Industrial Engineering Journal
ISSN: 0970-2555
Volume : 50, Issue 8, August 2021

Building a Reliable and Flexible Cloud-Based Content Storage System

[1]Dr. Rajagopal Ramasamy, [2]Malla Sowmya, [3]Dandu Srinivas
[1]Associate Professor, Department of CSE, Narsimha Reddy Engineering College, Secunderabad, Telangana
[2,3]Assistant Professor, Department of CSE, Narsimha Reddy Engineering College, Secunderabad, Telangana

Abstract: In a cloud computing architecture, publish/subscribe systems used as a service provide flexibility and ease of use in the construction of distributed applications. An pressing problem is how to supply the proper services in a distributed computing architecture. Because of the dynamic fluctuations in the rate of live content arrival for large-scale subscriptions, existing publish/subscribe systems encounter a challenge. In order to reduce latency in a cloud computing environment, this article proposes the ESCC (Elastic and Scalable Content based Cloud Pub/Sub System) method, which offers a design framework for an elastic and scalable content-based publish/subscribe system. Based on the churn requirements, ESCC dynamically modifies the server scale. Compared to other workload types, ESCC achieves a high throughput rate

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

I. Introduction

For the purposes of this study, we can refer to 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 as "virtual infrastructure" despite the fact that "cloud computing" is a new paradigm with shifting definitions. [1] To utilize information technology services, such as applications, servers, and data storage, a user of a computer does not need to be knowledgeable about technology or the owner of the infrastructure. An analogy with the electrical computing grid might be useful to comprehend cloud computing. The infrastructure is maintained and owned by a power company, and the energy is distributed by a distribution business. Consumers only use the resources without accepting ownership of or operating responsibility for them. [2]. This subscription-based service offers networked storage and computer resources. When considering cloud computing, keep in mind our experience with email. As the need for real-time data transmission increases across a range of sectors, including stock quotation dissemination, earthquake monitoring [1, earthquake warning [2, emergency weather warning [3, smart transportation systems [4], and social networks, emergency applications have attracted increased attention. Recently, the development of emergency applications may have followed two trends. One is the sudden change in the pace at which live stuff arrives. Consider ANSS [1], whose objective is to provide emergency responders accurate and real-time seismic information, as an example. A key method for asynchronous data delivery in many emergency applications is the publish/subscribe (pub/sub) paradigm. The ability of a pub/sub system to grow out to vast volumes without observable glitches is made possible by the dissociation of emergency application senders and receivers from one another in location, time, and synchronization [5]. However, there are significant problems with conventional pub/sub systems. The system must first confirm its ability for real-time event matching

before scaling up to very large capacities. Take Facebook, for example, which has billions of members and where 684,478 new items are uploaded on average every minute [6]. A second need for a good performance-price ratio is that the system be elastic to a sudden change in the rate of incoming events. This is because many servers will be in the idle states and will only transmit a small number of messages the bulk of the time if a particular number of servers are deployed in response to a sudden change in the speed of incoming events. The third requirement is that the service be robust to server failures. In emergency applications, a sizable number of computers and connections may suddenly become inoperable due to operator mistake or hardware failure, leading in the loss of events and subscribers. When there is an earthquake, sensors emit millions of signals in a short amount of time, but when there isn't, they only produce a few events. The other is the skewness of the large-scale subscriptions. To put it another way, a significant proportion of subscribers have similar interests. For instance, the dataset [4] of 297 K Facebook users shows that just 16% of topics have more than 16 subscribers, but the top 100 topics together have more than 1.1 million members. In contrast, most P2P-based systems [13] don't provide specialized brokers. All nodes are subscribers to the P2P-based overlay [6], which also acts as a publisher. All events and subscriptions are sent via multihop routing. The subscribers from the same subspace of the whole content space are then aggregated, either in a multicast group or a rendezvous node. As the rate of arrival events rises, the multi-hop routing could provide too much delay and traffic overhead. The potential for uneven demand on the rendezvous nodes or multicast groups as a result of a large number of skewed subscriptions limits scalability. Furthermore, it is difficult to provide elastic service in P2P-based systems due to node behavior being unpredictable. The problems listed above can't all be resolved by current pub/sub systems. In broker-based pub/sub systems [7–14], every publisher and subscriber has a direct connection to the brokers, which are a group of servers. In order for each broker to match events and send them to the interested subscribers, it is customary to duplicate subscriptions to all brokers or a selection of brokers. However, duplicate subscriptions require that each event be matched against the same subscriptions several times, which has a negative impact on scalability when a large number of events and subscriptions are received. Providing stretchable service to meet changing needs is also difficult. These systems often overprovision brokers in an attempt to reduce their loads since they lack the financial incentives to reduce the number of brokers during non-peak hours.

2. Related Work

Some elasticity methods used in IaaS clouds are presented in this section. The majority of public cloud services, from the most basic to more complex automation systems, generally provide some kind of elasticity capability. The solutions created by the academy, in turn, are comparable to those offered by commercial suppliers but contain fresh methods and strategies for resource elasticity providing. One of the oldest cloud service providers, Amazon Web Services [14], includes AutoScaling as part of the EC2 service, which is a replication technique. The answer is built on the idea of an Auto Scaling Group (ASG), which is a collection of instances that may be utilized for an application. With Amazon Auto-Scaling, the number of instances that need to be added or released is determined by a set of criteria that are automatically applied to each ASG. The Cloud Watch monitoring service provides the metric data, which include CPU use, network traffic, disk reads and writes. Load balancers are another component of the system, which are utilized to spread the burden across the running instances.Replication techniques

are also implemented by GoGrid [10] and Rackspace [11], although unlike Amazon, they lack native automated elasticity services. Both suppliers give APIs to limit the number of virtual machines created, leaving it up to the customer to construct more complex automated systems. Tools like RightScale [6] and Scalr [7] have been created to address the absence of automated procedures. RightScale is a management platform that offers flexibility and control for both private cloud solutions like CloudStack, Eucalyptus, and OpenStack as well as for various public cloud service providers like Amazon, Rackspace, GoGrid, and others. The solution offers automatic-reactive mechanisms built on an Elasticity Daemon, whose purpose it is to keep an eye on input queues and start worker instances to handle queued tasks. The quantity and timing of worker instance launches may be determined using several scaling metrics (from hardware and applications). Aiming to provide elasticity solutions for web applications that support a number of clouds, including Amazon, Rackspace, Eucalyptus, and Cloudstack, Scalr is an open-source project. now supports MySQL, PostgreSQL, Redis, MongoDB, Apache, and Nginx. The operations, like RightScale, leverage hardware and software monitoring data to initiate actions. OnApp Cloud [5], a software solution for IaaS cloud providers, offers a more complete elasticity solution. Its description states that replication and resizing of VMs may be implemented, enabling manual or automated modifications to virtual environments using user-defined rules and data collected by the monitoring system. More than an elastic infrastructure is required in order to fully benefit from the flexibility offered by clouds. Additionally, the apps must be capable of dynamically adapting to changes in their needs. Applications created on PaaS clouds often have inherent flexibility. Users may run their programs in these clouds' "containerized" execution environments without worrying about what resources would be consumed. In this instance, the cloud maintains the resource allocation automatically, saving developers from having to continually check on the health of the service or interact to request extra resources [8], [9]. Aneka [10] is an example of a PaaS platform that supports elasticity. Aneka uses local or public cloud resources to run additional container instances when an application requires extra resources. There are also situations where the user must specify the resources that apps require, like Microsoft Azure [12]. In order to make it possible to create flexible and adaptive applications for cloud settings, certain academic studies have introduced elasticity techniques for applications.Elastin, a framework that consists of a compiler and a runtime environment, was defined by Neamtiu [13]. Its purpose is to transform inelastic programs into elastic applications. The concept behind Elastin is the use of a compiler to integrate many program versions into a single program that can transition between configurations at runtime without shutting down the program. Each setting in the executable binary file is for a certain circumstance. The user decides which configuration should be utilized, and runtime changes may be made. For streaming applications, Vijayakumar et al. [9] suggested an elasticity method. The suggestion is to modify the virtual machine's CPU resources in line with the data streams. The streaming application is made up of a pipeline with many phases, each of which is allotted its own virtual machine. When there is a bottleneck, the elasticity mechanism analyzes the input and output flow at each step and raises the proportion of physical CPU allotted to the virtual machine hosting the bottlenecked stage. In their discussion of streaming applications, Knauth&Fetzer [4] put the emphasis on lowering energy usage. In order to offer flexibility, the suggested approach makes use of virtual machine consolidation and migration. The fundamental concept is to launch each step of the program within a virtual machine. All virtual computers are combined into a small number of real machines when demand is at its lowest. Virtual machines are moved to different servers as the demand rises, until only one virtual machine is hosted by each physical

server at that point. Work Queue, a framework for creating master-slave elastic systems, was introduced by Rajan et al. [14]. Applications created using Work Queue support the addition of slave replicas in-process. The slaves are implemented as executable files that the user may instantly start up on many devices as needed. When a job is carried out, the slaves interact with the master, who organizes the work's completion and the exchange of data.

3. PROPOSED SCHEME:

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

To create scalable and elastic total order in content-based pub/sub systems, we first present a distributed two-layer pub/sub framework built on the cloud computing environment. The framework's two primary levels are the matching layer and the delivery layer.

Events must be sorted according to the total order semantics and distributed to interested subscribers via the delivery layer. Matching events with subscriptions and sending events with matched subscribers to the delivery layer are the responsibilities of the matching layer. The main factors restricting the scalability of the matching service are the number of subscribers, the dispersion of subscribers and events, the complexity of the data, and the pace at which events arrive. The two main factors limiting the scalability of the full ordering service are the pace at which events arrive and the possibility of ordering conflicts. So, if both services are detachable, we may dynamically raise the capacity of either the event matching service or the ordering service as a whole depending on the peculiarities of the distinct workloads.

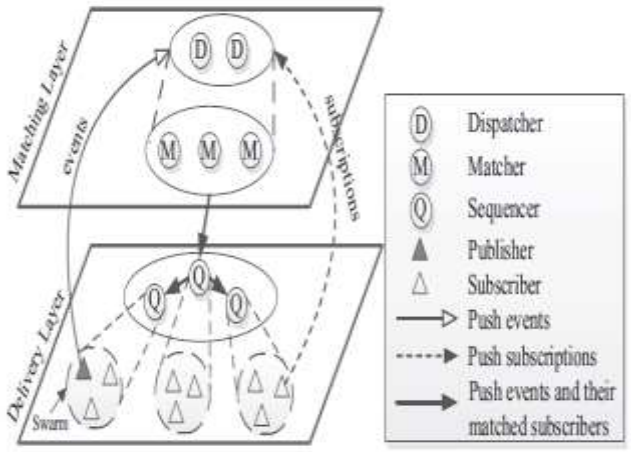

Figure 1.ESCC Architecture

3.2 Matching Layer

Once the matching layer has matched events and subscriptions, the delivery layer gets events together with their matched subscribers. In order to offer high matching throughput at this layer, we use the SREM technique, as shown in Figure 1. In SREM, a hierarchical multi-attribute space partitioning technique called HPartition is used to divide the content space into several hypercubes that are each under the management of a single server. Subscriptions and events from the same hypercube are paired throughout the matching process, greatly lowering the matching latency. Additionally, ESCC suggests using PDetection, a performance-aware detection approach, to adaptively alter server size in response to workload churn.

3.2 Delivery Layer

All ordering events must be made available to subscribers who are interested via the delivery layer. A performance-aware provisioning approach and a preexisting graph creation tool (PGBuilder) dubbed PProvision make up this layer's main innovation. The first makes an effort to scaleably reduce order latency as a whole. In PGBuilder, subscribers are divided up into several groups, each of which is managed by a distinct server. In other words, simultaneous detection of entire order conflicts by all PGBuilder servers reduces delivery latency dramatically. Each PGBuilder server builds prior graphs among arrival events in order to quickly detect non-conflicting events and disseminate them in parallel. To ensure reliable delivery, PGBuilder also provides a variety of dynamical maintenance tools.

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, a global queue (GQ) controls all of these clusters by supervising how each sequencer distributes arrival events into separate, unique clusters. Then, e is handed to the tail cluster. Conflicts between clusters are naturally created by the GQ, and each cluster may be seen of as a sliding window. Each sequencer only processes the head cluster as a result. Once all connected subscribers have received the events from the head cluster, it is removed from the global queue and the sequencer moves on to processing the next head cluster. As a consequence, instead of having to check for conflicts with all GQ events, each new arrival event simply has to check the tail cluster. By organizing events into many clusters, it considerably reduces the conflict detection latency regardless of the event arrival rate.

3.3 Delivery Strategy

In the ESCC approach, subscribers and sequencers have the two main roles. The joining or leaving of either job may greatly worsen total ordering performance, thus we will investigate ways to maintain continuous and efficient total ordering under dynamic networks.

i. Subscription: Keep in mind that each subscriber in our system, shown in Fig. 1, communicates their subscriptions to one of the dispatchers. The sequencer whose hash value is closest to the new subscriber's value is delivered when they join the system, in accordance with the consistent hashing procedure. To ensure trustworthy delivery, the sequencer sends the upcoming event only after receiving all acknowledgements from subscribers of the prior event. Thus, sequencers must get their local subscribers' most current viewpoint. Otherwise, waiting for acknowledgements from unsubscribed subscribers can cause an unreasonable delivery delay.

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:

When many new sequencers are added to the system, the root sequencer selects each new sequencer individually to be a child of the current sequencers. As a result, each retiring sequencer should be able to identify whether every local subscriber has to be transferred to one of the new sequencers based on the consistent hashing technique.

4. Performance Analysis

In this section, the design and implementation of the ESCC prototype as well as a performance analysis of the proposed framework are described. We created the prototype using modular, portable object-oriented middleware so that users may focus on the application logic rather than interfacing with low-level network programming interfaces. Every 200 ms, each matcher evaluates its waiting time in order to determine its scalability and adaptability. For adding and removing matchers, respectively, we set the timeout intervals Tout and T′ to 10 seconds. In other words, up until the continuous arrival or failure interval of two matchers exceeds 10 s, ESCC accumulates all fresh matchers and failed matchers. A matcher won't be successful, according to Ds's hypothesis, if the continuous arrival interval between two heartbeat signals is more than 10 s. The ESCC must alter the number of matchers before it can adjust to both linear and instantly growing event arrival rates, as seen in Figure 2. The approach next shows how the ESCC can adapt to decreasing event arrival rates that are both linear and instantaneous.

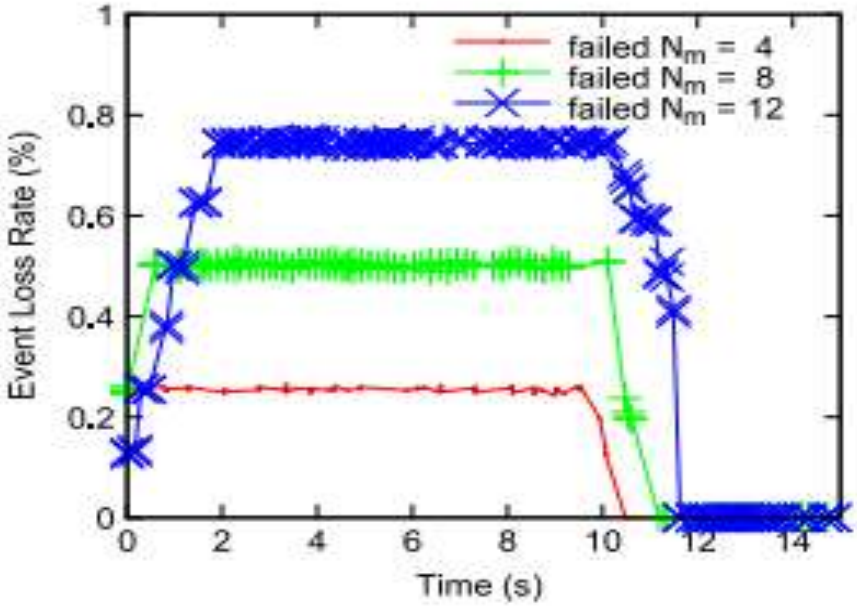

Figure 2. The changing of event loss rate with a number of matcher's failure.

5. Conclusion

This work introduces ESCC, a fresh, scalable, and elastic event matching approach for attribute-based pub/sub systems. In the context of cloud computing, ESCC employs a one-hop lookup overlay to reduce clustering latency. The ESCC uses a hierarchical multi-attribute space partitioning strategy to enable scalable clustering of subscribers and matches each event on a cluster. The performance-aware detection technique allows the system to adjust the size of matchers to changing workloads. Comparing ESCC to the present cloud-based pub/sub systems, our analytical and experimental results show that ESCC has a much higher matching rate and better load balancing with diverse workload characteristics. Furthermore, with minimum latency and traffic overhead, ESCC reacts to unforeseen workload fluctuations and server failures.

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