Hermes: A Distributed Event-Based Middleware Architecture

Peter R. Pietzuch* and Jean M. Bacon
Computer Laboratory
University of Cambridge
JJ Thomson Avenue, Cambridge CB3 0FD, UK
{Peter.Pietzuch, Jean.Bacon}@cl.cam.ac.uk

Abstract

In this paper, we argue that there is a need for an event-based middleware to build large-scale distributed systems. Existing publish/subscribe systems still have limitations compared to invocation-based middlewares. We introduce Hermes*, a novel event-based distributed middleware architecture that follows a type- and attribute-based publish/subscribe model. It centres around the notion of an event type and supports features commonly known from object-oriented languages like type hierarchies and supertype subscriptions. A scalable routing algorithm using an overlay routing network is presented that avoids global broadcasts by creating rendezvous nodes. Fault-tolerance mechanisms that can cope with different kinds of failures in the middleware are integrated with the routing algorithm resulting in a scalable and robust system.*

1. Introduction

Middleware systems like CORBA or Java RMI have proven to give a useful abstraction for building complex distributed applications. They provide a common higher-level interface to the application programmer and hide the complexity of dealing with a variety of underlying platforms and networks. Today, most middleware systems are *invocation-based* and thus follow a *request/reply paradigm*: A client requests a particular service from a server by either sending a request message or performing a remote method invocation (RMI) and then receives a reply in return. Although such a mode of operation works well in a local area network (LAN) context with a moderate number of clients and servers, it does not scale to large networks like the Internet. The request/reply paradigm only allows a one-to-one communication model and forces a tight coupling between the involved parties. Such behaviour is not desirable on the Internet because of the large number of potential communication partners, and the dynamic nature of all interactions with new clients joining the system and servers failing.

A different communication approach for large-scale distributed systems seems necessary. As argued in [14], event-based, publish/subscribe (pub/sub) systems are a viable new option. In such a system, events are the basic communication mechanism: First, event subscribers, i.e. clients, express their interest in receiving certain events in the form of an event subscription. Then, event publishers, i.e. servers, publish events which will be delivered to all interested event subscribers. As a result, this model naturally supports a decoupled, many-to-many communication style between publishers and subscribers. A subscriber is usually indifferent to which particular publishers supply the events that it is interested in. Similarly, a publisher does not need to know about the set of subscribers that will receive a published event.

Even though a number of pub/sub systems [8, 4, 23, 12] have been developed over the past years, we feel that little work has been done to unite the areas of middleware systems and publish/subscribe communication and to provide, what we call, an *event-based middleware*. The existing systems often lack traditional middleware functionality like type-checking of invocations, reliability, access control, transactions etc. They integrate poorly with object-oriented languages like Java or C++ that are used to implement distributed systems on top of a middleware and force the programmer to deal with low-level event transmission issues. The application examples for current pub/sub systems are often very restricted, such as instant-messaging and stock quote dissemination. This reflects that these systems are not intended as general middleware platforms. Our work focuses on providing a scalable event-based middleware that is powerful enough to be the building layer for any distributed application that would traditionally be implemented with an invocation-based middleware. We envision a world with large-scale e-commerce and business applications with thousands of components operating over the

*Research supported by QinetiQ, Malvern

Internet in a highly heterogeneous environment. The event-based publish/subscribe paradigm is a good choice for realising such applications when used in connection with a middleware layer.

This paper presents our idea of an event-based middleware by specifying requirements (Section 2) and describing the design of *Hermes* (Section 3), an event-based middleware being developed in our research group. We show how overlay networks are a useful abstraction (Section 3.1) and introduce the notion of *type- and attribute-based* publish/subscribe (Section 3.2) to bridge the semantic gap between events and programming language types without sacrificing the benefits of the pub/sub model. A type- and attribute-based routing algorithm that is built on top of a peer-to-peer overlay routing network is presented in Section 4. We claim that this approach is more scalable and fault-tolerant than existing systems as it does not rely on any global broadcast operations. Then, a novel mechanism for dealing with communication and node failures within the event-based middleware is shown (Section 4.3), thus resulting in a fault-tolerant system. The paper finishes with an overview of related work (Section 5) and a conclusion with an outline of future work (Section 6).

2. Requirements

In general, the goal of an event-based middleware is to support the implementation of large-scale distributed applications. This means that its requirements closely reflect the desired functionality of a modern middleware system. The event-based communication style provides different techniques to realise these requirements that must be taken into account at any design stage. In this section, we list some important middleware features, motivate them and explain how they can be achieved in a system in which events are the principal communication paradigm. Although requirements like scalability and expressiveness are addressed by existing pub/sub systems [7], traditional middleware features like interoperability, reliability and usability are largely ignored.

Scalability: A scalable middleware must be able to support a large number of clients and servers; this is a crucial requirement for Internet-scale distributed applications. A system is only truly scalable if all its components are, so that several aspects can be identified to ensure the scalability of an event-based middleware: First, the implementation of the middleware (e.g. matching events with subscriptions) has to be distributed itself because any centralised component or service can become a bottleneck. Second, no global state is to be kept by all middleware components, which means that any decision made by a component (e.g. for event routing) has to rely on a local view of the world only. Finally, resources like network bandwidth and memory must be consumed efficiently. For example, events should only be sent over the network if a subscriber is interested in them.

Interoperability: The whole idea of a middleware is to facilitate interoperability between heterogeneous components over a network. As a result, an event-based middleware should be language- and platform-independent and not rely on any particular support by the underlying network like e.g. IP multicast. It should be able to operate in a dynamic environment where components of the distributed system are implemented on a variety of fixed and mobile devices that join and leave the system at run-time. The event model should not be tied down to any particular language.

Reliability: Different clients of the event-based middleware will have different requirements regarding reliability and quality of service (QoS) guarantees given by the middleware. A range of operation modes from "best-effort" to "guaranteed and timely" event delivery must be supported. Fault-tolerance mechanisms have to be part of the middleware design so that isolated network or component failures do not affect the entire system. Techniques such as persistent events and replication help to achieve a more robust middleware implementation.

Expressiveness: Distributed applications benefit from an expressive model for specifying events and subscriptions. Subscriptions based on event data (content-based filtering) provide a fine-grained mechanism for subscribers to express their information need. Similarly, composite event expressions that detect patterns of events give an intuitive and powerful higher-level abstraction to subscribers.

Usability: It is important that the abstraction given by the middleware integrates cleanly with the application programming language so that it is easy to use. This linguistic support involves an intuitive mapping between events and programming language objects, support for static and dynamic type checking of subscriptions and publications, and the hiding of middleware implementation issues like internal event formats. Tools for constructing complex composite event expressions and querying event type repositories further aid the development process.

3. Design of an Event-Based Middleware

This section presents *Hermes*, a distributed event-based middleware architecture, and explains its design and the features that make it different from existing pub/sub systems. Figure 1 illustrates a distributed system implemented on top of Hermes. It consists of two components, *event clients* and *event brokers*. Event clients can be *event publishers* or *event subscribers* and use the services provided by the middleware to communicate using events. The event brokers represent the actual middleware and provide a distributed implementation of the functionality required by the event clients.

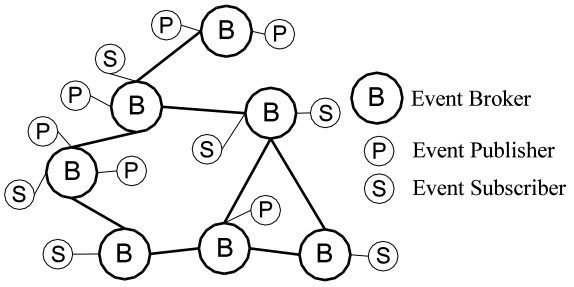

Figure 1. An Application built with Hermes

Since the entire functionality of the middleware is provided by the brokers, event clients are light-weight components, which can easily be implemented in any application programming language. They connect to an event broker before they use the middleware service. The main task of the event brokers is to accept subscriptions from event subscribers and then deliver events from publishers to all interested subscribers. Brokers are interconnected with each other in an arbitrary topology and use message-passing to communicate with their neighbours. Published events are translated into messages which are then routed through the network of brokers depending on their content and the existing subscriptions [9]. Because of scalability, subscriptions are pushed towards the publishers in the network so that filtering happens as close to the event publisher as possible (*source-side filtering*).

In any distributed pub/sub system, an *event dissemination tree* must be dynamically constructed so that events can be routed from publishers to all interested subscribers. An open research question is how this tree is built and where state is kept in the network. Usually, some form of *advertisement* mechanism helps the routing process: Before an event publisher is allowed to publish an event, it has to advertise its presence to the system by sending an advertisement message. Then, in connection with existing subscriptions, a dissemination tree is created and future published events can follow the paths set up by the tree. Hermes proposes a novel approach to create a tree by employing an *overlay routing network* that manages rendezvous nodes for advertisements and subscriptions in the network.

3.1. Overlay Routing Networks

An *overlay routing network* is a logical application-level network that is built on top of a general network layer like IP unicast. The nodes that are part of the overlay network can route messages between each other through the overlay network. There is an overhead associated with using a logical network for routing, as the logical topology does not necessarily mirror the physical topology. However, more sophisticated routing algorithms can be used and deployed since routing is implemented at the application level. A number of scalable overlay routing infrastructures were developed [21, 26], that provide higher-level services than IP-level routing, such as multicast, fault-tolerance, location-independence, and anonymity.

Since a network of event brokers behaves like an overlay network for content-based event dissemination, it is natural to use the abstraction of an overlay network layer for building an event-based middleware. Such a network can then dynamically adapt its topology during the lifetime of the system. In Hermes, the event brokers form the nodes of an overlay routing network that is similar to Pastry [21]. Each node has a unique random numerical identifier associated with it. The main abstraction provided by the overlay routing layer is a

```
route (message, destination_id)
```

function that allows a broker to send a message to another broker with a particular id which is then routed via the overlay network. If a node with this identifier does not exist, the message is delivered to the node with the numerically closest id.

Using an overlay network has several advantages: First, the fault-tolerance mechanisms provided by the overlay network are used to manage the logical network of brokers. Link and node failures are dealt with transparently by the overlay network. Second, the connection and disconnection of brokers to/from the network is handled by the overlay layer. Finally, the overlay routing operation allows brokers to find rendezvous points for building event dissemination trees, as explained in the next paragraph.

As mentioned before, advertisements are generated by event publishers to create an event dissemination tree. Subscriptions need to be joined with advertisements in the network so that routing paths for future notifications are set up. Different methods have been proposed to do this [8, 4, 12], but they rely on either broadcasting advertisements or subscriptions through the entire network of brokers. These global broadcasts are not scalable and will lead to inconsistent system state when network partitions occur. In contrast, Hermes uses *rendezvous nodes* in the network, which are special event brokers that are known to both publishers and subscribers. They function as meeting points for advertisements and subscriptions, similar to cores in Core-Based Multicast Trees [3]. For each event type, a rendezvous node exists in the network. To find a particular rendezvous point, a hash value of the event type name is calculated, and the result is the node id of the rendezvous node. The `route` function is then used to send an advertisement or subscription to the rendezvous node. No global knowledge is required. To prevent rendezvous nodes from being single points of failure, they are replicated throughout the network of brokers (Section 4.3).

3.2. Type- and Attribute-Based Publish/Subscribe

Traditionally, there has been an impedance mismatch between programming language objects and events. Most content-based pub/sub systems view events as untyped collections of attribute/value pairs, but modern programming languages only support statically or dynamically typed objects. As a result, an event-based middleware should support proper *event typing* so that events can be treated as first-class programming language objects [16]. The Cambridge Event Architecture (CEA) [1, 19] was developed with this idea in mind, and Hermes follows its approach by associating every event and subscription with an *event type* that is type-checked at runtime. The event type contains a number of data fields (i.e. attributes). We call the underlying model *type- and attribute-based* publish/subscribe.

Publish/subscribe systems can be divided into *topic-based* and *content-based* systems. Topic-based systems allow the subscriber to specify a topic of interest, but have the shortcoming that no filtering at a finer granularity can be made. On the other hand, content-based subscriptions allow filtering depending on event data, but in a large-scale system, grouping events into related types (i.e. topics) would help to manage a large number of different events. Therefore, a type- and attribute-based system does a combination of both: first, the event subscriber specifies the event type (i.e. topic) it is interested in, and then supplies a filter expression that operates on the attributes provided by this event type. Since the middleware knows the event type and its definition, it can type-check events and subscriptions at runtime, and inform the user about any mismatches. This helps to build a more robust distributed system, especially in an environment where event types are evolving.

Each event type is managed by an event broker that functions as the rendezvous node for this type. Event types are organised into *event type hierarchies* similar to class hierarchies in an object-oriented language. This means that event types can be derived from each other using inheritance to create more specialised types. A subscription that operates on a parent type will also match all events that are of a descendent type (*supertype subscription*). However, no global type hierarchy is enforced so that several independent hierarchies with distinct root types can exist. A single global hierarchy could not be enforced on an Internet-scale.

3.3. Architecture

The architecture of Hermes follows a layered approach as shown in Figure 2. The middleware is assumed to be deployed on an IP unicast network like the Internet. On top of that, an overlay routing network between the event brokers is established. This routing network enables the *type-based pub/sub layer* to set up rendezvous nodes that manage particular event types. The functionality provided by this layer is that of a topic-based pub/sub system. Filtering depending on the event data is then implemented by the *type- and attribute-based pub/sub layer*. It distributes filter expressions through the network of event brokers to achieve source-side filtering on event attributes.

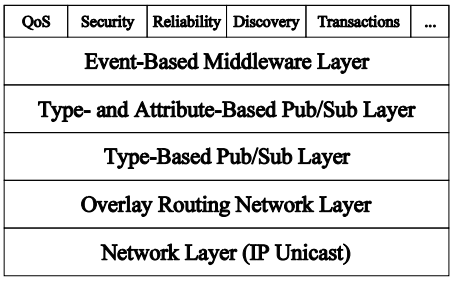

Figure 2. The Layered Architecture of Hermes

The *event-based middleware layer* provides the API that programmers use to implement applications. It allows the programmer to advertise, subscribe to, or publish events, to add or remove event types from the system, and it performs type-checking of events and subscriptions. The middleware layer consists of several modules that implement further middleware functionality such as fault-tolerance, reliable event delivery, event type discovery, security, transactions, mobility support etc.

4. Event Routing Algorithms

The choice of algorithm for event dissemination strongly determines the overall scalability of an event-based middleware. Hermes uses a scheme that is more scalable than existing approaches because it does not require global broadcasts. We first present the algorithm implemented by the type-based pub/sub layer which disseminates events solely based on their type, and then outline the extensions by the type- and attribute-based pub/sub layer that supports content-based filtering. Both pub/sub layers exchange four kinds of messages using the overlay network layer:

Type Messages add new event types to the system and set up rendezvous nodes for them. A type message contains the definition of an event type which is stored at a rendezvous node. Published events can be type-checked against this definition.

Advertisement Messages denote an event publisher's capability of publishing a certain event type. They set up event dissemination paths in the broker network and are routed towards rendezvous nodes.

Subscription Messages express a subscriber's interest in certain events. In connection with advertisements, they cre-

ate paths for events through the network of brokers and are routed towards rendezvous nodes. They may follow the reverse path of advertisements for filtering, as described later.

Publication Messages carry published events in message form. They are sent by event publishers and follow the paths created by advertisement and subscription messages.

4.1. Type-Based Routing

The type-based routing algorithm works as follows: Before an event can be published, the corresponding rendezvous node must be set up by routing a *type message* to the event broker, whose numerical id is the hash of the event type name. Figure 3 shows an overlay network with two subscribers $S_{\{1,2\}}$, two publishers $P_{\{1,2\}}$ and six event brokers, $B_{\{1..5\}}$ and R. A rendezvous node R has been set up by a type message t_2 coming from publisher P_2. Now, P_1 and P_2 can send *advertisement messages*, a_1 and a_2, to the rendezvous node R. The two subscribers subscribe to this event type (i.e. topic) by routing two *subscription messages*, s_1 and s_2, to the rendezvous node. Each broker along the path of an advertisement or subscription message keeps state about the messages it has forwarded. It stores the identifiers of the brokers that send or were sent the message. This gives a distributed history of all the message flows through the network.

Finally, publisher P_1 sends a publication message p_1 containing an event. The message follows the advertisement path up to the rendezvous node R. Whenever it reaches a broker that contains state about a subscription of the same type, the publication message follows the *reverse path* [13] of this subscription and, thus, reaches all subscribers. Publications are not necessarily routed through the rendezvous node, which could otherwise become a bottleneck. Rendezvous nodes are a way to ensure that advertisement and subscription messages will meet in the network so that an

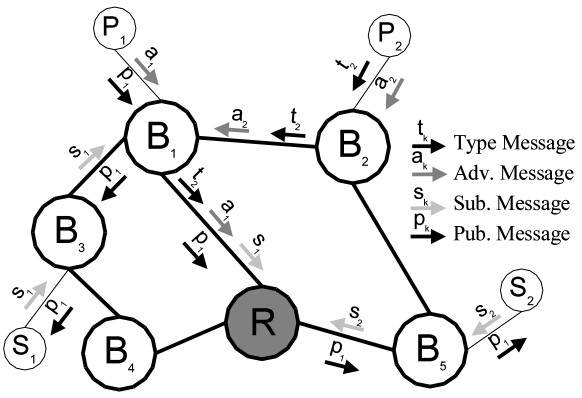

Figure 3. The Type-Based Routing Algorithm

event dissemination tree for publications is set up. Note that other brokers which are not part of the tree are unaffected by any of the messages.

Similar to the Siena's *coverage relation* [8], an event broker only passes on a message if an equivalent or more general message has not already been sent on. For example, the broker B_1 will only forward the first advertisement message a_1 and ignore the second one (a_2) because it does not convey any new information.

4.2. Type- and Attribute-Based Routing

The type- and attribute-based pub/sub layer extends the type-based layer by distributing filter expressions through the network of brokers. Subscribers can now issue subscriptions that filter events depending on the event data (i.e. the attributes). Figure 4 illustrates how filter expressions are distributed through the network and create state in event brokers. A local event matching algorithm [18] can be applied to efficiently match events against filter expressions held at a single broker.

As in the type-based case, a rendezvous node R is first created by a type message. After that, the two publishers, P_1 and P_2, announce their presence by sending advertisement messages. When subscriber S_1 decides to subscribe, it routes a subscription message s_1 to the rendezvous node R. Whenever the subscription reaches a node that holds state about an advertisement for the same event type, the subscription message follows the reverse path of this advertisement. Every broker that forwards a subscription message stores the filter expression in addition to the state kept for type-based pub/sub. Event publications (e.g. p_1) then follow the reverse path taken by subscriptions. Since filtering state exists in the brokers along this path, events are filtered as close to the source as possible. Again, subscriptions only need to be forwarded by a broker if the new filter expression is more general than any of the previous subscriptions along the same path. Therefore, the broker B_1 does not forward subscription s_2 assuming it is already covered by s_1. A broker might decide to merge filters [20] in order to reduce the state kept in the system.

Moreover, the type- and attribute-based pub/sub layer manages *hierarchies of event types*. When a new event type is added to the system, a *parent event type* can be specified. The rendezvous node for the parent type is informed and keeps a reference to all its descendent event types. For supertype subscriptions, a rendezvous node sends every subscription message to the rendezvous nodes of all its descendent types. Figure 5 gives a simple example of a supertype subscription with a hierarchy of three event types. The subscription for the event type e_p will result in notifications of any of the types $\{e_p, e_{s1}, e_{s2}\}$.

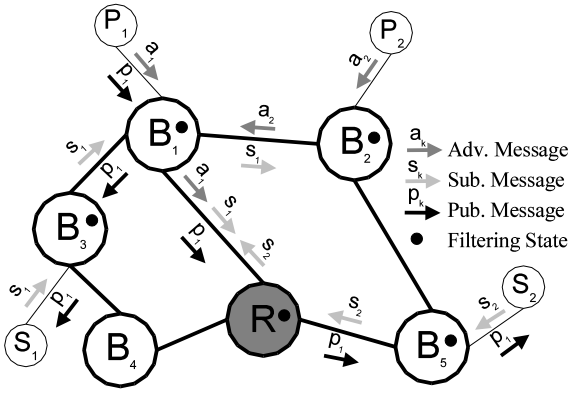

Figure 4. The Type- and Attribute-Based Routing Algorithm with Filtering

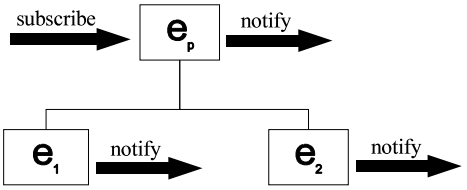

Figure 5. Supertype Subscription

4.3. Fault-Tolerance

Throughout the entire design, fault-tolerance plays an important role in a large-scale middleware system as link or node failures are frequent in wide-area networks with many nodes. Hermes takes advantage of the fault-tolerance and repair mechanisms provided by the overlay routing layer. This enables it to survive multiple link and broker failures and to adapt its routing state so that it can still deliver events to subscribers. The entire state in the event brokers is *soft state* that has to be refreshed periodically. A *heartbeat protocol* ensures that the neighbours of an event broker are reachable and alive. In general, a Hermes middleware has to handle four different kinds of failure, for which different techniques are available.

Link Failure: A link failure is a short- or long-term inability of a node to contact one of its logical neighbours. This could be caused by a failure at the IP routing level. The overlay routing algorithm will then adapt the logical topology of the broker network and circumvent the failed link [22, 26]. This should only affect the event dissemination tree to a limited degree.

Event Broker Failure: A regular heartbeat message is used to detect failed event brokers in the absence of other communication. A failed broker can cause a gap in the event dissemination tree. To heal the tree, the event broker that detected the failure re-routes the subscriptions and advertisements that previously went via the failed broker to the rendezvous node. After the overlay network has adapted, a new path via a different broker is set up while the old path will expire after some time because of soft state.

Event Client Failure: The failure of an event publisher or event subscriber is handled entirely by the soft state approach in Hermes. After an event client has failed, it will stop refreshing its advertisements or subscriptions. Any routing state in the network will expire after some time.

Rendezvous Node Failure: To prevent rendezvous nodes from being single points of failure, they are replicated. If an event client does not receive an acknowledgement from a rendezvous node after sending a subscription or advertisement message, it will try contacting another replica. The node id of a replica rendezvous node is obtained by concatenating a salt value to the event type name before calculating the hash function [26]. For load-balancing or latency reasons, an event client can use any of the replicated rendezvous nodes as its primary contact point. This is transparently handled by the fault-tolerance module in the middleware layer.

A challenge is to keep consistency between replicated rendezvous nodes and to ensure that event dissemination trees will cover all replicas. The problem is similar to multicast with multiple cores as described in [25]. A subscriber should be able to contact any of the replicated rendezvous nodes and still be able to receive events coming from event publishers attached to a different replica. Four different techniques can be identified to achieve this, and we are currently working on simulating their respective benefits.

Clients subscribe to all RNs: When a client issues a subscription, it has to separately send this subscription to all replicated rendezvous nodes. For a global subscription, all rendezvous nodes must be contacted. An event publisher may then advertise to any rendezvous node. This makes subscriptions more expensive in terms of message counts.

Clients advertise to all RNs: An event publisher advertises its event types to all replicas, and subscribers may subscribe to any replica. This creates an overhead for every event publisher in the system.

RNs exchange subscriptions: Event clients only subscribe to a single replica, but this replica then subscribes to all other ones. In case of network partitions between replicas, the subscriber will potentially not receive all events, but it will be able to subscribe as long as at least one replica is reachable. Replicas need to support a reconciliation protocol when they re-join after a network partition.

RNs exchange advertisements: In this scheme, the replicas exchange advertisements. This is more scalable than the previous scheme when the number of subscriptions exceeds the number of advertisements in the system.

4.4. Implementation

We have developed a Java-based implementation of Hermes, consisting of classes for event brokers, event sources, and event sinks. The communication between the components takes place by passing XML-defined messages. We have decided to adopt the XML Schema specification [24] for defining message formats and event types. XML Schema is an expressive language with a rich type system and user-defined data types. The event type model provided can then be mapped into any other programming language like Java or C++. We support a mapping between XML Schema and Java that hides the complexity of XML messages from the event client. Events appear to be Java objects and are transparently translated into XML messages using Java's structural reflection. The current implementation supports the type- and attribute-based publish/subscribe approach with filtering.

5. Related Work

In this section, we compare our work to a number of ongoing efforts in the area of publish/subscribe and event dissemination systems. Hermes can be seen as a result of the lessons learnt from the **Cambridge Event Architecture (CEA)** [2, 19]. The CEA provides event sources, event sinks, and event mediators to decouple sources from sinks. Direct source-sink notification is provided over a standard middleware like CORBA. We have experimented with expressing strongly-typed events in a language-independent way by defining them in the Object Definition Language (ODL) [10, 1]. A persistent event service that stores ODL-defined events in an object-oriented database and a composite event detector have been implemented. However, event mediators cannot be linked in an arbitrary topology, essentially limiting the scalability of the architecture.

Siena [8, 6] is a distributed content-based pub/sub system consisting of a network of event brokers. It focuses on the trade-off between scalability and expressiveness. Hermes uses the same distributed filtering algorithm to propagate events on the reverse paths of previous subscriptions. However, Siena relies on a global broadcast operation to disseminate advertisements through the entire network, which limits its scalability. It does not support the notion of an event type, and does not provide any other middleware services. The network of brokers is not able to cope with failures because of the static logical topology.

A Java-based implementation of an event service is **JEDI** [12, 11]. It consists of active objects, which behave like event sources and sinks, and event dispatchers, which are similar to event brokers. Although its routing algorithm is comparable to our approach, as event dissemination trees are created dynamically after electing a group leader, the group leader must perform a global broadcast to all other event dispatchers to announce its presence. Every event dispatcher must have knowledge of all group leaders, which is not scalable. JEDI supports disconnect and re-connect operations that allow mobile event clients to migrate from one event dispatcher to another. Event types, fault-tolerance and further middleware services are not supported, and the system is tied to a single programming language.

In [5], an architecture for a global event-based notification service called **Herald** is proposed. It is a general framework that only provides a topic-based pub/sub service on the Internet using publishers, subscribers and rendezvous points. Any middleware functionality such as content-based filtering, event types, naming, composite events, etc. is supposed to be provided by application-level services. Fault-tolerance based on replicating rendezvous points is suggested, but no algorithm for keeping consistent state between replicas is described.

Two topic-based event dissemination systems built on top of an overlay routing network are **Bayeux** [27] and **Scribe** [22]. Both use rendezvous nodes that are created by routing a message to the topic identifier. No content-based filtering of topic content is supported, but rendezvous nodes can be replicated for fault-tolerance. Another limitation lies in the fact that all the event publications must be sent via the rendezvous node which can become a bottleneck. Hermes extends these systems and provides a type- and attribute-based pub/sub service.

The term **type-based publish/subscribe** was first introduced in [17, 15]. That work attempts to give an event type model that cleanly integrates with the type model of an object-oriented programming language. Events are treated as first-class (Java) objects, and subscribers specify the class of objects they are willing to receive. No attribute-based filtering is supported, as this would break encapsulation principles. Instead, arbitrary methods can be called on the event object to provide a filtering condition. Although such an approach unites publish/subscribe with object-orientation, an efficient implementation would be difficult since filter expressions can be arbitrarily complex and thus hard to optimise or distribute. We feel that a large-scale distributed system benefits from expressing subscriptions based on attribute values, as this allows information interest to be expressed at a finer granularity.

6. Conclusion

Large-scale distributed systems have different requirements from systems developed in a single local-area network. The complexity of designing and building these systems must be reduced by using an appropriate middleware. It turns out that loosely-coupled event-based communication has strong advantages compared to a tightly-coupled

invocation-based paradigm. In this paper, we have presented the design of Hermes, a distributed event-based middleware. We feel that Hermes addresses many of the issues by focusing on clean programming language integration without sacrificing scalability or efficiency. Type- and attribute-based publish/subscribe gives an intuitive model for using an event-based middleware by first specifying the type and then filtering within the event attributes. The proposed fault-tolerance mechanisms can be transparent to distributed application programmers, but make the system robust on a global scale on which link or node failures are unavoidable. The abstraction of an overlay routing layer helps to hide some of the routing complexity and enables Hermes to scale to a large number of event clients and brokers.

To obtain performance and scalability figures, we are currently working on a full implementation of Hermes within a simulator. In addition, we are investigating ways of dynamically changing the overlay network topology in response to the distribution of subscriptions, advertisements, and events. Future work will include the provision of more middleware services like composite event detection, persistent events, access control, transactions, and support for mobile event clients with partial connectivity.

References

[1] J. Bacon, A. Hombrecher, C. Ma, K. Moody, and W. Yao. Event Storage and Federation using ODMG. In *Proc. of the 9th Int. Workshop on Persistent Object Systems (POS9)*, pages 265–281, Lillehammer, Norway, Sept. 2000.

[2] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic Support for Distributed Applications. *IEEE Computer*, pages 68–77, Mar. 2000.

[3] T. Ballardie, P. Francis, and J. Crowcroft. Core Based Trees (CBT). In *ACM SIGCOMM '93*, Ithaca, N.Y., USA, 1993.

[4] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *ICDCS*, pages 262–272, 1999.

[5] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Proc. of the 8th Workshop on Hot Topics in OS (HotOS-VIII)*, 2001.

[6] A. Carzaniga, J. Deng, and A. L. Wolf. Fast Forwarding for Content-Based Networking. Technical report, Dept. of Computer Science, Univ. of Colorado, Nov. 2001.

[7] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Challenges for Distributed Event Services: Scalability vs. Expressiveness. In *Engineering Distributed Objects '99*, Los Angeles, CA, USA, May 1999.

[8] 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.

[9] A. Carzaniga and A. L. Wolf. Content-based Networking: A New Communication Infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, Scottsdale, USA, Oct. 2001.

[10] R. G. G. Cattell et al. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[11] G. Cugola and E. D. Nitto. Using a Publish/Subscribe Middleware to Support Mobile Computing. In *Middleware for Mobile Computing Workshop*, Heidelberg, Germany, 2001.

[12] 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.

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

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

[15] P. T. Eugster and R. Guerraoui. Content-Based Publish/Subscribe with Strucutural Reflection. In *Proc. of the 6th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS01)*, Jan. 2001.

[16] P. T. Eugster, R. Guerraoui, and C. H. Damm. On Objects and Events. In *Proc. for OOPSLA 2001*, Tampa Bay, USA, Oct. 2001.

[17] P. T. Eugster, R. Guerraoui, and J. Sventek. Type-Based Publish/Subscribe. Technical report, EPFL, Lausanne, Switzerland, June 2000.

[18] F. Fabret, A. Jacobsen, F. Llirbat, et al. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *ACM SIGMOD 2001*, pages 115–126, May 2001.

[19] C. Ma and J. Bacon. COBEA: A CORBA-Based Event Architecture. In *Proc. of the 4th USENIX Conf. on O-O Tech. and Systems*, pages 117–131, Santa Fe, USA, Apr. 1998.

[20] G. Mühl and L. Fiege. Supporting Covering and Merging in Content-Based Publish/Subscribe Systems: Beyond Name/Value Pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001.

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

[22] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The Design of a Large-Scale Event Notification Infrastrucutre. In *Proc. of the 3rd Int. Workshop on Networked Group Communication (NGC2001)*, Nov. 2001.

[23] B. Segall and D. Arnold. Elvin has left the Building: A Publish/Subscribe Notification Service with Quenching. In *Proc. of AUUG Technical Conference '97*, Brisbane, Australia, Sept. 1997.

[24] The World Wide Web Consortium. XML Schema Part 1 + 2: Structures and Datatypes. W3C Rec., W3C, May 2001.

[25] D. Zappala and A. Fabbri. An Evaluation of Shared Multicast Trees with Multiple Active Cores. In P. Lorenz, editor, *LNCS 2093*, pages 620+, July 2001.

[26] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical report, Computer Science Division, Univ. of California, Berkeley, USA, Apr. 2001.

[27] S. Q. Zhuang, B. Y. Zhao, A. Joseph, et al. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *Proc. of the 11th Int. Workshop on Network and OS Support for Digital Audio and Video (NOSSDAV01)*, June 2001.