Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware

Peter R. Pietzuch
Peter.Pietzuch@cl.cam.ac.uk

Jean Bacon
Jean.Bacon@cl.cam.ac.uk

University of Cambridge Computer Laboratory
JJ Thomson Avenue
Cambridge CB3 0FD, United Kingdom

ABSTRACT

Overlay broker networks are an important part of an event-based middleware. In this paper, we investigate the requirements of overlay broker networks and argue that using peer-to-peer techniques for their creation and the content-based routing of events has many advantages. We support our claims with an experimental evaluation of Hermes, an event-based middleware architecture that uses a peer-to-peer routing substrate, in comparison with a standard publish/subscribe system that has a simple, predefined overlay topology. The results reveal that Hermes has better routing efficiency and keeps less routing state at the event brokers.

Keywords

event-based middleware, overlay broker networks, content-based routing, simulation, experimental evaluation

1. INTRODUCTION

Publish/subscribe communication [7] is becoming increasingly important in the development of large-scale distributed systems. Almost all traditional synchronous, request/reply middleware platforms that are used today also provide some form of asynchronous messaging that helps to build systems out of loosely-coupled components following the publish/subscribe paradigm. Other middleware platforms, such as JMS [15], Gryphon [1], and web services are strongly based on asynchronous messaging and will be the building blocks for the next generation of internet-wide, ubiquitous applications. In such a world, scalability is paramount and must be ensured at all layers of the system. A messaging middleware may potentially have millions of dynamic clients and, therefore, must itself be implemented in a distributed fashion.

Content-based routing has been established as a powerful concept to disseminate messages, or events, from publishers to subscribers. Most event-based middleware architectures employ an overlay network of event brokers. These brokers then perform content-based routing of events at the application-level since content-based routing at the network-level, such as provided by certain reliable multicast protocols, is not widely deployed yet and often complex to implement in routers.

However, application-level overlay networks face the well-known problem of mapping the logical overlay network onto the physical network topology [4]. An overlay network of event brokers that does not reflect the underlying physical network topology will result in poor performance and little fault-tolerance when disseminating events. So far, this problem has largely been neglected in the context of event-based middlewares. Previous work considered the overlay network topology to be static and left the task of specifying logical connectivity between event brokers to the deployer. This is clearly not a feasible approach when a large-scale system may involve thousands of event brokers running at geographically dispersed sites throughout the Internet. Following recent trends in autonomic, ubiquitous, and peer-to-peer computing, it would be desirable to have self-managing overlay networks of event brokers that are highly scalable and at the same time give good performance in terms of minimising (1) physical network utilisation, (2) event dissemination latency, and (3) routing state kept at brokers.

In this paper, we evaluate the performance of *Hermes* [12], a distributed, event-based middleware that uses peer-to-peer techniques to build and maintain a scalable overlay network of event brokers for event dissemination. We argue that building this network of event brokers on top of a peer-to-peer routing substrate results in several desirable properties for publish/subscribe systems. We formalise the content-based routing algorithm used by Hermes with pseudo code and give details of our Hermes implementation and the implementation of the peer-to-peer routing substrate that it uses (called *Pan*). In addition, we define several metrics that can be used to evaluate the efficiency of event-based middleware architectures. We then compare Hermes with an implementation of a standard Siena-like [3] event dissemination architecture (called *CovAdv*) that uses advertisements and subscriptions and computes covering relations between them. To verify our scalability and efficiency claims of Hermes, we decided to implement both systems in a discrete event simulator (called *DSSim*) that enables us to set up experiments with realistic physical network topologies following the transit-stub model [16] and a large number of event brokers and clients. To our knowledge, this is the first comparison of peer-to-peer overlay networks to traditional

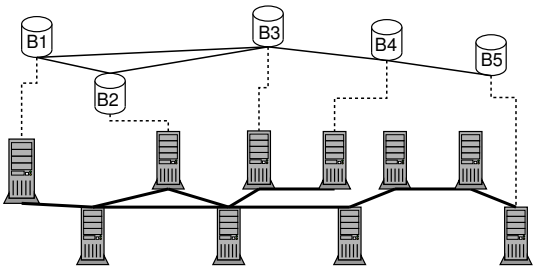

Figure 1: An overlay broker network with 5 brokers mapped onto a physical network topology with 10 nodes

approaches in the area of publish/subscribe systems.

The paper is organised as follows: In Sect. 2, we discuss overlay broker networks in publish/subscribe systems in more detail, in particular, addressing scalability issues and the relation to peer-to-peer systems. Section 3 provides a brief overview of the routing algorithms used by Hermes and CovAdv, focusing on Hermes' type- and attribute-based routing approach. After that, we introduce the implementation of our simulator (DSSim), the peer-to-peer routing layer (Pan), Hermes, and CovAdv (Sect. 4). The main part of the paper presents our evaluation approach and discusses the experimental results comparing Hermes with CovAdv in Sect. 5. The paper finishes with an overview of related work (Sect. 6) and a conclusion (Sect. 7).

2. OVERLAY BROKER NETWORKS

Application-level event brokers are usually deployed in a network with full IP connectivity. However, for scalability reasons and in order to do content-based routing, an event broker only needs to know about a subset of all brokers in the system. Every event broker has such a set of *neighbouring brokers* and it chooses a next hop from this set when it is making a content-based routing decision. The graph formed by the event brokers and the neighbouring relation between them is called the *overlay broker network*. Links to neighbouring brokers have different costs associated with them because a single hop in the overlay network may result in multiple hops (via several IP routers) in the underlying physical network topology with the physical link latency varying from fast LAN links to relatively slow WAN links. Figure 1 shows an overlay broker network on top of a physical network topology. In this example, Broker B_4 has a neighbouring Broker B_5 in the overlay network that is geographically far away in the physical network — the shortest path in the physical network is 6 hops long. Any communication between these two brokers will be expensive in terms of network utilisation.

2.1 Cost Metrics

It is obvious that the efficiency of event dissemination in the event-based middleware is governed by the quality of the mapping between the overlay and the physical network. Content-based routing of events is the process of finding the "best" event dissemination tree in the overlay network that includes all interested subscribers. Even if routing in the overlay network is perfect, it will never be able to do better than the shortest route in the underlying physical network. Various cost metrics can be defined to measure the quality of an event dissemination tree in the overlay network:

(1) Latency. Many applications require an event dissemination tree that minimises the time until all interested subscribers have received a particular event. Therefore, routing paths in the overlay network should only add a small latency penalty compared to routing in the physical network.

(2) Hop Count. Since content-based routing is done by application-level brokers, an increased hop count can be expensive in terms of (1) the delay experienced due to event processing at the brokers, (2) the required processing power at brokers, and (3) the increased likelihood of broker failure. Minimising the number of event brokers involved in an event dissemination tree may therefore be desirable.

(3) Bandwidth. The advantage of overlay networks is that they can share a physical network with other applications. This means that an event dissemination tree should use as little bandwidth as possible leaving more bandwidth for other applications. Especially if the bandwidth of physical links differs, the event dissemination tree should prefer high bandwidth links.

Often, a combination of the previous metrics is used to determine the cost of a route in the overlay network. For instance, the total latency may be the sum of the physical link latencies and broker processing delays introduced at each hop. A bandwidth-oriented metric may prefer physical links that are currently not congested. In the experimental evaluation in Sect. 5, we consider the three simple metrics outlined above.

2.2 Current Approaches

Even though most distributed publish/subscribe systems rely on an overlay broker network, little effort has been spent on deciding how they are created at deployment time.

In Siena [3], a *server topology* determines the interconnections between event brokers in the overlay network. Three different distributed server topologies are given: (1) A *hierarchical* topology is a tree with a root broker. Since there are no redundant paths, each broker is a single point of failure. In addition, the root broker has to cope with a higher amount of traffic limiting the scalability of this topology. (2) The *acyclic peer-to-peer* topology interconnects event brokers forming an acyclic undirected graph. Again, no redundancy is available. (3) In the *generic peer-to-peer* topology, cycles are permitted giving multiple paths to a destination. However, Siena's content-based routing algorithm first executes a distance-vector protocol to create a spanning tree before routing events. It is unclear how this will cope with dynamic topology changes and link failures. The CovAdv implementation that we use for comparison against Hermes follows Siena's approach of an acyclic peer-to-peer topology.

The Gryphon messaging middleware [1, 2] has a multi-broker topology with clusters of event brokers called *cells*. Redundant *link bundles* connect cells with each other forming a generic graph. Although the topology can dynamically evolve over time to a certain extent, the overlay broker network needs to be explicitly specified at deployment time.

The same is true for the JEDI system [5] and the Rebeca project [9]. Both systems have predefined hierarchical tree topologies that are not influenced by the underlying physical network.

2.3 Requirements

Most of the current approaches for building overlay bro-

ker networks like the ones mentioned above have in common that they require the administrator to explicitly provide a (static) list of neighbouring event brokers for each broker. However, this is not realistic in a large-scale deployment of a middleware: (1) An administrator responsible for the event brokers running at a particular site may not be aware of all brokers in the entire system. This makes it hard to choose an optimal set of neighbouring brokers with respect to one of the above cost metrics. (2) It is difficult to ensure global properties of the overlay broker network such as acyclicity, a certain amount of redundancy, or being a minimum spanning tree. Calculating a minimum spanning tree for a large network can be an expensive operation. (3) During the life-time of the system, it may be necessary to change the overlay broker network. This may be because additional event brokers are added in order to improve performance or reliability of the publish/subscribe system, or because the cost of links in the physical network has changed.

Therefore, we propose the following requirements for overlay broker networks in an event-based middleware:

(1) Efficiency. The overlay broker network should encourage the creation of event dissemination trees that are efficient when evaluated under one of the previously introduced cost metrics. If two brokers are equivalent from the perspective of the content-based routing algorithm, the broker with the lower cost metric should be included in the neighbouring broker set.

(2) Redundancy. The overlay broker network should include redundant paths between event brokers to cope with failures of physical links or brokers. Using a spanning tree for an overlay broker network should therefore be avoided and the content-based routing algorithm should be able to handle cycles in the overlay network.

(3) Single Bootstrap Broker. Ideally, when a new event broker is added to the system, it should only be necessary to provide a single address of an already existing broker. The new event broker will then use this broker to bootstrap its set of neighbouring brokers according to the other requirements listed here. This bootstrap broker will not necessarily become part of the neighbouring broker set.

(4) Scalability. The management of the overlay broker network should be scalable so that a large number of event brokers can be supported. The algorithms for adding and removing event brokers from the overlay broker network should only require the participation of a modest subset of all brokers existing in the system. We discuss other scalability requirements that relate to overlay broker networks and content-based routing in Sect. 2.5.

(5) Dynamic Topology. The topology of the overlay broker network should be dynamic instead of static. This means that the content-based routing algorithm has to handle the addition and failure of brokers in the overlay.

In the next section, we show how a peer-to-peer routing substrate creates an overlay network with these desired properties. Our experiments in Sect. 5 will then demonstrate how Hermes benefits from this compared to a conventional content-based routing scheme.

2.4 Peer-to-Peer Overlay Techniques

Recently, distributed hash tables have been adopted to create peer-to-peer overlay networks in distributed systems. Hermes' content-based routing algorithm is built on top of *Pan*, an implementation of a Pastry-like [13] peer-to-peer routing substrate. Here, we briefly sketch Pastry's routing algorithm — a complete description of Pastry and a discussion of its properties can be found in [13].

Each Pan node has a unique node identifier. The main operation provided by the routing layer is `route(message, nodeId)` which routes a message to a node with a particular nodeId. Each node keeps a *routing table* and a *leaf set*. The routing table contains entries for nodes whose nodeIds match the first d digits of the current nodeId and then differ afterwards. Routing can be seen as a generalisation of prefix routing by always sending a message to a node that shares a larger nodeId prefix with the destination nodeId than the current node. The leaf set contains the k nodeIds that are numerically closest to the current node's nodeId. If the destination nodeId falls within the range of the leaf set, the message is directly sent to the numerically closest nodeId. This ensures that if the destination nodeId does not exist in the system, the message will be delivered to the node with the numerically closest nodeId instead.

Event brokers are then installed at Pan nodes and communicate using the peer-to-peer routing layer. Their neighbouring broker set is the union of the Pan node's routing table and its leaf set. From the perspective of overlay networks for an event-based middleware, this satisfies the requirements from the previous section: (1) Routing messages in the peer-to-peer layer is efficient because a node's routing table is filled with entries that are close to it in the physical topology with respect to the cost metric. Messages take $O(\log N)$ hops on average where N is the number of nodes. (2) The overlay topology can handle failures — if the next hop is not responding, a (longer) route via a different node is tried. (3) The protocol for joining new nodes only requires the address of a single existing node. At the end of the join, the new node will have a new routing table and leaf set and all other nodes that need to know about the new node will have updated their data structures. (4) The cost of joining new nodes is small in terms of the number of involved nodes and message exchanges. The overlay network can consists of a large number of nodes. (5) The entries in the routing tables and leaf sets can evolve over time.

2.5 Scalability

One-to-many communication schemes have the advantage of being very scalable since the sender does not need to know the identities of all recipients. However, care must be taken that no layer in the event-based middleware limits the scalability of the entire system. A problem with many content-based routing algorithms that exist today is that they require the *broadcast* of messages to all event brokers in the publish/subscribe system in order to create common state. For example, an advertisement message in Siena will be routed to all brokers if it is not covered by any already existing advertisement. This defeats scalability because it introduces a potentially unknown delay until the message has been successfully processed by all event brokers. This delay depends on the number of brokers in the system. Moreover, these broadcasts limit the robustness of the event-based middleware since a single broker failure will cause the entire application-level broadcast to fail. Therefore, Hermes does not rely on global propagation of information through its overlay broker network.

Another important aspect of scalability is the amount of *routing state* created in event brokers. Broadcasts tend to

create routing state at distant brokers that should ideally not get involved. This is especially true when all the publishers and subscribers that are part of an event dissemination tree are local within a site. In that case, all communication should remain local to that site and no other state at distant brokers should be necessary. In Sect 5, we evaluate the space complexity and distribution of Hermes' routing approach.

3. EVENT-BASED MIDDLEWARE

In this section, we briefly describe Hermes and its content-based routing algorithm. A more detailed explanation can be found in [12]. As a new contribution, we show how *type-based routing* and *type- and attribute-based routing*, which are the two flavours of content-based routing used by Hermes, can be integrated in a single system. A pseudo code representation formalises this.

3.1 Hermes

A system built using Hermes [12] consists of two types of components: *event brokers* and *event clients*. Event brokers form the overlay broker network and implement all the functionality of Hermes. Event clients can be either *publishers* that produce events or *subscribers* that consume them. They are light-weight and need to connect to an event broker before using any of the services provided by the event-based middleware. An event broker that maintains client connections is called a *local event broker* and can be *publisher-hosting*, *subscriber-hosting*, or both. Hermes supports proper event typing so that every published event is an instance of an *event type*. An event type has a type name and a list of attributes. All event types are organised in an inheritance hierarchy and events are type-checked at publication time.

Event brokers communicate with four major kinds of messages: (1) *Type messages* set up rendezvous nodes in the broker network which are event brokers responsible for certain event types as outlined below. (2) *Advertisements* denote a publisher's desire to publish events of a certain type. (3) *Subscriptions* are used by subscribers to express their interest in publications. Finally, (4) *Publications* contain event instances published by publishers. Note that Hermes uses Siena's notion of *coverage* between advertisements and subscriptions, as introduced in [3].

3.1.1 Type- and Attribute-Based Routing

Hermes supports two variants of content-based routing: In the *type-based* (t-based) routing algorithm, subscribers receive all events of a certain type (or any of its subtypes), whereas *type- and attribute-based* (t/a-based) routing allows subscribers to further filter on the event type's attributes. As a result, t/a-based routing creates more filtering state in the system, but has the advantage that published events are not forced to flow via rendezvous nodes. When both routing mechanisms are used together, care has to be taken that the same event is not delivered more than once to an interested subscriber. Figure 2 shows the processing logic for messages at the event brokers in pseudo code that achieves this.

Before an event type can be used for the first time, a type message is sent to the event broker whose nodeId is the hash of the event type name. This broker then becomes the *rendezvous node* (RN) for this type. Rendezvous nodes ensure that flows of advertisements and subscriptions meet in the system. Before a publisher can publish events, an

```
1  processAdvertisement(advMsg):
2  advRT.From ← advRT.From ∪ (advMsg,advMsg.lastNode)
3  IF (advRT.To.covered(advMsg) = ∅) THEN
4    advRT.To ← advRT.To ∪ (advMsg,advMsg.nextNode)
5  ELSE
6    advMsg.nextNode ← null

7  processSubscriptionType(subMsg):
8  [in an analogous manner to processAdvertisement]

9  processSubscriptionTypeAttr(subMsg):
10 processSubscriptionType(subMsg)
11 ∀node IN advRT.From.match(subMsg)
12   IF (node ∉ subRT.To.covered(subMsg)) THEN
13     subRT.To ← subRT.To ∪ (subMsg,node)
14     send(subMsg,node)

15 processPublication(eventMsg):
16 nodeSetType ← advRT.To.match(eventMsg)
17  ∪ subRT.From.match(eventMsg)
18 nodeSetTypeAttr ← subRT.From.match(eventMsg)
19 IF (eventMsg.type) THEN
20   ∀node IN nodeSetType
21     eventTypeMsg ← eventMsg
22     eventTypeMsg.typeAttr ← (node ∈ nodeSetTypeAttr)
23     nodeSetTypeAttr ← nodeSetTypeAttr \ node
24     send(eventTypeMsg,node)
25 IF (eventMsg.typeAttr) THEN
26   eventMsg.type ← false
27   ∀node IN nodeSetTypeAttr
28     send(eventMsg,node)
```

Figure 2: Pseudo code for handling advertisements, subscriptions, and publications in Hermes for type-based and type- and attribute-based routing

advertisement is routed towards this RN via the peer-to-peer routing substrate. At each broker along the path to the RN, the advertisement is processed (line 1). A new entry in the *advertisement routing table* (advRT) is created that records the advertisement and the node where it came from (line 2). If the advertisement is not covered by an earlier one (line 3), it continues to be forwarded to the next event broker on path and this is recorded in the advRT (line 4), otherwise it is dropped (line 6).

T-based subscriptions (line 7) are processed in a similar way except that information about them is stored in the *subscription routing table* (subRT) instead. T/A-based subscriptions (line 9) are first processed like t-based ones (line 10) but then they need to set up filtering state as close to the publishers as possible. Thus, the subscription must follow the reverse path of all matching advertisements (line 11) unless it is already covered by a previous subscription (line 12). If the subscription is sent to a node, a new entry in the subRT is created (line 13). Note that t-based and t/a-based subscriptions do not cover each other.

Publications (line 15) need to be routed differently depending on whether they satisfy t-based or t/a-based subscriptions: In the former case, they follow the forward path of advertisements and the reverse path of subscriptions, in the latter case, they just follow the reverse path of subscriptions and get filtered along the way. A publication has two flags (type and typeAttr) that indicate whether it was being routed according to t-based or t/a-based semantics.

When a publication is processed, the two sets of matching nodes for each of the two cases are calculated (lines 16–18). If the current publication was coming along a t-based flow (line 19), it continues to be routed this way (lines 24). However, it is removed from the t/a-based destination set and its flag is set accordingly if it is on a t/a-based flow,

```
connectPublisherToBroker(publisher,creds)
connectSubscriberToBroker(subscriber,creds)
addEventType(typeowner,creds,eventType)
subscribeType(subscriber,creds,eventType,callback)
subscribeTypeAttr(subscriber,creds,eventType,
  filter,callback)
advertise(publisher,creds,eventType)
publish(publisher,creds,event)
...
```

Figure 3: The public API of a Hermes event broker

as well, in order to avoid duplicate transmission (line 22–23). After that, the publication is sent to the remaining t/a-based destinations (lines 25–28).

4. IMPLEMENTATION

For the experimental evaluation, we implemented Hermes and CovAdv in a distributed systems simulator. In the following, we give details of our implementations and show the Hermes API. We decided to implement our own simulator and not to use a standard network simulator such as ns-2 because we found the scalability of network simulators to be inadequate. In addition, we felt that a more realistic network model would only have complicated our simulation as our evaluation involved rather high-level metrics such as message counts and hop counts.

4.1 Distributed Systems Simulator (DSSim)

Our distributed systems simulator, *DSSim*, is implemented in Java as a standard discrete event simulator. It distinguishes between a *physical network topology* and a *logical network topology* obtained by mapping logical nodes that execute a distributed algorithm onto the physical nodes. It offers a number of simple physical topologies such as a euclidean plane and a spherical topology but also supports transit-stub topologies [16] as generated by the BRITE topology generator [8]. Routing in the physical topology follows a hierarchical two-level distance vector algorithm. Visualisation plug-ins allow the physical and logical topologies to be shown as dynamic graphs during the simulation. The interface used by the logical nodes for communication mainly deals with message-passing and is general enough to be implemented e.g. with JMS or CORBA so that the simulated system can be deployed on a real network.

4.2 Pan

Our Pastry implementation, called *Pan*, is built on top of DSSim. This successfully reproduced the average hop count and distance results of the Pastry routing algorithm from [4] within a small error. Pan provides a slightly more powerful callback interface than Pastry which includes information about the last hop. Furthermore, we added a mechanism for request/reply interaction between two Pan nodes that is required by Hermes to set up rendezvous nodes.

4.3 Hermes

The API exported by a Hermes event broker is shown in Fig. 3. For simplicity of exposition, we ignore the complementary functions to disconnect, unadvertise, etc. and any of the error exceptions. Every function takes the caller's identity and a set of credentials to perform access control. Two functions are provided for clients to `connect` to an event broker. The `addEventType` function sets up a new rendezvous node for an event type. Two subscription functions are included that correspond to t-based and t/a-based subscriptions. Finally, `advertise` and `publish` enable publishers to carry out the corresponding operations.

Hermes uses the Pan API for sending and receiving messages. However, when Hermes' content-based routing algorithm performs reverse-path forwarding [6] of messages, it does not use Pan's `route` operation but sends the messages directly to the destination. Routing these messages with Pan would substantially increase the hop count of an already set up event dissemination tree.

4.4 CovAdv

CovAdv, our Siena-like publish/subscribe implementation, has the same external API as Hermes. It uses advertisement messages sent by publishers, subscription messages that follow the reverse path of advertisements, and publications that follow the reverse path of subscriptions. Covering between advertisements and subscriptions in routing tables is implemented and, for this, the same code is shared between Hermes and CovAdv.

CovAdv's overlay broker network is statically defined at deployment time by a list of event brokers for the neighbouring broker sets. Advertisements are sent to all brokers in this set unless a covering advertisement exists. We decided to only support acyclic graphs for overlay broker networks in CovAdv. This is because the implementation of an incremental distance-vector algorithm, which would be needed otherwise, turned out to be complex. It would have required patching up subscription and advertisement routing tables whenever a more efficient path was discovered. Instead, CovAdv's overlay broker network was arranged to be a minimum spanning tree.

5. EVALUATION

This section presents the experimental results that we obtained from comparing our Hermes implementation with CovAdv. We start by describing the experimental setup in terms of topologies and parameters. After that, we justify our choice of experiments and discuss the four experiments that evaluate Hermes and its overlay broker network.

5.1 Experimental Setup

All experiments were carried out in our simulator, DSSim. The underlying physical topology was a transit-stub topology generated by BRITE with 1000 nodes consisting of 10 autonomous systems with 100 nodes each. Event brokers, publishers, and subscribers were randomly assigned to physical nodes unless stated otherwise. We varied the *number of event brokers* (n_e), the *number of publishers* (n_p), the *number of subscribers* (n_s), and the *number of event types* (n_t) in the different experiments. Each publisher published n_{pt} different event types and each subscriber subscribed to n_{st} types using a type- and attribute-based subscription. We always compared the result of Hermes against a run of CovAdv with the same parameters. Each data point in the plots is an arithmetic mean of 5 runs.

The overlay broker network in the case of Hermes was formed by sequentially adding event brokers. The bootstrap broker chosen for each new addition was the closest (in terms of network latency) already existing broker in the topology. Pan used nodeIds with 4 digits of base 4 and the leaf set size was 4. For CovAdv, we precomputed a

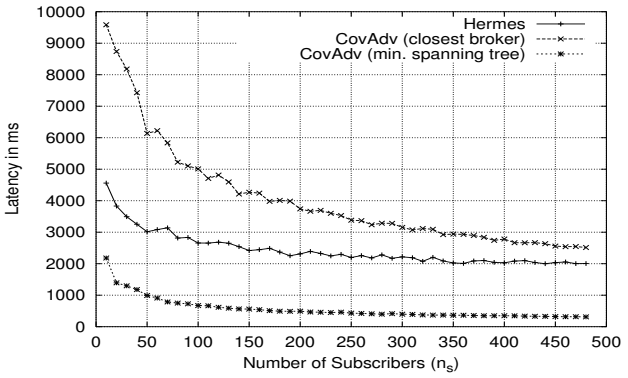

Figure 4: E1: Latency per event notification versus number of subscribers (n_s)

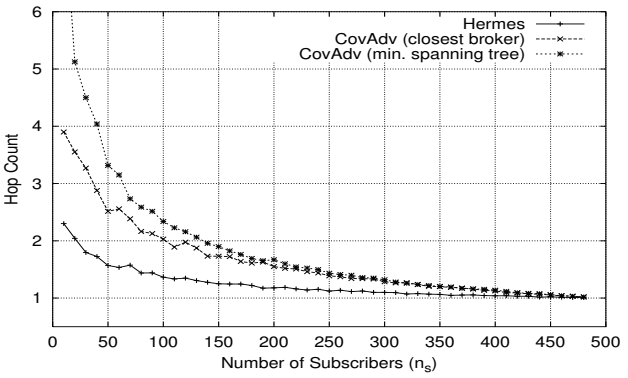

Figure 5: E1: Hop count per event notification versus number of subscribers (n_s)

minimum spanning tree and used this for the neighbouring broker set. Even though this gave CovAdv the advantage of having an optimal overlay broker network, it only affected the first experiment.

5.2 Experiments

The four experiments (E1–E4) investigate different aspects of event dissemination in a publish/subscribe system. The first experiment E1 looks at the efficiency of event routing in the system in terms of latency and number of hops (cost metrics (1) and (2) from Sect 2.1) in order to evaluate the quality of the overlay broker network built by Hermes. Following our requirement of minimising routing state in the system (cf. Sect. 2.5), experiment E2 looks at routing tables sizes. In a similar manner, experiment E3 addresses the distribution of routing tables sizes across all brokers. Finally, we consider bandwidth (cost metric (3)) and measure the number of messages sent by both Hermes and CovAdv when publishing a fixed number of events.

5.2.1 E1: Routing Efficiency

In this experiment, the number of event brokers $n_e = 500$, the number of publishers $n_p = 10$, the number of event types $n_t = 1$, and each publisher/subscriber operated on this single event type. We varied the number of subscribers $n_s = 10 \ldots 500$ and assigned at most *one* subscriber per event broker. We ran the experiment three times: with Hermes, with CovAdv using the same bootstrap broker as Hermes, and with CovAdv using a minimum spanning tree for its overlay broker network.

Figure 4 shows the *experienced latency per received notification* in ms as n_s varies. In all three cases, the average latency decreases as more subscribers are added to the system because the event dissemination tree becomes more densely populated. In a realistic deployment of CovAdv in which it uses the closest event broker for its neighbouring broker set, it shows the largest latency times. Especially when the system is sparsely populated with subscribers, CovAdv has a poor latency behaviour because its overlay broker network does not adequately reflect the physical topology. Hermes does a better job and is less dependent on the number of subscribers. As expected, the lowest latency is achieved by CovAdv when using a precomputed minimum spanning tree.

In Fig. 5, we plot the average *hop count per received notification.* Here, Hermes is most efficient because of its larger neighbouring broker sets. Note that even when $n_s = 10$, Hermes only uses $\log_{16}(500) = 2.24$ hops on average. As the number of subscriber-hosting brokers increases, all three routing approaches converge towards a single hop because a notification occurs after each routing step.

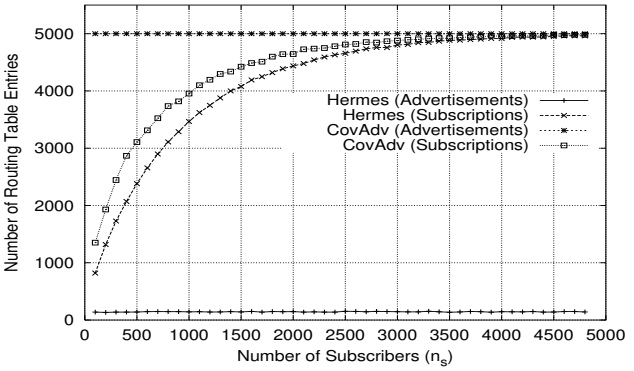

Figure 6: E2: Routing tables sizes versus number of subscribers (n_s)

5.2.2 E2: Space Efficiency

The next experiment plotted the *total number of entries in the advertisement* (advRT) *and subscription* (subRT) *routing tables* at all the brokers as a function of the number of subscribers ($n_s = 100 \ldots 5000$). The other parameters were: $n_e = 500$, $n_p = 10$, $n_t = 10$, and each publisher/subscriber used 5 different event types ($n_{st} = n_{pt} = 5$). As can be seen from Fig. 6, Hermes creates a fraction of the state of CovAdv in advRTs because it does not broadcast advertisements to all brokers. Because the number of publishers does not change, the number of entries in the advRTs stays constant. Due to Hermes' more efficient routing in the overlay network, it can better take advantage of subscription coverage and therefore uses slightly less state in the subRTs than CovAdv. When n_s becomes large, the routing tables get completely filled and thus converge towards the same value.

We also varied the number of publishers ($n_p = 100 \ldots 5000$) and kept the number of subscribers fixed ($n_s = 100$) (cf. Fig. 7). Again, the sum of advRT entries for CovAdv is constant because the advertisements broadcasts create a complete set of advRT entries at all the brokers. In contrast to that, advRT entries in Hermes scale sub-linearly with

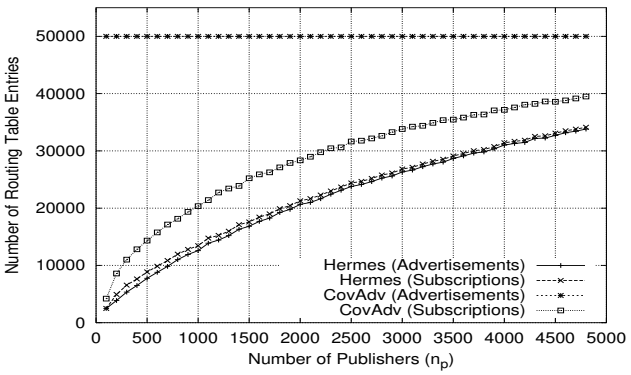

Figure 7: E2: Routing tables sizes versus number of publishers (n_p)

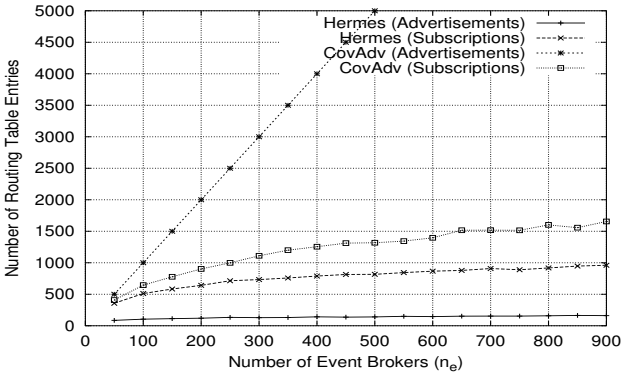

Figure 8: E2: Routing table sizes versus number of event brokers (n_e)

the number of publishers. Since t/a-based subscriptions are used which cause more filtering state as more publishers are added to the system, the amount of state in subRT increases sub-linearly, as well.

Finally, we changed the size of the overlay broker network ($n_e = 50 \ldots 1000$) and kept the other parameters fixed ($n_p = 10$ and $n_s = 100$) to see how scalable the overlay broker network itself is. The plot in Fig. 8 clearly points out that most additional state is created by CovAdv advertisements. Hermes' overlay broker network scales more favourably.

5.2.3 E3: Space Distribution

In the third experiment E3, we investigated the *distribution of routing table sizes* in the system. For this, all parameters were kept constant: $n_e = 500$, $n_p = 10$, $n_s = 1000$, $n_t = 10$, and $n_{st} = n_{pt} = 5$. In Hermes, the majority of event brokers had 10 routing table entries, as shown in the distribution plot in Fig. 9. No broker had more than 15 entries and about 40 brokers were not involved in routing, at all. The CovAdv implementation spreads out routing table state much more through the system — a vast majority of brokers had a maximum of 20 routing table entries and the minimum number of entries was 10.

5.2.4 E4: Message Complexity

The last experiments carried out counted the *numbers of messages* (advertisements, subscriptions, and publications) that were sent as the number of subscribers and publishers

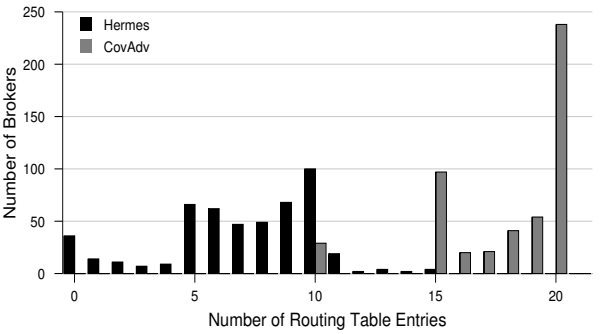

Figure 9: E3: Distribution of routing tables sizes

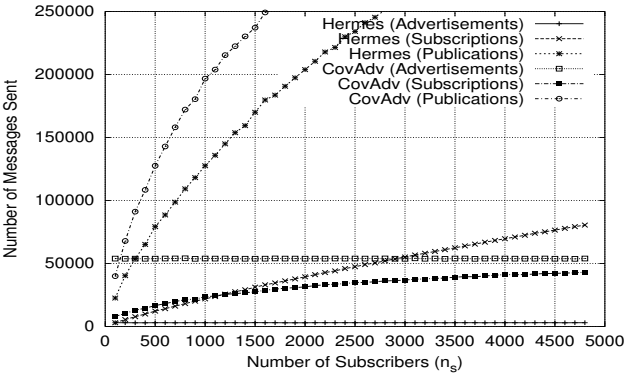

Figure 10: E4: Number of messages sent versus number of subscribers (n_s)

changed. We used the same number of event brokers as before ($n_e = 500$) but increased the number of distinct event types ($n_t = 100$, $n_{st} = n_{pt} = 10$) in order to reduce covering and have more sent messages. In Fig. 10, the number of subscribers was $n_s = 100 \ldots 5000$ with $n_p = 100$ publishers. It is interesting to observe that Hermes sends significantly fewer publications than CovAdv due to its more efficient content-based routing. However, it is less scalable in terms of subscriptions sent because of the additional messages that are sent to reach rendezvous nodes. As discussed before, the counts of advertisements stay constant with more messages being sent by CovAdv.

In Fig. 11, when the number of publishers is variable ($n_p = 100 \ldots 5000$ and $n_s = 10$), a similar behaviour occurs. However, the number of advertisement messages increases slowly for both Hermes and CovAdv because there is less coverage due to the larger number of event types.

5.3 Summary

We feel that the results in this section substantiate some of the earlier scalability claims made about Hermes. Several interesting results were obtained: (1) Hermes' overlay broker network gives a good compromise in terms of latency compared to a minimum spanning tree. It is more efficient in terms of hop count which is especially important for application-level routing. (2) A large portion of the state at brokers in a publish/subscribe system is created by advertisements. This can be unnecessary since it assumes the worst case of all event brokers hosting subscribers. Using rendezvous nodes to ensure that subscriptions meet with ad-

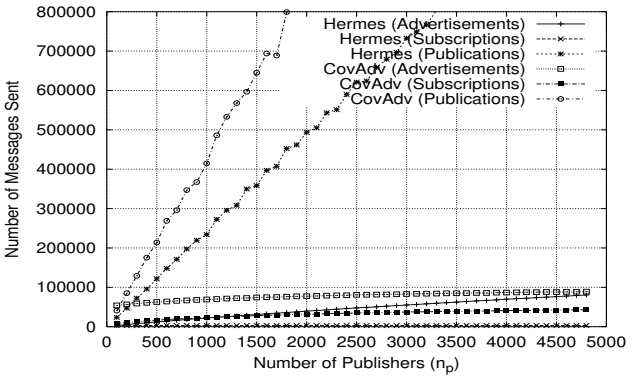

Figure 11: E4: Number of messages sent versus number of publishers (n_p)

vertisements can be less expensive. (3) Finally, the number of messages is reduced if the routing efficiency of the overlay broker network is improved despite Hermes' need for extra subscription messages for rendezvous nodes.

6. RELATED WORK

Some related work exists on the evaluation of publish/subscribe systems. In [3], Siena is evaluated using an approach comparable to ours. However, its overlay broker network closely follows the physical network topology which may not be achievable in reality. The evaluation does not include notification latency or routing state.

In contrast to that, a non-simulational approach is taken by the Rebeca project [10, 9]. An extensive comparison of different content-based routing algorithms is provided and includes a first attempt at formalising publish/subscribe semantics. Unfortunately, the cost of routing in the physical network topology is not addressed and only a single publisher in a tree-like overlay broker network is assumed.

The work on Pastry [13] and Scribe [14] deals with the quality of peer-to-peer overlay networks for routing and application-level multicast. However, Scribe does not include content-based filtering of messages and thus creates less state in the system. Gryphon comes with a detailed evaluation of IP multicast for content-based publish/subscribe [11]. However, IP multicast does not create an application-level overlay broker network.

7. CONCLUSION

As publish/subscribe systems grow in size, in addition to efficiency and scalability of the event-based middleware, the deployment and management of overlay broker networks will become a major concern. Ideally, as little administrative intervention as possible should be required when event brokers are deployed — but without compromising routing efficiency. In this paper, we focused on the requirements of overlay broker networks. Hermes follows these requirements with its type- and attribute-based routing algorithm on top of a peer-to-peer routing substrate. Several experiments compared Hermes with a traditional publish/subscribe system in terms of efficiency and state. As future work, we would like to extend the delivery semantics of Hermes to provide guarantees in case of failure in the overlay broker network. This will add the notion of persistent events to the event-based middleware.

8. ACKNOWLEDGEMENTS

Peter Pietzuch is funded by UK EPSRC and QinetiQ, Malvern. The authors would like to thank the members of the Opera Group, in particular András Belokosztolszki and Brian Shand for their useful discussions.

9. REFERENCES

[1] G. Banavar, T. Chandra, et al. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *ICDCS*, pages 262–272, 1999.

[2] S. Bhola, R. Strom, S. Bagchi, et al. Exactly-once Delivery in a Content-Based Publish-Subscribe System. In *Proc. of DSN'02*, pages 7–16, 2002.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Trans. on Computer Systems*, 19(3):332–383, Aug. 2001.

[4] M. Castro, P. Druschel, et al. Exploiting Network Proximity in Peer-to-Peer Overlay Networks. Technical report, MS Research Cambridge, 2002.

[5] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI Event-Based Infrastructure and its Applications to the Development of the OPSS WFMS. *IEEE Trans. on Software Engineering*, 27(9):827–850, Sept. 1998.

[6] Y. K. Dalal and R. M. Metcalfe. Reverse Path Forwarding of Broadcast Packets. *Comm. of the ACM*, 21(12):1040–1047, Dec. 1978.

[7] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. Technical report, EPFL, Lausanne, Switzerland, 2001.

[8] A. Medina, A. Lakhina, et al. BRITE: An Approach to Universal Topology Generation. In *Proc. of MASCOTS'01*, pages 346–356, 2001.

[9] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, TU Darmstadt, 2002.

[10] G. Mühl, L. Fiege, F. Gärtner, and A. Buchmann. Evaluating Advanced Routing Algorithms for Content-Based Publish/Subscribe Systems. In *Proc. of MASCOTS'02*, pages 167–176. IEEE Press, 2002.

[11] L. Opyrchal, M. Astley, et al. Exploiting IP Multicast in Content-Based Publish-Subscribe Systems. In *Proc. of Middleware*, pages 185–207, 2000.

[12] P. R. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 611–618, Vienna, Austria, July 2002.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of Middleware 2001*, pages 329–350, Nov. 2001.

[14] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The Design of a Large-Scale Event Notification Infrastructure. In *Proc. of the 3rd Int. Workshop on Networked Group Communication (NGC'01)*, Nov. 2001.

[15] Sun. Java™ Message Service. Specification, Sun, 2001. http://java.sun.com/products/jms/.

[16] E. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. of INFOCOM'96*, pages 594–602, San Francisco, USA, 1996.