HSIENA: a hybrid publish/subscribe system*

Fabio Petroni[1] and Leonardo Querzoni[1]

Department of Computer, Control, and Management Engineering Antonio Ruberti
Sapienza University of Rome – Rome, Italy
fabio.petroni.1986@gmail.com, querzoni@dis.uniroma1.it

Abstract. The SIENA publish/subscribe system represents a prototypical design for a distributed event notification service implementing the content-based publish/subscribe communication paradigm. A clear shortcoming of SIENA is represented by its static configuration that must be managed and updated by human administrators every time one of its internal processes (brokers) needs to be added or repaired (e.g. due to a crash failure). This problem limits the applicability of SIENA in large complex critical infrastructures where self-adaptation and -configuration are crucial requirements. In this paper we propose HSIENA, a hybrid architecture that complements SIENA by adding the ability to self-reconfigure after broker additions and removals. The architecture has a novel design that mixes the classic SIENA's distributed architecture with a highly available cloud-based storage service.

1 Introduction

The widespread adoption of the clients/server interaction paradigm has led in the past to the development of distributed applications with a rigid structure, constrained by the lack of flexibility of point-to-point and synchronous interactions. The evolution of the Internet, pushed in the last years by the huge growth of large-scale systems in the form of peer-to-peer and social applications, is clearly marking the limits of this approach to communication, and raising the demand for more flexible interactions schemes. The publish/subscribe interaction paradigm [3] has been introduced in the past as an alternative to the clients/server sibling, with the aim of providing a form of communication where interacting parties are decoupled. This paradigm is today witnessing a wide adoption thanks to its ability to support large scale applications [8, 2, 5, 13] and represents a preferred choice for communication in large complex critical infrastructures (LCCI).

In a publish/subscribe interaction participants to the communication can act both as producers (*publishers*) or consumers (*subscribers*) of information that takes the form of *events*. Subscribers can express which events they want to receive issuing *subscriptions* that express conditions on the content of events (*content-based subscription model*) or just on a category they belong to (*topic-based subscription model*). The paradigm states that once an event is published,

* This work has been partially funded by the DOTS-LCCI Italian project.

for each subscription whose conditions are satisfied by the event (i.e., the event matches the subscription), the corresponding subscriber must be notified. The basic building block of systems implementing the publish/subscribe paradigm is an *event notification service* (ENS) whose goal is to diffuse any published event from the publisher to the set of matched subscribers. The complete decoupling offered by this form of interaction makes it appealing for modern distributed applications characterized by very large scale and variable loads.

The SIENA publish/subscribe system [6, 7] is widely recognized as a representative example of a distributed ENS adopting the content-based subscription model. However, its adoption in LCCI scenarios has been hindered by the lack of adequate support to system reconfiguration. In particular the addition of a new broker to the ENS infrastructure, due for example to a load surge that cannot be adequately managed by the existing brokers, requires a manual intervention of the administrators that would bring the system to a full halt, with a strong impact on service availability and continuity. Similarly, node failures are not tolerated by the current SIENA design, and their correct management requires again manual intervention by system administrators. Both these characteristics are fundamental in LCCIs where unexpected load surges or failures must be quickly tackled with, thus raising the demand for communication infrastructures able to self-adapt and configure.

In this paper we propose HSIENA, a hybrid architecture that complements SIENA by adding the ability to self-reconfigure after broker additions and removals. The architecture has a novel design that mixes the classic SIENA's distributed architecture based on managed brokers with a highly available cloud-based storage service (hence the *hybrid* adjective). Brokers use this storage service as a shared memory space they can rely-on to adapt at runtime the ENS application-level network without service disruption. This shared memory space will contain all the information needed to build the internal state of each broker (with the exclusion of content-based addresses) and will be updated every time a new broker joins the system or as soon as a broker failure is detected. Due to costs associated with accesses (read and write) to the cloud-based storage service (both in terms of access latency and economical costs), HSIENA is designed to limit its usage only to broker additions and removals, that should be considered as "rare" events.

The rest of this paper is organized as follows: Section 2 provides fundamental background knowledge on the publish/subscribe paradigm, on the SIENA system and on the current state of the art related to this work; Section 3 introduces the HSIENA system describing its architecture, the data structures it uses, the fundamental functionalities it provides and how it manages concurrency and event diffusion. Finally, Section 4 concludes the paper.

2 Background

The publish/subscribe interaction paradigm provides users of a system with an alternative communication model with respect to the classical client/server model. In a publish/subscribe system users interact playing one of two roles: publishers that produce information and inject (publish) it into the system in the form of events, and subscribers that consume events received from the system. A third component, the event notification service, has the role of receiving the events injected by the publishers and notify all the subscribers that can be interested in those events. The ENS plays in the system a role of a mediator between publishers and subscribers, decoupling the interactions among them in time, space and flow.

Subscribers can define the set of events they are interested in by issuing subscriptions. Each subscription works as a filter on the set of events injected in the system: the subscriber will be notified only about events that satisfy (match) the conditions expressed by its subscriptions. Subscriptions can be expressed in various ways depending on the subscription model adopted by the system. Currently, two models have been widely accepted by the community working on publish/subscribe: the topic-based and the content-based models.

The content-based model provides users with an expressive subscription syntax. This model requires the definition of a global event space represented by a collection of attributes each characterized by a name and a type (integer, floating-point, strings, etc.). Given a specific event space, an event is a collection of values, one for each attribute. The greater flexibility of this model comes from the possibility of defining each subscription as a complex expression constituted by constraints expressed on the attributes defined by the event space. A subscriber will by notified by the event notification service about an event only if it satisfies the expression contained in one of the node's subscriptions. Examples of systems that adopt the content-based subscription model are SIENA [6, 7], JEDI [9] and Gryphon [14].

2.1 SIENA pub/sub system

The SIENA publish/subscribe system [7] is based on a distributed architecture where the ENS is made up of several machines, named *event brokers*. Brokers act as access points for clients (publishers and subscribers) that want to access the system and, furthermore, they are in charge of routing events among them in order to correctly notify published events to the intended recipients. SIENA's ENS is designed as a layered architecture:

- at the bottom layer stands an application-level *overlay network* represented by a generic graph interconnecting all the brokers. SIENA does not define how this overlay networks should be built and maintained, but only assumes that it is always connected;
- just above it a *broadcast layer* defines multiple spanning-trees (one for each broker as a possible source of events) that are used to diffuse published events toward all brokers where target subscribers are attached; spanning trees are defined with broadcast function $B : N \times I \rightarrow I^*$ that is supposed to be statically configured by a system administrator;

– at the top a *content based layer* is used to prune branches of spanning-trees that do not lead to potential target subscribers in order to reduce the amount of network traffic generated during event diffusion. Pruning is done on a per-event basis using information collected from previously issued subscriptions. Such information is maintained by each broker in a *content-based forwarding table* whose content is updated on the basis of two complementary protocols called *receiver advertisements (RA)* and *sender request/update replies (SR/UR)*.

Note that the two lowermost layers defined in SIENA are *static*, i.e. the authors assume that they are managed by system administrators and do not provide any algorithms for their maintenance. However, the correctness of event routing mechanism working on top of the layered architecture is based on the assumption that the spanning-trees defined in the broadcast function B are always connected.

2.2 Related Work

A few works in the literature has tackled the problem of making the SIENA pub/sub system more autonomous and adaptable by allowing automatic reconfigurations and fault tolerance at the overlay and broadcast layers. Among these we can cite XSIENA [10] that adopts a strategy based on soft-state and lease timeouts to maintain up-to-date routing information within the system and thus tolerate possible reconfigurations. The complexities of readapting the application-level network topology after failures have been explored in [12] where the authors also proposed an enhancement to the "strawman approach". Finally, self-organization and reconfiguration of the application-network has also been exploited in [4] with a different goal, i.e. performance optimization.

The idea of using a cloud-based services for building more reliable publish/subscribe system has only been recently applied Hedwig [2] that leverages the Zookeeper service for configuration storage (e.g. locations of ledgers, subscriber information), topic leader election and system membership. However, it should be noted that Hedwig has been designed as a system completely integrated within the cloud provider architecture, while our solution uses a cloud based storage service as a completely external service and can thus be deployed *outside* of the cloud provider architectural boundaries.

From this point of view our approach closely follows what has been proposed in [11]. Also in that paper a hybrid architecture is proposed, but, differently from HSIENA, a cloud-based storage service is only used to maintain simple loosely structured information.

3 HSIENA architecture and algorithms

The following section introduces HSIENA a hybrid architecture that complements SIENA by adding the ability to self-reconfigure to manage broker additions and removals. We will first give an overview of HSIENA architecture together with the principles we adopted in its design; then we will detail the data structures it uses and the algorithms that support broker additions and removals.

3.1 Overview and design principles

The main issues that must be faced to add self-reconfiguration capabilities are: (i) reconfigure the overlay network, (ii) recalculate the broadcast function B, (iii) diffuse information about the new B in the ENS and (iv) update the content based forwarding tables.

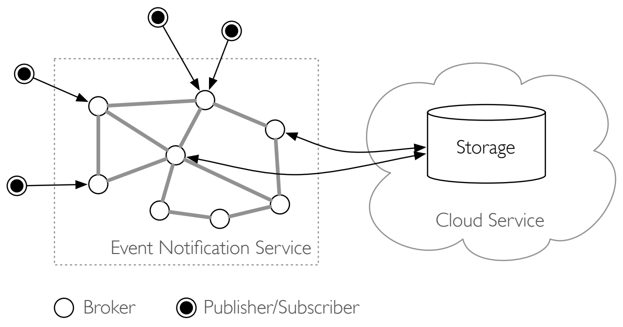

Fig. 1. HSIENA hybrid architecture

These operations must be conducted by managing issues related to concurrency among brokers and trying to reduce the possible loss of events published in the system while reconfigurations are ongoing. To fulfill its goals, the HSIENA system employs an hybrid architecture where the classical SIENA distributed ENS is paired with a cloud-based storage service (see Figure 1). The storage service is used by brokers as a *shared memory* where some of the global system state can be maintained, to be retrieved and updated every time a new broker needs to join the system or an existing one leaves or fails. The usage of a cloud-based storage service provides guarantees on availability and reliability of stored data, does not impose a limit on the system scalability and provides basic concurrency control features. However, accessing such service has a non negligible impact on performance (due to latencies induced by remote read/write operations) and overall cost (the cloud service provider applies specific billing policies for each operation).

Brokers entering the system first creates links toward other brokers. We here assume that these links are initialized as the result of a call to an external bootstrap service whose details are out of the scope of this paper. Then, the broker reads from the storage service the current state of the system and uses a variation of the classic Floyd-Warshall algorithm to build its broadcast spanning-tree and update the global B function. When the system state is updated, the broker starts a procedure to inform other brokers in the system that the global state has been updated and that they should access the storage service to update their local data structure. During this last phase content-based forwarding tables are updated in accordance with the new B function. Broker removals follow a similar approach.

Transitions from a system state to the next one, due to a broker addition or removal, are marked by an *epoch number*. States relative to old epochs are maintained in the storage service until all events published during that epoch has been correctly notified to the intended recipients.

3.2 Data structures

The cloud-based storage service is used to maintain three main data structures. The first (and most important) is a list of couples of matrices. Each couple is associated to a specific *epoch* i and is constituted by D^i and $Pred^i$ matrices. The size of both matrices is $|N^i| \times |N^i|$ integer values, where N^i is the set of brokers in the system for epoch i and their content is calculated using the Floyd-Warshall algorithm (further details are given in Section 3.3). The D^i matrix contains in position $d[x, y]$ the length of the path connecting broker b_x to b_y in the spanning tree routed on b_x (the matrix is symmetrical as the B^i function must respect the *All-pairs path symmetry* condition as is defined in [7]). The $Pred^i$ matrix contains in position $p[x, y]$ the id of the broker that precedes b_y in the path that connects b_x to b_y. The $Pred^i$ matrix describes the broadcast functions of every node $x \in N^i$. In fact, let V_x^i be the set of neighbors of x (its interfaces set) at epoch i:

$$v \in B_x^i(y, \cdot) \leftrightarrow p[y, v] = x \ \ \forall v \in V_x^i \ \forall x, y \in N^i \tag{1}$$

The second data structure is a single integer variable whose value represents the current epoch.

Finally a list of ongoing operations *Ops* is used to manage concurrent access from various brokers to the previous data structures. This list is managed as a queue and brokers willing to execute some operations on the global state (i.e. write updated D_i and $Pred_i$ tables and/or update the current epoch) must first enqueue here their operations and act only after previous modification have been concluded and the corresponding entries removed from the list (see Section 3.5 for further details).

Such data structures can be easily stored in any cloud-based storage service. In the following, we will consider a concrete example where Amazon SimpleDB [1] is used for this purpose. More specifically the HSIENA system access its global state stored on SimpleDB using the *CreateDomain*, *DeleteDomain*, *BatchPutAttributes*, *ConditionalPut* and *Select* commands offered by the SimpleDB API.

Brokers locally maintain a copy of the B function and the content based forwarding table for several epochs i. A *garbage collection* mechanism can be employed to remove old data structures both from brokers and from the storage service. Data structures relative to epoch i can be removed as soon as all the brokers made their transition to some epoch $j > i$.

3.3 Broadcast layer reconfiguration protocol

When a broker k must be added or removed from the system the first step to be performed is a reconfiguration of the broadcast layer whose goal is to update the B function on all brokers. This reconfiguration is performed by HSIENA using a protocol whose internal functioning is tailored to the specific operation that is going to be performed: insertion or removal. While node insertion are always voluntary, a broker can leave the system either voluntarily (i.e. a controlled shut-down or reboot) or due to a crash failure. In the latter case we assume that the reconfiguration protocol is executed on behalf of the crashed broker by one of its former neighbors. Note that, for the sake of clarity, here we will omit details related to concurrency management. Such details will be added in section 3.5.

Broker k executes the following protocol:

1. reads the current epoch number i from the storage service;
2. reads D^i and $Pred^i$;
3. executes the specific *insertion* or *removal algorithm*;
4. stores D^{i+1} and $Pred^{i+1}$ on the storage service and updates the current epoch to $i + 1$;
5. floods the system with an *Epoch Update* (EU) message containing (i) its id, (ii) a bit representing the fact that the update has been caused by the insertion or the removal of a broker and (iii) the current epoch number $i+1$. The message flooding is realized directly on the overlay network connecting all brokers and we assume that k is the first broker to receive the message.

When a broker y receives an EU message related to the insertion or removal of broker k that moved the system to epoch $i + 1$ it follows these steps:

1. discards it if the same message was previously received and halt the protocol;
2. forwards the message on the overlay network to continue the flooding operation;
3. for each $s \in N^{i+1}, B(s, \cdot) = \emptyset$;
4. performs a *select* operation on the storage over D^{i+1} in order to know which rows have the element at column y equal to 1. This set represents the new set of neighbors V_y^{i+1}. Interfaces are added or removed accordingly;
5. for each $x \in V_y^{i+1}$ performs a *select* operation on the storage over $Pred^{i+1}$ in order to know which rows have the element at column x equal to y. Let R be the set of ids (rows) in which this occur. It sets $\forall s \in R, B^{i+1}(s, \cdot) = B^{i+1}(s, \cdot) \cup \{I_x\}$

If the operation is an insertion, in order to avoid event loss during reconfiguration, the node reads the row y from $Pred^{i+1}$. If in this line the id k is present, then it changes the predicate associated with interface I_{out} to ALL (i.e. the predicate is matched by any content based address), where I_{out} is the interface through which y reaches k in the spanning tree routed at y. Note that executing the same operation in case of a broker removal cannot prevent event loss, and can thus be skipped.

The insertion and removal algorithms (step 3 of the *reconfiguration protocol*) both work locally on the basis of D^i and $Pred^i$ matrices in order to produce D^{i+1} and $Pred^{i+1}$.

Broker insertion algorithm

1. Broker k builds its set of neighbors V_k^{i+1} resorting to an external bootstrap service[1];
2. adds a new line and a new column, both labelled with its id k, to D^i and $Pred^i$ obtaining locally the matrices D^{i+1} and $Pred^{i+1}$;
3. fills line k of matrix D^{i+1} using the following rule:

   $$\forall y \in N^i, y \neq k, d[k,y] = min\{d[x,y] : x \in V_k^{i+1}\} + 1 \qquad (2)$$

   The addition of a single unit is due to the fact that we consider the weights of links connecting brokers all equal to 1. Column k is filled with the same values as D is symmetric;
4. every time an entry $d[k,y]$ is updated in D^{i+1} it checks which is the neighbor x (i.e. $x \in V_k^{i+1}$) that minimizes $d[x,y]$ and then updates $Pred^{i+1}$ using the following rule:

   $$\forall y \in N, p[k,y] = p[x,y]$$
   $$p[y,k] = x$$

5. loops on every $x, y \in N^{i+1}$ with $x, y \neq k$: if $d[x,y] > d[x,k] + d[k,y]$ then it sets $d[x,y] = d[x,k] + d[k,y]$ and $p[x,y] = p[k,y]$.

Broker removal algorithm

The removal algorithm is constituted by a recursive routine that takes as input a set X of row ids, a set Y of column ids and a set V of visited broker ids. The first call is executed using $X = \{1, \cdots, k-1, k+1, \cdots, n\}$, $Y = V_k^i$ and $V = \{k\}$.

1. for each $y \in Y$
   (a) defines $X' = \{x \in X : p[x,y] \in V\}$;
   (b) defines $Y' = V_y^i / V$;
   (c) if $Y' = \emptyset \wedge X' \neq \emptyset$ the overlay network is disconnected and the procedure halts as the manual intervention of a system administrator is needed[2];

[1] This bootstrap service, although not precisely defined in this paper, could be easily designed using the cloud based storage service where a list of ids and IP addresses of brokers currently in the system can be maintained.

[2] Note that spanning trees defining B are built by the insertion procedures using minimum paths among any pair of brokers in the system. Therefore, if we consider the union of all the spanning trees, we obtain the graph representing the overlay network connecting all brokers. As a consequence, the crash of a broker disconnects the system if and only if this broker did not appear in any spanning tree as a leaf. This property gives us a good methodology to understand if a disconnection has occurred just knowing the id of the failed broker and the *Pred* matrix.

(d) if $X' \neq \emptyset$ executes a recursive call using X', Y' and $V \cup \{y\}$ as input parameters;
(e) for each $x \in X'$:
  – sets $d[x, y] = min(d[x, z], z \in Y') + 1$ in D^{i+1};
  – every time an entry $d[x, y]$ is updated it checks which is the neighbor z (i.e. $z \in Y'$) that minimizes $d[x, z]$ and then updates $Pred^{i+1}$ with $p[x, y] = z$;

2. returns from the recursive call.

3.4 Content based layer reconfiguration protocol

The insertion or removal of a broker can easily disrupts the correctness of information stored on several content based forwarding tables. Therefore, after the conclusion of the broadcast layer reconfiguration protocol, broker k must take care of starting a new phase where content based forwarding tables, whose information could have become stale, are updated accordingly with the new system configuration.

If the phase is started after an insertion procedure, k executes the following protocol:

1. for each interface I_a it builds an *ad-Hoc Sender Request* (HSR) message that, beside all the information included in a standard *Sender Request* message, includes a set of brokers S_a such that $\forall s \in S_a, B_k^{i+1}(s, \cdot) \supseteq I_a$, and forwards it through I_a;
2. waits for a corresponding *ad-Hoc Update Reply* (HUR) message from the same interface I_a and uses its content to update the predicate associated to I_a in the content based forwarding table.

HSR messages are treated by a brokers y receiving it almost like a normal *Sender Request* messages. The only exception being that for any id $s \in S_a$ included in the HSR y checks if $B_y^{i+1}(s, \cdot) = \emptyset$. If the condition is satisfied it removes s from S_a and locally stores a triple $< HSRnumber, s, P_y >$ (where P_y is y's predicate).

Also HUR messages are treated similarly to standard *update replies*. When a broker y receives an HUR message for each triple $< HSRnumber, s, P_y >$ previously stored it adds to the body an *Extended Update Reply* EUR message containing a tuple $< s, P_y, \emptyset >$. For each EUR message $< s, P_x, \emptyset >$ already contained in the HUR it adds to the predicate its own predicate P_y (i.e. the EUR message become $< s, P_x \cup P_y, \emptyset >$). Finally, multiple EUR messages targeted to the same broker s are collapsed in a single EUR message containing the union of the respective predicates.

When k receives the expected HUR messages, it knows each of them will contain as many EUR messages as the number of broker ids included within S was. Also in this case multiple EUR messages targeted to the same broker s are collapsed in a single EUR message containing the union of the respective predicates. For each EUR message $< s, P, \emptyset >$, k calculates the route the message will follow on the overlay network in order to reach s. This route is obtained by looking at paths defined within $Pred^{i+1}$ and is then included in the EUR message as a list of brokers, i.e. $< s, P, \{k, \cdots, s\} >$. The message is then forwarded by brokers in the ENS simply following the path included in it. When a broker y receives an EUR message targeted to it, it updates the predicate associated in its content based forwarding table to the interface the message was received from by performing an union of the existing predicate with the one contained in the message.

In the case of a broker removal procedure HSIENA simply requires brokers receiving an EU message to reissue their predicate through a standard *receiver advertisement* procedure.

Note that after both these procedures brokers in the overlay have content based forwarding tables containing predicates that are correct (i.e. events are notified to all the intended recipients) but not exact (i.e. some events could be routed on some edges of the overlay without reaching any matched subscription). This behavior is consistent with the *inflation* phenomenon that already affects SIENA and can thus be mitigated using the standard SIENA mechanisms.

3.5 Concurrency management

Broker insertions and removals can happen concurrently in HSIENA as their actions are not coordinated by a central administration. Consequently, multiple brokers can concurrently execute the previous procedures, while the coherence of information stored both on the brokers and in the storage service must be preserved. To this aim, a concurrency management strategy is needed such that concurrent operations will always leave at least the information stored on the storage service in a coherent state.

To reach this goal we assume that the storage service provides a *test-and-set* primitive that can be used to write data without ignoring previous updates. In SimpleDB this primitive is called *ConditionalPut*.

The list of ongoing operations Ops is the core of our concurrency management solution. Each element in the list is a tuple $< i, [INS/REM], y, V >$ indicating the broker y is performing a $[INS/REM]$ operation that will bring the system to epoch i. V is a set of broker ids used only for INS operations. The list Ops is managed by brokers as a queue, so insertion operations only happens at the tail. When a broker y starts an insertion (removal respectively) operation:

1. it reads Ops from the storage service;
2. reads the last epoch number i (the epoch number of the last element in the queue or the current epoch number, if the queue is empty) and the current epoch number e;
3. adds an item $< i+1, INS, y, V_y^{i+1} >$ ($< i+1, REM, y, \emptyset >$ resp.) in the queue;
4. tries a *test-and-set* operation to store the new version of the list on the cloud; if the operation fails (i.e. some other broker is concurrently executing the same operation), it re-executes the procedure from point 1;

5. executes the insertion or removal procedures as described in Section 3.3 for every element in *Ops* with the exception of those for which the epoch number is less or equal to e (these operations have been already concluded);
6. reads again the current epoch number e' from the storage service; if $e' > i+1$ it skips the *epoch update* procedure and proceeds to the last step;
7. tries a *test-and-set* operation in order to update current epoch number to $i+1$, if it fails, it re-executes from the previous point;
8. deletes its entry from *Ops* and tries to write it on the storage service, using again the *test-and-set* procedure.

Such procedure guarantees that, in the worst case, the same insertion/removal operations will be executed multiple times by different nodes. However, these operations, thanks to their deterministic evolution, will always bring the system in the same final state.

3.6 Event routing

Event routing in HSIENA is performed using the same mechanism employed in SIENA with the only exception that messages containing events must include the epoch in which each event was published. In principle, an event published in epoch i should be diffused on one of the spanning trees defined in B^i. However, epoch transitions can impose updates on B and this clearly impacts event diffusion. Suppose an event e is published in epoch i while the system is transitioning to epoch $i+1$ (generally an epoch $i' > i$). This case can arise due to the fact the publish operations and epoch changes happen concurrently, and the broker where the event is published possibly did not receive the EU message before the event publication time. To this respect we must distinguish two cases: if the epoch update has been caused by the insertion of a new broker the event diffusion could continue using the old B^i function (with the obvious drawback that the newly inserted broker will not receive the event) as none of the broker present in the spanning tree used for routing it within the ENS disappeared. Contrarily, if the epoch update has been caused by the removal of a broker there is a possibility that the spanning tree has been disconnected by such removal and that the event diffusion will not be completed correctly.

To overcome these issues we propose a straightforward solution that sacrifices some efficiency in favor of better robustness: every time a broker receives an event published in an epoch i older than the epoch i' it knows, it stops the diffusion in i and republish the event in epoch i'. In this way the event will be diffused using updated spanning trees that will take into account possible broker removals. The dual case where a broker receives an event published in an epoch i newer than the epoch i'' it knows, is easily solved by forcing an epoch update on the broker.

The drawback of this approach is that it can cause brokers to receive the same event multiple times (in different epochs); this issue can be mitigated by locally keeping trace of received events in order to avoid local notifications, while waste of network resources cannot be avoided.

4 Conclusions

This paper introduced HSIENA, a hybrid architecture that complements the SIENA publish/subscribe system by adding the ability to self-reconfigure after broker additions and removals. HSIENA has a novel design that mixes the classic SIENA's distributed architecture based on managed brokers with a highly available cloud-based storage service that brokers use as a shared memory space they can rely-on to adapt at runtime the ENS application-level network without service disruption. Currently we are implementing a prototype of HSIENA to test its behaviour under various realistic loads. Our purpose is to asses both its ability to support insertion and deletions while providing service continuity (i.e. continuing to notify events also during reconfiguration phases) and to study the tradeoff existing between the level of service HSIENA can guarantee and the cost incurred for maintaining state information stored on a cloud service.

References

1. Amazon SimpleDB, `http://aws.amazon.com/simpledb/`
2. Hedwig, `https://cwiki.apache.org/confluence/display/BOOKKEEPER/HedWig`
3. Baldoni, R., Querzoni, L., Tarkoma, S., Virgillito, A.: Distributed Event Routing in Publish/Subscribe Communication Systems. Springer (2009)
4. Baldoni, R., Beraldi, R., Querzoni, L., Virgillito, A.: Efficient publish/subscribe through a self-organizing broker overlay and its application to siena. Comput. J. 50(4), 444–459 (2007)
5. Bharambe, A.R., Rao, S., Seshan, S.: Mercury: a scalable publish-subscribe system for internet games. In: Proceedings of the 1st workshop on Network and system support for games. pp. 3–9 (2002)
6. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and evaluation of a wide-area notification service. ACM Transactions on Computer Systems 3(19), 332–383 (August 2001)
7. Carzaniga, A., Rutherford, M.J., Wolf, A.L.: A routing scheme for content-based networking. In: INFOCOM (2004)
8. Cooper, B., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment 1(2), 1277–1288 (2008)
9. Cugola, G., Nitto, E.D., Fuggetta, A.: The jedi event-based infrastructure and its application to the development of the opss wfms. IEEE Transactions on Software Engineering 27(9), 827–850 (September 2001)
10. Jerzak, Z., Fetzer, C.: Soft state in publish/subscribe. In: Gokhale, A.S., Schmidt, D.C. (eds.) DEBS. ACM (2009)
11. Montresor, A., Abeni, L.: Cloudy weather for p2p, with a chance of gossip. In: Asami, T., Higashino, T. (eds.) Peer-to-Peer Computing. pp. 250–259. IEEE (2011)
12. Picco, G.P., Cugola, G., Murphy, A.L.: Efficient content-based event dispatching in the presence of topological reconfiguration. In: ICDCS. pp. 234–243. IEEE Computer Society (2003)
13. Pietzuch, P., Shand, B., Bacon, J.: Composite event detection as a generic middleware extension. Network, IEEE 18(1), 44–55 (2004)
14. The Gryphon Team: Achieving Scalability and Throughput in a Publish/Subscribe System. Tech. rep., IBM Research Report RC23103 (2004)