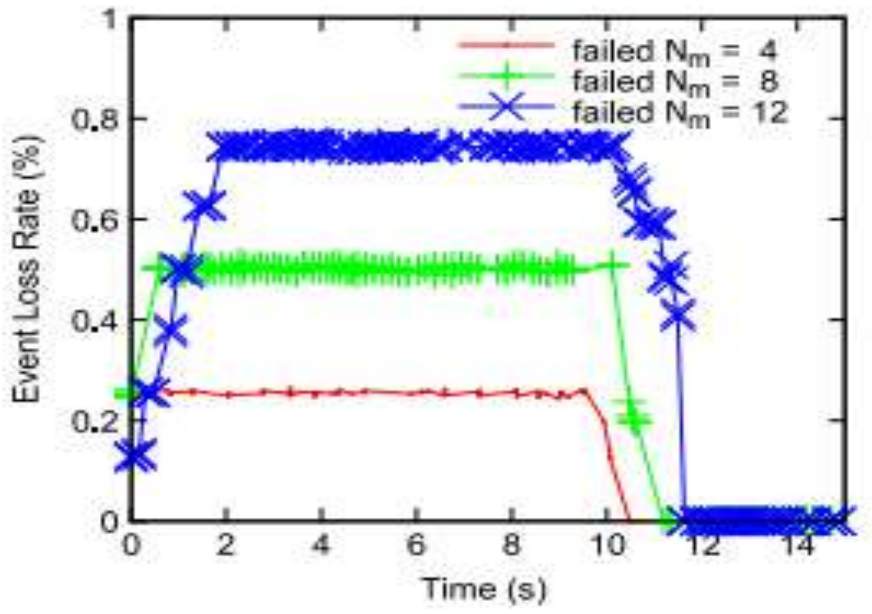

Figure 2. The changing of event loss rate with a number of matcher's failure.

5. Conclusion

In this study, ESCC, a brand-new scalable and elastic event matching method for attribute-based pub/sub systems, is introduced. To minimize clustering latency in the cloud computing environment, ESCC uses a one-hop lookup overlay. The ESCC achieves scalable clustering of subscribers and matches each event on a cluster using a hierarchical multi-attribute space partitioning approach. The system may modify the size of matchers in response to shifting workloads thanks to the performance-aware detection approach. Our analytical and experimental findings suggest that ESCC has a much greater matching rate and better load balancing with various workload characteristics when compared to the current cloud-based pub/sub systems. Additionally, ESCC responds to unexpected workload changes and server outages with little latency and traffic overhead.

References:

[1] M. Gjoka, M. Kurant, C.T. Butts, A. Markopoulou, Walking in Facebook: a case study of unbiased sampling of OSNs, in: International Conference on Computer Communications, INFOCOM, 2010, pp. 1–9.

[2] P.T. Eugster, P. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Computing Surveys (CSUR) 35 (2) (2003) 114–131.

[3] Datacreatedperminite. URL: http://www.domo.com/blog/2012/06/ how-much-data- is-created-every-minute/?dkw=socf3/.

[4] P. Pietzuch, J. Bacon, Hermes: a distributed event-based middleware architecture, in: 22nd International Conference on Distributed Computing Systems Workshops, 2002.

[5] F. Cao, J.P. Singh, Efficient event routing in content-based publish/subscribe service network, in: International Conference on Computer Communications, INFOCOM, 2004.

[6] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, D.C. Sturman, An efficient multicast protocol for content-based publish–subscribe systems, in: IEEE International Conference on Distributed Computing Systems, ICDCS, 1999, pp. 262–272.

[7] F. Cao, J.P. Singh, Medym: match-early with dynamic multicast for contentbased publish–subscribe networks, 2005, pp. 292–313.

[8] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, D. Sturman, Exploiting IP multicast in content-based publish–subscribe systems, in: IFIP/ACM International Conference on Distributed Systems Platforms, 2000, pp. 185–207.

[9] A. Riabov, Z. Liu, J.L. Wolf, P.S. Yu, L. Zhang, Clustering algorithms for content-based publication–subscription systems, in: IEEE 22nd International Conference on Distributed Computing Systems, ICDCS, 2002, pp. 133–142.

[10] A. Carzaniga, Architectures for an event notification service scalable to widearea networks, Ph.D. Thesis, POLITECNICO DI MILANO, 1998.

[11] Y.-M. Wang, L. Qiu, C. Verbowski, D. Achlioptas, G. Das, P.-Å Larson, Summarybased routing for content-based event distribution networks, Computer Communication Review 34 (5) (2004) 59–74.

[12] A. Carzaniga, M.J. Rutherford, A.L. Wolf, A routing scheme for content-based networking, in: IEEE International Conference on Computer Communications, INFOCOM, 2004.

[13] W.W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, A.P. Buchmann, A peer-topeer approach to content-based publish/subscribe, in: Proceedings of the 2nd International Workshop on Distributed Event-Based Systems, 2003, pp. 1–8.

[14] I. Aekaterinidis, P. Triantafillou, Pastrystrings: a comprehensive content-based publish/subscribe DHT network, in: IEEE 26nd International Conference on Distributed Computing Systems, ICDCS, 2006