Dynamic Load Balancing in Distributed Content-based Publish/Subscribe

Alex King Yeung Cheung and Hans-Arno Jacobsen

Middleware Systems Research Group
University of Toronto, Toronto, Ontario, Canada
http://www.msrg.utoronto.ca
{cheung,jacobsen}@eecg.utoronto.ca

Abstract. Distributed content-based publish/subscribe systems to date suffer from performance degradation and poor scalability caused by uneven load distributions typical in real-world applications. The reason for this shortcoming is due to the lack of a load balancing solution, which have rarely been studied in the context of publish/subscribe. This paper proposes a load balancing solution specific to distributed content-based publish/subscribe systems that is distributed, dynamic, adaptive, transparent, and accommodates heterogeneity. The solution consists of three key contributions: a load balancing framework, a novel load estimation algorithm, and three offload strategies. Experimental results show that the proposed load balancing solution is efficient with less than 1.5% overhead, effective with at least 91% load estimation accuracy, and capable of distributing all of the system's load originating from an edge point of the network.

Keywords: Publish/subscribe, load distribution, content-based routing, load balancing, load estimation, subscriber migration, offloading algorithm
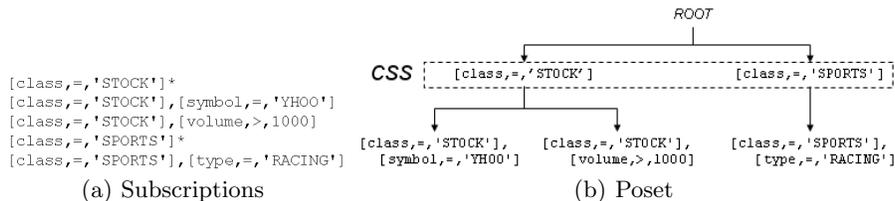
1 Introduction

Brokers in a distributed publish/subscribe system located at different geographical areas may suffer from uneven load distribution due to different population densities, interests, and usage patterns of end-users. A broker in a hotspot area where there is high message traffic resulting from a large number of publishers and subscribers may get overloaded in two ways. First, a broker can be overloaded if the incoming message rate into the broker exceeds the processing/matching rate supported by the matching engine. This effect is exacerbated if the number of subscribers is large because the matching rate is inversely proportional to the number of subscriptions in the matching engine [9]. Second, overload can also occur if the output transmission rate exceeds the total available output bandwidth. In both cases, queues accumulate with increasingly more messages waiting to be processed, resulting in increasingly higher processing and delivery delays. Worse yet, a broker may crash when it runs out of memory from queueing too many messages.

Since the matching rate and both the incoming and outgoing message rates determine the load of a broker, and these factors depend on the number and nature of subscriptions that the broker services, load balancing is possible by offloading *specific* subscribers from higher loaded to lesser loaded brokers. Hence, we develop a load balancing algorithm that distributes load by offloading subscribers from heavily loaded brokers to less loaded brokers. Our contributions to support this idea include (1) a load balancing framework described in Section 3 that isolates subscribers to the edge brokers in the network and organizes load balancing activities into sessions between two brokers at a time; (2) a novel load estimation algorithm presented in Section 4 that profiles subscription load using bit vectors; (3) offload algorithms proposed in Section 5 to load balance on each performance metric of the broker by selecting the appropriate subscribers to offload based on their profiled load characteristics; and (4) experimental results shown in Section 6 that demonstrates the behavior and performance of our load balancing solution.

2   Background and Related Work

Content-based Publish/Subscribe is widely used in large-scale distributed applications because it allows processes to communicate asynchronously in a loosely-coupled manner. Publish/subscribe middleware can be readily found in online games [3], decentralized workflow execution [11], real-time monitoring systems [15], and the Enterprise Service Bus (ESB) of Service Oriented Architecture (SOA) infrastructures. In this communication paradigm, clients that send publication messages into the system are referred to as *publishers*, while those that only receive messages are called *subscribers*. Publishers issue publications in the form of attribute key-value pairs. Subscribers issue subscriptions to their nearest broker to specify the type of publications they want to receive. For the remainder of our discussion, we will assume that a subscription maps to a single subscriber. Subscriptions consist of predicates made up of attribute key-operator-value tuples to specify the filtering conditions on each attribute. A set of *brokers* connected together in an overlay network forms the publish/subscribe routing infrastructure (see Figure 3). In essence, brokers forward publication messages from the publishers to *matching* subscribers based on the routing paths established by subscriptions. Optimizations such as subscription aggregation [5], subscription merging [13], rendezvous nodes [14], and epidemic algorithms [7] may be employed to make the system more scalable or robust. However, hotspots can still arise because there is no load balancing mechanism.

The space of interest defined by a subscription's filtering conditions is called *subscription space*. A broker's *covering subscription set* (CSS) refers to the set of most general subscriptions whose subscription space is not covered by any other subscription. For example, a broker with the set of subscriptions shown in Figure 1a has a CSS identified by the subscriptions marked with an asterisk. For more efficient retrieval of a broker's CSS, the *partially-ordered set* (poset) [4] is used to maintain subscription relationships. The poset is a directed acyclic graph where each unique subscription is represented as a node in the graph as shown in Figure 1b. Nodes can have parent and children nodes where parent

[class,=,'STOCK']*	
[class,=,'STOCK'],[symbol,=,'YHOO']	
[class,=,'STOCK'],[volume,>,1000]	
[class,=,'SPORTS']*	
[class,=,'SPORTS'],[type,=,'RACING']	

(a) Subscriptions   (b) Poset

Fig. 1. Example of the poset data structure.

nodes have a subscription space that is a superset of its children nodes, while subscriptions with intersection or empty relationships will appear as siblings. As shown, the CSS is readily available as the immediate children nodes under the imaginary *ROOT* node.

Load Balancing has been a widely explored research topic for the past two decades since the introduction of parallel and distributed computing. Load balancing solutions can be found in the network layer [8], operating system layer [12], middleware layer [1], and application layer [2]. However, all of the above approaches cannot estimate the load of a subscription nor account for subscription spaces. These limitations prevent them from balancing load effectively in a heterogeneous content-based publish/subscribe system.

Load Balancing in Content-based Publish/Subscribe was never directly addressed in the past although distributed content-based publish/subscribe systems have been widely studied. Hence, the proposed solution in this paper is to the best of our knowledge the first dynamic load balancing algorithm for broker-based publish/subscribe systems to date.

Meghdoot [10] is a peer-to-peer content-based publish/subscribe system based on a distributed hash table that distributes load by replicating or splitting the locally heaviest loaded peer in half to share the responsibility of subscription management or event propagation. In general, their load sharing algorithm is only invoked upon new peers joining the system and peers are assumed to be homogeneous. Chen et al. [6] proposed a dynamic overlay reconstruction algorithm called *Opportunistic Overlay* that reduces end-to-end delivery delay and also performs load distribution on the CPU utilization as a secondary requirement. Load balancing is triggered only when a client finds another broker that is closer than its home broker. It is possible that subscriber migrations may overload a non-overloaded broker if the load requirements of the migrated subscription exceed the load-accepting broker's processing capacity. Our work differs from Meghdoot and Opportunistic Overlay by proposing a dynamic load balancing algorithm for non-DHT-based publish/subscribe systems that accounts for heterogeneous brokers and subscribers, and distributes load evenly onto all resources in the system without requiring new entities to join the federation. We also present a detailed subscriber migration protocol that enforces end-user transparency and best-effort delivery to minimize message loss.
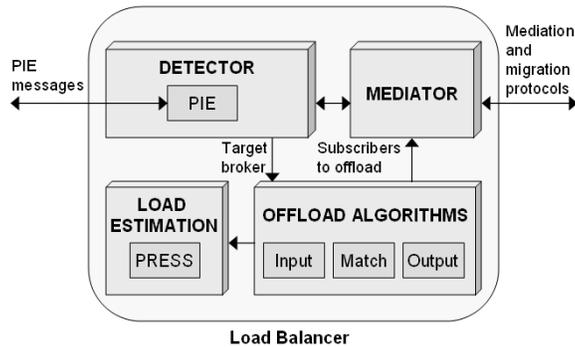
Fig. 2. Components of the load balancer.

3 Load Balancing Framework

The components that make up the load balancing solution are shown in Figure 2. The solution consists of the detector, mediator, load estimation tools, and offload algorithms that determine which subscribers to offload. The detector detects and initiates a trigger when an overload or load imbalance occurs. The trigger from the detector tells the mediator to establish a load balancing session between the two entities, namely *offloading broker* (broker with the higher load doing the offloading) and the *load-accepting broker* (broker accepting load from the offloading broker). Depending on which performance metric is to be balanced, one of the offload algorithms is invoked on the offloading broker to determine the set of subscribers to delegate to the load-accepting broker based on estimating how much load is reduced and increased at each broker using the load estimation algorithms. Finally, the mediator is invoked to coordinate the migration of subscribers and ends the load balancing session between the two brokers. The following sections will describe the load balancing framework and the operations of each component in greater detail.

3.1 Underlying Publish/Subscribe Architecture

The Padres[1] Efficient Event Routing (PEER) architecture organizes brokers into a hierarchical structure as shown in Figure 3. Brokers with more than one neighboring broker are referred to as *cluster-head brokers*, while brokers with only one neighbor are referred to as *edge brokers*. A cluster-head broker with its connected set of edge brokers, if any, forms a *cluster*. Brokers within a cluster are assumed to be closer to each other in network proximity than brokers in other clusters. Publishers are serviced by cluster-head brokers, while subscribers are serviced by edge brokers.

   PEER is designed with five goals in mind. First, PEER allows the load balancing scheme to move subscribers to control load in the edge brokers because they have no broker-to-broker through-traffic to route. Second, higher dissemination efficiency is achieved by having cluster-heads forward publication messages to all

---

[1] Our work is built onto the Padres [11] system
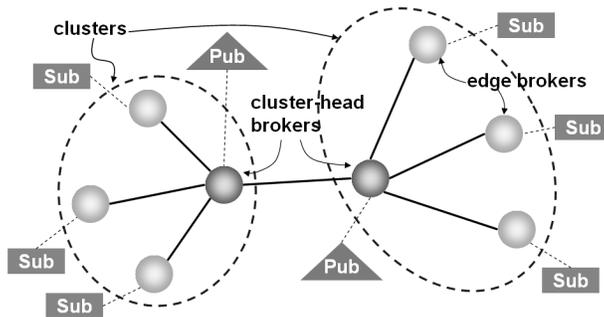
Fig. 3. PEER architecture.

matching clusters almost simultaneously because cluster-heads have negligible processing delays since they do not service any subscribers. Third, cluster-head brokers may be load balanced by moving publishers and inter-broker subscriptions, as will be the focus of future work. Fourth, PEER's organization of brokers into clusters allows for two levels of load balancing: *local-level* (referred to as *local load balancing*) where edge brokers within the same cluster load balance with each other; and *global-level* (referred to as *global load balancing*) where edge brokers from two different clusters load balance with each other. Edge brokers only need to exchange load information with edge brokers in the same cluster, and neighboring clusters can exchange aggregated load information about their own edge brokers. Fifth, local load balancing preserves subscriber locality by keeping subscribers within their original cluster, assuming that subscribers connect to the closest broker in the first place. On the other hand, global load balancing trades off locality for a better balanced system by migrating subscribers between edge brokers in neighboring clusters.

3.2 Load Detection Framework

In order for brokers to know when and which brokers are available for load balancing, they have to exchange load information with each other. With this data, a detection algorithm can then trigger load balancing whenever it detects an overload or a wide load difference with another broker.

Protocol for Exchanging Load Information *Padres Information Exchange* (PIE) is a distributed hierarchical protocol for exchanging load information between brokers using publish/subscribe primitives. Brokers publish PIE messages intermittently to let other brokers in the federation know of their existence and availability for load balancing. PIE, as well as other load balancing control messages described in later sections, has a higher routing priority than normal publish/subscribe traffic so that their delivery is not affected by the brokers' load. A PIE message contains five attributes: (1) the broker's three performance metrics, specifically *input utilization ratio*, *output utilization ratio*, and average *matching delay* per message; (2) load balancing states, which can be one of *OK*, *BUSY*, *N/A*, or *STABILIZING*; (3) the set of edge brokers it

(a) Local load balancing.   (b) Global load balancing.
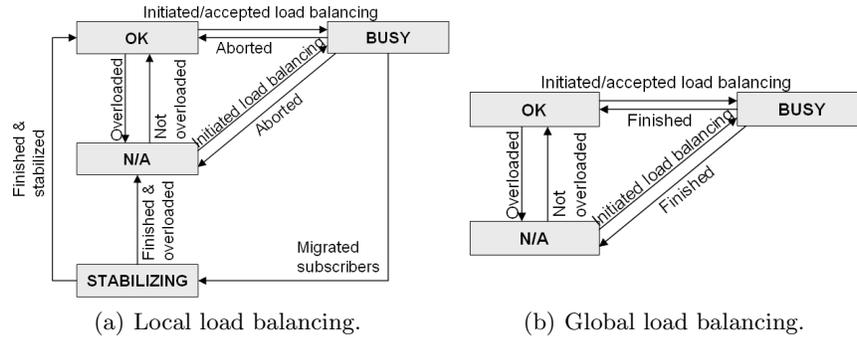
Fig. 4. State transition diagrams for an (a) edge broker and (b) cluster-head.

is currently balanced with (more on this in Section 3.3); (4) the identifier of the cluster to which the broker belongs; and (5) the broker's unique identifier. Input utilization ratio (I_r) captures the broker's processing utilization as defined by the formula: $I_r = RateOfIncomingMessages/MaximumMatchRate$. Maximum match rate is obtained by taking the inverse of the average matching delay per message. Matching delay is defined as the time spent by the matching engine between taking a message as input and producing zero or more messages as output. Output utilization ratio (O_r) captures the broker's output bandwidth utilization as: $O_r = OutputBandwidthUsed/TotalOutputBandwidth$.

PEER's bi-level structuring allows for local and global-level PIE messages. Local PIE messages are published and subscribed by edge brokers within the same cluster to enable local load balancing. Global PIE messages are published and subscribed by cluster-head brokers to enable global load balancing. They only propagate one cluster-hop away and contain averaged load information from their cluster's local PIE messages. Cluster-head brokers without any edge brokers simply forward global PIE messages one extra hop to all of their neighbors.

Detection Algorithm Detection allows a broker/cluster to monitor its current resource usage and also compare it with neighboring brokers/clusters so that a broker/cluster can invoke load balancing if necessary. Detection runs periodically at a broker/cluster only if it has a status of *OK*, *N/A*, and *STABILIZING*. An *OK* status means that the broker is available for load balancing, *N/A* means that it is overloaded, *STABILIZING* means that it is waiting for load to stabilize after load is exchanged, and *BUSY* means that it is currently in a load balancing session with another broker/cluster. A diagram showing the transition conditions between the local and global load balancing states are shown in Figures 4a and 4b, respectively.

The *local detection algorithm* running on an edge broker is composed of two steps. The first step identifies whether the broker itself is overloaded by examining four utilization ratios, namely input, output, CPU, and memory utilization ratio. The parameter *lower overload threshold* is introduced that prevents the broker from accepting further load by updating the broker's status to *N/A* when

one of its utilization ratios exceeds 0.9. If a utilization ratio exceeds the *higher overload threshold* at 0.95, then load balancing is invoked immediately. In between the two thresholds is an inert period where the broker neither accepts nor invokes load balancing. All utilization ratios can be associated with different overload threshold values.

If the first step of the detection algorithm cannot find any overloaded resource, then the second step is to check if any one of the input utilization ratio, output utilization ratio, or matching delay differs from a neighbor by more than a threshold. Recall, load information about neighoring brokers is gathered through PIE. The threshold for utilization ratios is called the *local ratio triggering threshold*, and for matching delay is *local delay trigger threshold*. Both are set to 0.1 by default. The difference for utilization ratio is just the magnitude of the difference, while for matching delay, the following formula is used:

$$d_{\%Diff} = \left| \frac{d_1 - d_2}{N_f} \right| \tag{1}$$

d_1 and d_2 are the two delay values used in the comparison. N_f represents the normalization factor and is set to 0.1 by default so that delay differences much less than 0.1s do not yield high percentage differences and trigger unwanted load balancing. Then, a *broker-action* list of `<load-accepting broker, performance metric/offload action>` is generated that is sorted in descending order of highest performance metric difference. The list is passed to the local mediator to establish a load balancing session with an available load-accepting broker.

After a broker just finishes a load balancing session, its load information may mislead the broker into making an incorrect load balancing decision. For example, brokers accepting load may not experience an increase in utilization immediately. This may cause the broker to accept more load balancing sessions, which may cause its resource consumption to overshoot. To prevent this from occurring, both the offloading and load-accepting broker should inherit a status of *STABILIZING* for *stabilize duration* period (default is 30s) before setting its status back to *OK* (see Figure 4a). After this time, all performance metrics should not fluctuate more than the *stabilize percentage* (default is 0.05) between subsequent detection runs. When a broker has a *STABILIZING* status, it cannot accept load balancing requests nor invoke load balancing unless the broker is overloaded.

Alternatively, in place of their utilization ratio counterparts, it is also possible for the load balancer to use input queuing delay and output queuing delay as performance metrics. However, using queuing delay measurements do not accurately indicate the load of a broker at the instant the metric is measured because it is obtained after the message gets dequeued. Therefore, the measurement is lagging by the delay measured.

In the *global detection algorithm*, a cluster-head uses a subset of the status indicators in local load balancing (see Figure 4b) to indicate its cluster's load balancing status. The only difference here is that a cluster is *N/A* if one or more edge brokers are *N/A*. This allows the overloaded brokers to offload subscribers to brokers within the same cluster first to promote locality. The global detection

algorithm is almost the same as the local detection algorithm, except that the global detector uses different threshold values (namely, *global ratio triggering threshold* and *global delay triggering threshold*, both default to 0.15) and it works with aggregated performance metric values.

3.3 Mediation Protocols

All load balancing activities are coordinated by exchanging messages using the underlying publish/subscribe infrastructure for simplicity and efficiency. Specifically, request-reply and one-way protocols are implemented in publish/subscribe to coordinate broker and subscriber activities.

Mediating Load Balancing Sessions Once the local detection algorithm composes the broker-action list of candidate brokers for load balancing, the *local mediator* sends a load balancing request sequentially to brokers in the sorted list. When a load-accepting broker gets this request, its local mediator replies back with its current status. If the status is *OK*, the request is accepted and both brokers update their status to *BUSY*. In the *OK* case, the load-accepting broker also appends its load information in the reply so that the requesting broker can use this information for its load estimation and offload algorithms to compute which subscribers are suitable for offload. For all other states, the load-accepting broker rejects the load balancing request. In general, two or more load balancing sessions between a pair of brokers can occur concurrently, but a broker cannot participate in multiple load balancing sessions at the same time.

The *global mediator* running at the cluster-head broker uses the same protocol as the local mediator to set up global load balancing. The difference here is that after a successful handshake, both cluster-heads have to tell all edge brokers in their own clusters to subscribe to the other cluster's local PIE messages. This allows edge brokers from one cluster to load balance with edge brokers in the other cluster. Global load balancing ends when all edge brokers are balanced with each other as indicated by the *balanced set* field in local PIE messages. Local PIE subscriptions of the other cluster are undone by unsubscribing when global load balancing ends.

Mediating Subscriber Migration Once the offloading algorithm is done with its computation, it returns back to the mediator a list of subscribers to offload. The mediator has to migrate the indicated subscribers to the new broker in the most efficient and timely manner with minimal delivery loss. First, the mediator sends a control publication message to each subscriber in the offload list telling them to issue their subscription to the new load-accepting broker. Subscribers issue a subscription to the load-accepting broker containing the ID of the load balancing session and the total number of migrating subscribers. These two pieces of information allow the load-accepting broker to know when it has received all migrating subscribers in the current load balancing session. For efficiency and best-effort guarantee of minimal delivery loss, the receiving broker waits for $N \times$ *migration timeout* seconds for all migrating subscribers to connect, where N is the total number of migrating subscribers. Meanwhile, subscribers need to detect and drop duplicate publications (by using a short message history) because

they are subscribing to the same subscription at two brokers simultaneously. When all subscribers have connected to the load-accepting broker, or when the timeout occurs, the receiving broker sends a DONE control publication message back to the offloading broker to terminate the load balancing session. This message ensures that the publication paths for all migrated subscribers have been set up to flow to the load-accepting broker. When the offloading broker receives the DONE message, it tells the migrating subscribers to wait for all the messages currently in its input queue to be matched and delivered from the output queues before sending an unsubscribe message. This waiting period corresponds to the offloading broker's current input queuing delay, plus the matching delay, plus the output queuing delay. Once the migrating subscribers unsubscribe from the offloading broker, the migration process is complete. Note that all control and duplicate messages are handled transparently by a thin software layer on the client side that hides the intricate details of load balancing from the end-user application.

4 Load Estimation Algorithms

Load estimation is used by the offload algorithms to estimate a subscription's load contribution in the form of additional input and output publication rate on the load-accepting broker as well as the load reduced at the offloading broker.

4.1 Estimating Load Requirements of Subscriptions

Padres Real-time Event-to-Subscription Spectrum[2] (PRESS) is a space and time-efficient technique for estimating the bandwidth requirements and common publication set of two or more subscriptions based on current events. It uses bit vectors to record the matching pattern of subscriptions, hence the term *event-to-subscription*. It does not require the publish/subscribe system to use advertisements, nor does it assume that publications are in any sort of distribution. The operation of PRESS is best explained as part of the local load balancing algorithm after the mediation step where two brokers have agreed to load balance with each other.

First, the offloading broker *locally subscribes* to the CSS of the load-accepting broker (as supplied in the *OK* reply message from the replying broker). Locally subscribe means that subscriptions are sent to the matching engine, but never get forwarded to neighboring brokers. This is sufficient because the offloading broker only wants to know which publications it currently sinks are also received by the load-accepting broker. Next, all client subscriptions in the matching engine are allocated a bit vector of length N initialized to 0, where N represents the number of samples. Sampling starts immediately after getting the load-accepting broker's *OK* reply message and ends after N publications have been received or a timeout T is met, whichever comes first. Both N and T are configurable parameters which default to 50 and 30s, respectively. The algorithm starts at the right-most position of the bit vector for all subscriptions. A '1' is set if the subscription matched the incoming publication, '0' otherwise, before moving onto the next

---

[2] *Real-time* refers to sampling using live incoming publications to the broker

10

(a) Candidate subscriptions to offload

Candidate Subscriptions	Bit Vector
[class,=,'STOCK']	110111
...,[volume,>,15]	110100
...,[volume,>,150]	100000
[class,=,'SPORTS']	001000

(b) Load-accepting broker's CSS

Load-Accepting Broker's CSS	Bit Vector
...,[volume,>,50]	110000
...,[volume,<,5]	000001
[class,=,'MOVIES']	000000
CSS bit vector	110001

Table 1. Bit vector example ($\ldots \Rightarrow$ *[class,=,'STOCK']*)

bit on the left. During the sampling period, the total incoming publication rate is measured. Tables 1a and 1b show an example of the bit vectors measured at the offloading broker with $N = 6$ for subscriptions at the offloading broker and load-accepting broker, respectively, given the following publication arrival order:

1. [class,'STOCK'],[volume,0]
2. [class,'STOCK'],[volume,10]
3. [class,'STOCK'],[volume,20]
4. [class,'SPORTS'],[type,'racing']
5. [class,'STOCK'],[volume,100]
6. [class,'STOCK'],[volume,500]

Equation 2 shows the formula to calculate the publication rate matching a particular subscription represented by s_{PR}, where i_r represents the total input publication rate of the offloading broker, n_{BS} represents the number of bits set in the subscription's bit vector, and N represents the number of samples taken in PRESS.

$$s_{PR} = i_r \cdot \frac{n_{BS}}{N} \tag{2}$$

For example, if the total input publication rate i_r at the offloading broker is assumed to be 3msg/s, then for the subscription [class,=,'STOCK'] having 5 out of the 6 bits set, its publication rate comes out to 2.5msg/s. Moreover, the additional incoming publication rate for each subscription can be calculated by using Equation 2 with the bit count obtained from the *ANDNOT* bit operation of the candidate subscription's bit vector with the aggregated load-accepting broker's CSS bit vector. Take the subscription [class,=,'STOCK'] as an example. After the *ANDNOT* bit operation, the bit vector for [class,=,'STOCK'] is:

$$110111 \ ANDNOT \ 110001 = 000110 \tag{3}$$

With a bit count of 2 in 000110, and reusing 3msg/s for i_r, the additional incoming publication rate on the load-accepting broker for this subscription is 1msg/s. In some cases, offloading a subscription may alter the CSS of the load-accepting broker. With PRESS, it is not necessary to resample all subscriptions again because the aggregated CSS bit vector can be updated by merging it with the offloaded subscription's bit vector using the *OR* bit operator. For example, if [class,=,'STOCK'] was chosen for offloading, then the load-accepting broker's CSS is updated to 110111.

Regarding the space and time efficiencies of PRESS: if there are 10,000 subscribers with N set to 100, PRESS only uses 1Mb of memory. Given that the load-accepting broker's CSS is usually small (it is just one in the case of [class,=,*]), an increase in the matching delay is negligible.

Offload Algorithm	Performance Metric Being Balanced	Methodology	Side Effects
Input	Input utilization ratio	Offload subscriptions in the CSS	Output utilization ratio is also decreased at offloading broker and increased at load-accepting broker
Match	Matching delay Overloaded CPU utilization ratio Overloaded memory utilization ratio	Offload subscriptions with least traffic	None
Output	Output utilization ratio	Offload subscriptions with highest traffic in Phase-I	None
		Offload subscriptions with highest traffic and minimal side-effects in Phase-II	Increases input utilization ratio of load-accepting broker

Table 2. Properties of all offload algorithms.

4.2 Estimation of Performance Metrics

Matching delay is estimated by the linear formula [9]:

$$d'_m = \left(\frac{n + \Delta n}{n} \right) \cdot d_m \tag{4}$$

where d'_m is the new matching delay, d_m is the current matching delay, n is the number of subscriptions in the matching engine, and Δn is the change in the number of subscriptions. Estimating the input and output utilization ratios can be done simply by using their original equations but with new estimated values for incoming message rate and output bandwidth consumption using PRESS, respectively.

5 Offload Algorithms

After profiling all subscriptions using PRESS, the offloading broker will use the profiled data along with the load-accepting broker's load information to feed into the offload algorithm to compute the set of subscribers to offload. The offload algorithm to choose depends on what performance metric to balance, which is decided initially by the detector in the broker-action list as mentioned in Section 3.2. Table 2 summarizes the key properties of the three offload algorithms.

5.1 Input Offload Algorithm

This algorithm is invoked by the offloading broker when the input utilization ratio needs load balancing. The aim here is to reduce the offloading broker's input utilization ratio and increase the same metric on the load-accepting broker with minimal effect on the other performance metrics. There are two strategies to reduce the input utilization ratio: increase the rate at which messages are

matched, or reduce the rate of incoming publication messages. Increasing the rate of matching is achieved by reducing the number of subscriptions in the matching engine. However, this action conflicts with the match offload algorithm that is trying to balance the matching delay and therefore is not applied here. Hence, the incoming publication rate can only be reduced by offloading subscriptions in the CSS because their subscription space is a superset of all subscriptions not in the CSS. With the poset [4], CSS lookup will take O(1) time. Once the subscriptions in the CSS are identified, a report card is calculated for each of them. A report card consists of the following fields:

1. **Number of subscribers** of this subscription to offload.
2. Resulting **load percentage difference** between the two brokers by offloading this subscription, where a negative value indicates that the offloading broker will become less loaded than the load-accepting broker. This value is calculated using the estimated input utilization ratios of the two brokers in the input offload algorithm, matching delays in the match offload algorithm, and output utilization ratios in the output offload algorithm.
3. Boolean value indicating if this **subscription is covered** by the load-accepting broker's CSS.
4. **Publication rate reduced** at the offloading broker estimated using PRESS.
5. **Output bandwidth required** per subscriber estimated using PRESS.

The number of subscribers to offload per unique subscription is restrained by two conditions. First, the offload should not overload any of the load-accepting broker's resources. Second, the performance metric of interest of the two brokers should be balanced within the *balanced threshold*, which is 0.005 by default; or bring the offloading broker's metric below the load-accepting broker's. The performance metric of interest for the input, match, and output offload algorithms are the input utilization ratio, matching delay, and output utilization ratio, respectively.

After calculating the report cards to determine the number of subscribers to offload for each subscription in the CSS, the subscription that results in the two brokers' input utilization ratio difference closest to zero is chosen for offloading. This selection scheme ensures that subscriptions with the highest input publication rate are chosen first, which helps to reduce the number of subscriptions offloaded, and in so doing, reduces the impact on the load-accepting broker's matching delay. If all subscriptions will result in a higher load difference than before, then the selection process terminates. This guarantees that all load balancing actions will always converge to a state where the brokers have smaller load differences.

Subscriptions chosen to be offloaded are removed from the poset to prevent future consideration for offloading. Load information about both brokers (the one obtained in the mediation process) is updated with estimated values according to the offloaded subscription's report card. Updated load information about both brokers are used on the next iteration of the subscription selection algorithm. The selection process ends when no more subscriptions are available for offloading, the offloading broker's input utilization ratio is below that of the load-accepting

broker, or the absolute difference between the two brokers' input utilization ratios fall within the *balance threshold.*

5.2 Match Offload Algorithm

Although the input utilization ratio varies directly with the matching delay, balancing the input utilization ratio does not balance the matching delay. The objective of this offload algorithm is to balance the matching delays without affecting the input and output utilization ratios of the two brokers. Intuitively, subscriptions with the lowest publication traffic are most suited to this criterion. Furthermore, subscriptions that introduce less incoming traffic into the load-accepting broker are more favorable. In this algorithm, report cards are computed for all subscriptions in the offloading broker, then they are sorted by ascending output bandwidth. The number of subscribers to offload for each unique subscription is almost identical to the algorithm outlined in the input offload algorithm section. The only difference is that input utilization ratios are replaced by matching delays.

If the match offload algorithm is invoked because the broker is overloaded and wants to reduce its CPU utilization ratio, input utilization ratio, or memory utilization ratio, then subscriptions should continue to be offloaded until the CPU utilization ratio, input utilization ratio, and memory utilization ratio drops below the lower overload threshold. After a subscription is chosen to be offloaded, load information about both brokers are updated. The same criterion used in the input offload algorithm applies here for terminating the match offload process.

5.3 Output Offload Algorithm

This algorithm attempts to balance the output utilization ratios of two brokers by manipulating the amount of output bandwidth used at each broker. Prioritizing subscriptions for the offload process is divided into two phases. In *Phase-I*, subscriptions that are covered by or equal to the load-accepting broker's CSS are considered. These subscriptions are further classified into three types by using the fields computed for every subscription's report card. Offloading *Type-I* subscriptions will reduce the input publication rate of the offloading broker. These should be offloaded first because they reduce the overall input load of the system. *Type-II* subscriptions are similar to Type-I, except that they do not reduce the input publication rate because all subscribers for a subscription cannot be offloaded to produce a more balanced state. *Type-III* subscriptions are considered last in Phase-I because they do not reduce the input publication rate of the offloading broker even if all subscribers for that particular subscription are offloaded. The algorithm for calculating the number of subscribers to offload for each unique subscription is similar to the input offload algorithm shown previously, except that input utilization ratios are now replaced by output utilization ratios.

After a subscription is chosen to be offloaded, load information about both brokers is updated. If both brokers are balanced, then the algorithm stops and forwards the subscriber migration list to the mediator. Otherwise, *Phase-II* is invoked to further balance the output utilization ratio with some side-effects.
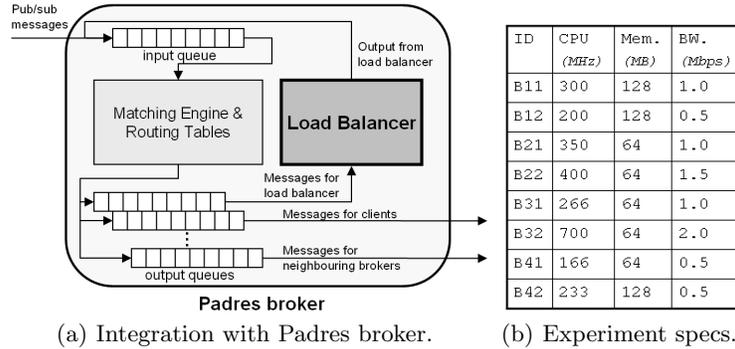
(a) Integration with Padres broker.

(b) Experiment specs.

ID	CPU (MHz)	Mem. (MB)	BW. (Mbps)
B11	300	128	1.0
B12	200	128	0.5
B21	350	64	1.0
B22	400	64	1.5
B31	266	64	1.0
B32	700	64	2.0
B41	166	64	0.5
B42	233	128	0.5

Fig. 5. Evaluation scheme.

All subscriptions considered in Phase-II are not contained in the CSS of the load-accepting broker. Therefore, these subscriptions may have the side-effect of significantly increasing the incoming publication rate of the load-accepting broker. What may happen is that there will be an oscillation between the input offload algorithm trying to balance the input utilization ratio disrupted by Phase-II of the output offload algorithm, and Phase-II of the output offload algorithm trying to balance the output utilization ratio disrupted by the input offload algorithm. To prevent this unstable situation from happening, Phase-II terminates when the input utilization ratios of both brokers are balanced, even if the output utilization ratios are not. An exception applies if the offloading broker is output overloaded, in which case the offloading broker will stop offloading once its output utilization ratio is below the *lower overload threshold*. With this exception, no oscillation occurs because the offloading broker cannot take back any subscriptions since it has a status of *N/A* at the *lower overload threshold*.

The sorting and selection scheme in Phase-II is exactly the same as in the input offload algorithm with the use of load differences. If the subscription offloaded in Phase-II covers other local subscriptions, then Phase-I is invoked to offload those covered subscriptions because they are now covered by the load-accepting broker's CSS. Otherwise, if the subscription offloaded in Phase-II does not cover any other subscriptions, then Phase-II continues to run.

6 Experiments

6.1 Experimental Setup

The proposed load balancing solution is implemented with 20,000 lines of Java code in Padres [11], a distributed content-based publish/subscribe system developed by the Middleware Systems Research Group (MSRG) from the University of Toronto. The load balancer diagram previously shown in Figure 2 is integrated into the Padres broker as the *Load Balancer* component illustrated in Figure 5a. Our experiments are run with Padres brokers under a simulated environment that accounts for all processing delays such as matching, queuing, and bandwidth delays. Default values for the load balancing parameters are used unless otherwise specified. Publishers on creation are assigned to publish stock

quote publications of a particular company at a defined rate. Publishers can be configured to change publication rates at any point in time in the experiment. Stock quote publications are real world values obtained from Yahoo! Finance containing a stock's daily closing prices. A typical publication looks like this:

```
[class,'STOCK'],[symbol,'YHOO'],[open,25.25],[high,43.00],[low,24.50],
    [close,33.00],[volume,17030000],[date,'12-Apr-96']
```

Subscribers are assigned to a fixed subscription based on one of the templates with the probabilities shown below. SUB_SYMBOL is randomly chosen out of the known stock symbols, with SUB_HIGH, SUB_LOW, and SUB_VOLUME replaced by a randomly chosen value of the same attribute from the stock's publication set.

```
20%   [class,=,'STOCK'],[symbol,=,'SUB_SYMBOL'],[high,>,SUB_HIGH]
20%   [class,=,'STOCK'],[symbol,=,'SUB_SYMBOL'],[low,<,SUB_LOW]
20%   [class,=,'STOCK'],[symbol,=,'SUB_SYMBOL'],[volume,>,SUB_VOLUME]
34%   [class,=,'STOCK'],[symbol,=,'SUB_SYMBOL']
5%    [class,=,'STOCK'],[volume,>,SUB_VOLUME]
1%    [class,=,'STOCK']
```

6.2 Local Load Balancing Experiments

The setup used for the local load balancing experiment involves four heterogeneous edge brokers connected to one cluster-head, labeled as *B0*, to form a star topology. *B0* has CPU speed of 2.0GHz with 256MB memory and 10Mbps bandwidth. *B1* has CPU speed of 100MHz with 64MB memory and 0.6Mbps. *B2* has twice the respective performance of *B1*, and *B3* has twice that of *B2*. *B4* has 10 times the respective performance of *B1*. On experiment startup, brokers *B0*, *B1*, *B2*, *B3*, and *B4* are instantiated in order within the first 0.5s. After 5s into the experiment, 40 unique publishers with a randomly chosen publication rate between 0 and 60msg/min start publishing to broker *B0*. 2000 subscribers join broker *B1* at a time chosen randomly between 10s and 1010s using a uniform random distribution. Of all subscribers, 25% have zero traffic, which means their subscriptions do not match any publications in the system. After 3000s, 50% of the publishers are randomly chosen to have their publication rates increased by 100%. This shows the dynamic behavior of the load balancer under changing load conditions. For the experiment on edge broker scalability shown in Figure 6g, all edge brokers have 500MHz CPU, 128MB memory, and 3Mbps bandwidth. All edge brokers are added to the same cluster.

Broker Load Distribution Referring to Figures 6a, 6b, and 6c, broker *B1* becomes overloaded as all the subscribers attempt to connect to it as their first broker while *B1* attempts to offload them to other edge brokers simultaneously. At 1400s, *B1*'s utilization ratios drop to zero because it offloaded all subscriptions to counter the 100% CPU utilization ratio before that. Finally at 1800s, load balancing converges and all of the brokers' performance metrics are within the local triggering threshold, which was set to 0.1. The imbalance at 3000s is neutralized automatically by the load balancing algorithm and arrives at a balanced state at 3400s in the experiment. Although not shown here, by load balancing on the input and output utilization ratios, the input and output queuing delays are also balanced, respectively.
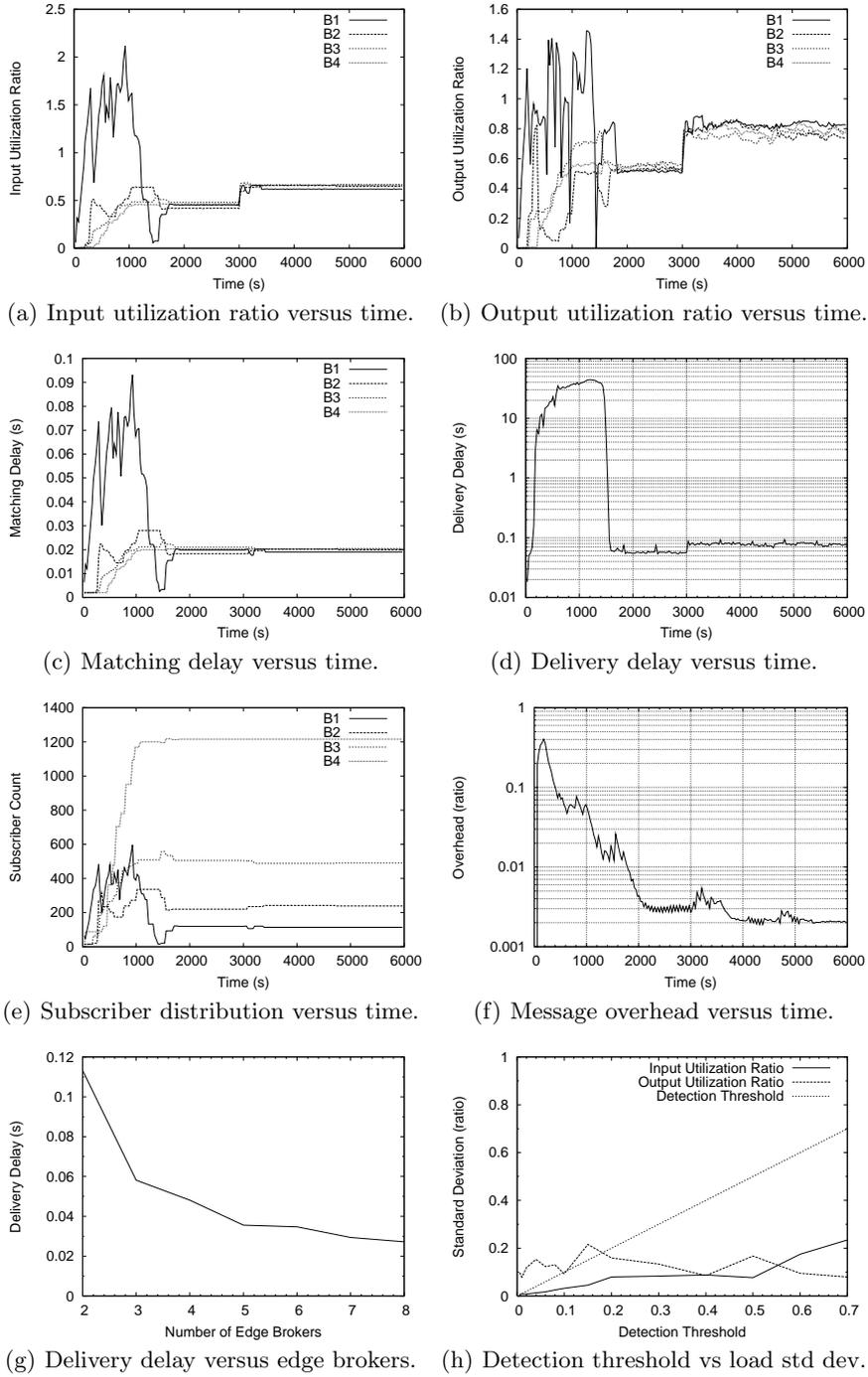
(a) Input utilization ratio versus time.

(b) Output utilization ratio versus time.

(c) Matching delay versus time.

(d) Delivery delay versus time.

(e) Subscriber distribution versus time.

(f) Message overhead versus time.

(g) Delivery delay versus edge brokers.

(h) Detection threshold vs load std dev.

Fig. 6. Local load balancing results.

Client Perceived Delivery Delay Figure 6d shows that an overloaded broker (*B1* in this case) can significantly increase the end-to-end delivery delay. In this experiment, the delivery delay is increased by 750 times. By having a load balancing algorithm in place, this overload period is dramatically reduced and high delay periods are minimized.
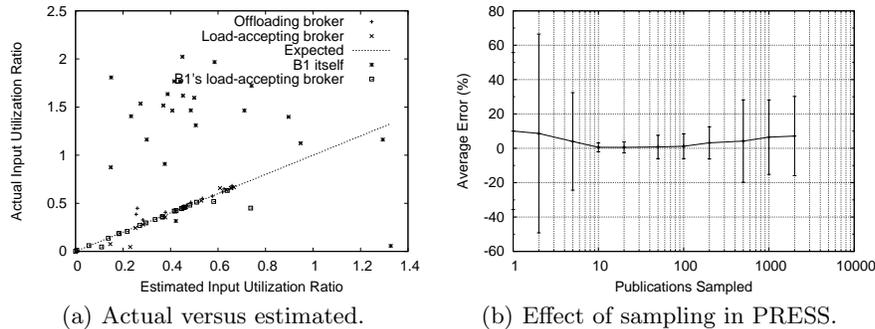
Subscriber Distribution Among Brokers Figure 6e shows that the load balancing algorithm can account for heterogeneous brokers by assigning more subscribers to more powerful brokers. For instance, *B2* services twice as many subscribers as *B1* because *B2* is twice as powerful as *B1*. The same pattern is also observed for brokers *B3* and *B4* relative to *B1*.

Load Balancing Message Overhead Figure 6f shows that the message overhead is 1.5% in the presence of load balancing from 1000s to 2000s, and 0.2% after load balancing has converged. Large spikes in this graph denote large batches of subscribers migrating at that instance in time. The decrease in overhead ratio in the first 2000s is because of the increase in publication traffic routed to new incoming subscribers.

Edge Broker Scalability Figure 6g shows that by increasing the number of edge brokers in a cluster, the delivery delay is reduced because the load balancing algorithm evenly distributes load onto all available resources in the system.

Effect of Detection Threshold on Load Distribution The input utilization ratio difference between all edge brokers is always kept well below the detection threshold as shown in Figure 6h. However, that is not the case for the output utilization ratio because of the stability constraint in the output load balancing algorithm.

Load Estimation Accuracy Figure 7a shows the accuracy of the input utilization ratio estimation. Graphs for the matching delay and output utilization ratio estimation are not shown here because they have the same trends and exception cases as Figure 7a. Dots on the $y = x$ line denote 100% accuracy. Looking at the "offloading broker" and "load-accepting broker" data points, the input utilization ratio accuracy is the lowest of all three performance metrics

(a) Actual versus estimated. (b) Effect of sampling in PRESS.

Fig. 7. Input utilization ratio estimation accuracy.

(a) Avg cluster input utilization ratio.

(b) Avg cluster output utilization ratio.

(c) Avg cluster matching delay.
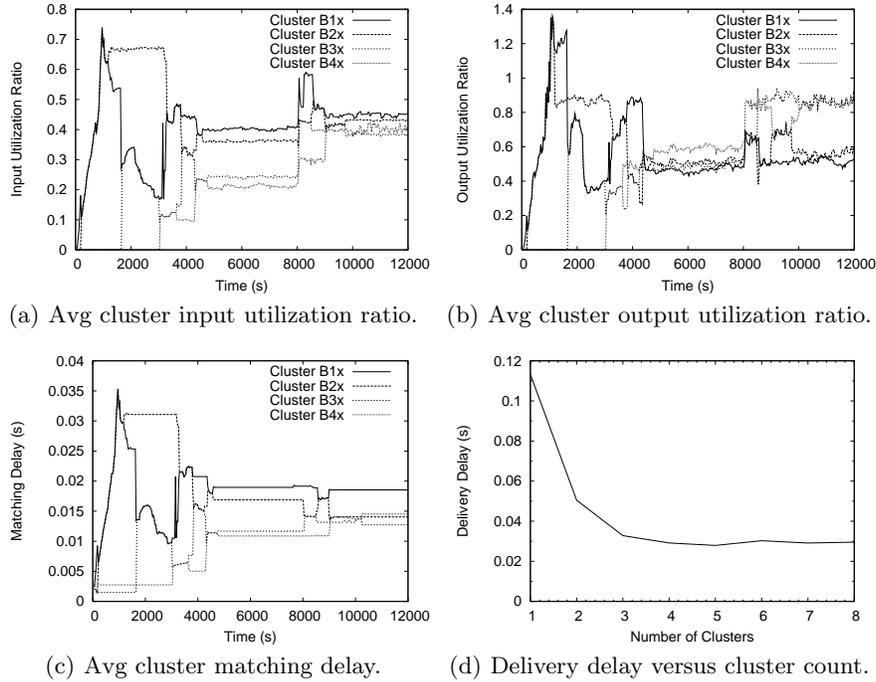
(d) Delivery delay versus cluster count.

Fig. 8. Global load balancing results.

with an average of 91%, including standard deviation. We expect the accuracy for input utilization ratio estimation to be lowest because PRESS' load estimation of the future is based on present data. Estimation points taken from *B1* in the face of incoming subscribers are plotted using different point styles labeled as "`B1 itelf`" and "`B1's load-accepting broker`". These points are underestimated because load estimation does not account for the load imposed by newly incoming subscribers into the system that occurrs between 10s and 1010s. Figure 7b shows that the estimated input utilization ratio reaches closer to 0% error with less deviation as the number of publications sampled increases from 1 to 10. However, beyond 50 samples, the accuracy drops with higher deviations because publications sampled in the early stages no longer accurately portray the matching publication pattern of a subscription when sampling is done.

6.3 Global Load Balancing

The setup used for the global load balancing experiment involves 12 brokers organized into 4 clusters, with 2 edge brokers per cluster. Brokers *B11* and *B12* connect to their cluster-head *B10*, *B21* and *B22* connect to *B20* as their cluster-head, and so forth for *B3x* and *B4x* clusters. Cluster-heads *Bx0* connect to each other sequentially in a chain topology. All clusters have a cluster-head broker with 3000MHz CPU, 1GB RAM, and 10Mbps bandwidth. The CPU speed, memory size, and bandwidth of the edge brokers are given in Figure 5b. At the start

of the experiment, all brokers join the federation. After 5s, 40 unique publishers with a randomly chosen publication rate between 0 and 60msg/min start publishing to broker *B10*. After 10s, 2000 subscribers join broker *B11*. Of the 2000 subscribers, 20% or 400 of them have zero traffic. At 8000s, 50% of the publishers are randomly chosen to have their publication rates increased by 100%. For the experiment on cluster scalability as shown in Figure 8d, each cluster has one cluster-head with 2GHz CPU, 512MB memory, and 10Mbps bandwidth; and two edge brokers with 500MHz CPU, 128MB memory, and 3Mbps bandwidth.

Cluster Load Distribution The average load at each cluster is shown in Figures 8a, 8b, and 8c. Whenever a cluster performs global load balancing with another cluster, the two clusters' loads are merged on the graph because both clusters see the same set of edge brokers. Global load balancing takes longer to converge because the clusters are arranged in a chain topology which limits parallelizing load balancing sessions. For example, cluster *B1x*'s load remains unchanged from 1000s to 3000s when *B2x* load balances with *B3x* as shown in Figure 8a. After 4500s, global load balancing converges, ending up with clusters further away from *B1x* having lesser load. The imbalance at 8000s results in a more balanced state for the input utilization ratios while the output utilization ratios diverges slightly to promote stability of the load balancing algorithm.

Cluster Scalability When clusters are organized in a chain-like topology, there is a load diminishing effect on clusters further away from the source of load, namely cluster *B1x*. Figure 8d shows that with a global detection threshold of 0.15, clusters more than 3 cluster-hops away from cluster *B1x* no longer reduce the overall delivery delay. This is consistent with the idea of preserving subscriber locality at the expense of a fully evenly loaded system.

7 Conclusions

In this paper, we presented a load balancing solution with three main contributions: a load balancing framework, load estimation methodologies, and three offload algorithms. The load balancing framework consists of the PEER architecture, a distributed load exchange protocol called PIE, and detection and mediation mechanisms at the local and global load balancing levels. The core of the load estimation is PRESS, which uses an efficient bit vector approach to estimate the input and output publication loads of a subscription. Each of the three offload algorithms are designed to load balance on a particular performance metric with minimal side-effects and proven stability. Both the load estimation and offload algorithms are independent of the load balancing framework. Our solution inherits all of the most desirable properties that make a load balancing algorithm flexible. PIE contributes to the *distributed* and *dynamic* nature of our load balancing solution by allowing each broker to invoke load balancing whenever necessary. *Adaptiveness* is provided by the three offload algorithms that load balance on a unique performance metric. The local mediator promotes *transparency* to the subscribers throughout the offload process. Finally, load estimation with PRESS allows the offload algorithms to account for broker and subscription *heterogeneity*. Experimental results show that our load balancing

solution is well-controlled and effective at reducing high processing delays resulting from overload conditions while at the same time imposes minimal overhead.

In the near future, we plan on expanding our load balancing scheme onto cluster-head brokers, develop optimizations to our offload algorithms, and explore the possibilities of publisher migration in relation to load balancing.

8   Acknowledgements

References

1. T. Barth, G. Flender, B. Freisleben, and F. Thilo. Load distribution in a corba environment. In *Proceedings of DOA'99*, page 158, Washington, DC, USA, 1999.
2. F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proceedings of the HPDC'96*, page 100, Washington, DC, USA, 1996.
3. A. R. Bharambe, S. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of NetGames'02*, NY, USA, 2002.
4. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM TOCS*, 19(3):332–383, 2001.
5. R. Chand and P. A. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of NCA'03*, page 123, Washington, DC, USA, 2003.
6. Y. Chen and K. Schwan. Opportunistic Overlays: Efficient content delivery in mobile ad hoc networks. In *Proceedings of ACM Middleware'05*, 2005.
7. P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation. In *Proceedings of ICDCS'04*, pages 552–561, Tokyo, Japan, 2004.
8. D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A scalable and highly available web server. In *Proceedings of COMPCON'96*, page 85, Washington, DC, 1996.
9. F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Rec.*, 30(2):115–126, 2001.
10. A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: content-based publish/subscribe over P2P networks. In *Proceedings of ACM Middleware'04*, pages 254–273, NY, USA, 2004.
11. G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of ACM Middleware'05*, 2005.
12. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of ICDCS'88)*, pages 104–111, 1988.
13. G. Mühl. Generic constraints for content-based publish/subscribe systems. In *Proceedings of CoopIS'01*, volume 2172, pages 211–225, 2001.
14. P. R. Pietzuch and J. M. Bacon. Hermes: A distributed event-based middleware architecture. In *Proceedings of the 1st International Workshop on DEBS*, 2002.
15. Vitria. http://www.vitria.com/.