ABSTRACT

Implicit invocations are a popular mechanism for exchanging information between software components without binding these strongly. This decoupling is particularly important in distributed systems when interacting components are not known until runtime. In most realistic distributed systems though, components require some information about each other, be it only about their presence or their number. Runtime systems for implicit invocations—so-called publish/subscribe systems—are thus often combined with other systems providing such information.

Given the variety of requirements for information about interacting components across applications, this paper proposes a generic augmentation of implicit invocations: rather than extending a given publish/subscribe API and system in order to convey a particular type of information across interacting components, we describe domain-specific joinpoints that can be used to advise application-level invocation routers—so-called brokers—used by publish/subscribe systems. This enables aggregation of application-specific information to and from components in a scalable manner.

After presenting our domain-specific joinpoint model, we describe its implementation inside the REDS publish/subscribe middleware. The empirical evaluation of our approach shows that: (a) it outperforms external aggregation systems, by collecting and distributing information with a limited overhead; (b) the deployment of new functionalities has virtually no overhead, even if it occurs while the publish/subscribe system is running.

1. INTRODUCTION

Many distributed software systems are designed according to the paradigm of implicit invocations [39, 43]: software components announce events which are conveyed to target components that registered interest in such events, instead of invoking the target components directly. By shifting communication away from direct invocations via component references to asynchronous consumption of events of interest produced anonymously by components, application components can be added (and removed) at runtime. This increases flexibility through decoupling—event sources need not manage their targets explicitly—and supports scalability by allowing for asynchronous, streamlined communication.

1.1 Publish/Subscribe Systems

Publish/subscribe systems [35] provide the communication backbone for software systems based on implicit invocations. They transmit events produced, potentially at high rates, by many publishing components to many subscribed components. For years, research on publish/subscribe systems has focused on scalable and flexible architectures that support fine-grained content-based addressing models, in which published events are routed as messages based on the subscribers’ interests in the content of messages. Such architectures commonly use so-called brokers to form decentralized overlay networks of application-level routers that interconnect publishers and subscribers. Brokers perform efficient en-route filtering and selective forwarding of messages based on their content. While extremely scalable and flexible, these systems have only seen limited adoption in practice. In particular, they have not dethroned simpler systems based on centralized brokers and ones that route
messages directly based on namespaces, i.e., by associating messages with single topic (or subject) names.

One reason for the poor adoption of content-based systems, particularly decentralized publish/subscribe systems, is that simpler systems assigning topics to individual single brokers can more easily provide additional information on the usage of these topics. Many applications that utilize implicit invocations for scalable communication, however, require such information, such as the number of publishers and subscribers connected to a given topic or the the set of subscribers that might have received published events. Concrete scenarios are presented in the remainder of this section.

Currently, these applications must use less scalable, centralized, publish/subscribe systems or group communication systems that can provide membership information albeit at high overhead. This reduces the performance benefits of implicit invocations. Moreover, these systems do not provide broader feedback information efficiently, such as the relevance of a message based on the number of potentially interested subscribers, unless using out-of-band communication which can hamper scalability.

We present two scenarios that motivate the need for additional information exchange in a publish/subscribe system, without sacrificing scalability, when implementing implicit invocations. In the first scenario—systems monitoring—subscribers seek additional information about publishers, while in the second scenario—online advertising—publishers receive information about subscriptions.

### 1.2 Example Scenarios

The monitoring of networked systems is a common application of publish/subscribe-style communication. Publishers may be servers in a data center that are monitored for load spikes and failures. The subscribers may be monitoring systems that determine the “health” of the data center, and assist with root-cause analysis of failures. The overall rate of monitoring data from servers can become high [30]: for example, probes may provide measurement data about utilization of server resources with sample rates of many times a second. When a client subscribes to all events that may be of interest, it can become easily overwhelmed. Similarly, by subscribing only to very specific events, a client may miss relevant ones, which are causal antecedents of observed symptoms. This prevents root cause discovery, thus making the diagnosis of failures difficult. Fig. 1 sketches the implementation of a monitor client based on the Java Message Service (JMS) [36] API. The API provides no way before or after the publish call to know the number of potential or actual receivers for the event. There may be no corresponding subscriptions at all, in which case the client might want to use another communication band or raise an alarm.

Thus it would be desirable to aggregate performance characteristics, for example, by summarizing key metrics as statistical distributions. If aggregation happens in the broker network, clients can receive information about the general health of a large number of data center components without being overloaded by low-level events. As specific problems in the data center develop, subscriptions can be refined to focus on more detailed events for root cause analysis.

In online advertising on the web [20], advertising agents act as publishers of advertising messages (ads), and users are subscribers to particular ads. Subscriptions reflect user interests, potentially inferred from browsing activities. For targeted online advertising to be most successful, however, advertisers want to have information about potential consumers who subscribe to their advertisements. This goes beyond a basic publish/subscribe model, in which publishers and subscribers remain anonymous—in that case, subscribers cannot gauge the number of potentially matching publishers and vice-versa.

An aggregation mechanism provided within the broker network would satisfy the requirements of both parties in this scenario. An aggregation function can ensure that the feedback sent to the publishers only includes statistical distributions of subscribers’ interests, and thus would preserve anonymity. Subscribers can safely use fine-grained subscriptions, and thus ensure that they receive only ads that might be of interest to them.

### 1.3 Application-Specific Integrated Aggregation

In order to support the above scenarios, we propose to augment publish/subscribe systems with application-specific integrated aggregation (ASIA), which offers the ability to convey additional information, besides published messages, as required by communicating components in an application.

Compared to the use of an external aggregation system, the integration of aggregation with the publish/subscribe system has benefits in terms of efficiency (e.g., combining network communication for aggregation with actual messages) and expressiveness (giving access to broker internals).

However, it raises an immediate question: how to express aggregation logic in a generic manner that is suitable for decentralized evaluation? In particular, the precise nature of aggregation remains application-specific. In the above monitoring scenario, for example, monitoring clients should have the flexibility to provide their own aggregation functions, potentially at runtime.

The core idea behind ASIA is inspired by aspect-oriented programming (AOP) [29]: we define a set of joinpoints that are specific to the filtering and routing of messages in decentralized publish/subscribe systems. These joinpoints can be advised in order to perform aggregation of relevant information by altering and augmenting the default broker code. Our joinpoints can also be used to support more general behavior change in brokers, such as protocol extensions.

### 1.4 Contributions

```java
public class JMSMonitoringClient {
    ...
    TopicSession t = ...;
    TopicPublisher tp = t.createPublisher(...);
    ObjectMessage om = t.createObjectMessage();
    Long loadLevel = ...;
    ...
    om.setObject(loadLevel);
    tp.publish(om);
    ...
}
```

Figure 1: Example of system monitoring client based on JMS [36] pushing load level readings regardless of subscribers. The client has no indication of the number of potential or actual event receivers.
This paper reports on our experience with the design and implementation of a “reflective” publish/subscribe system according to the ASIA model. More specifically, we:

- present domain-specific joinpoints for decentralized implicit invocation (publish/subscribe) messaging systems;
- describe an implementation of this model within the Java-based REDS publish/subscribe system [10];
- demonstrate the benefits of the ASIA model by showing the performance improvements of integrated aggregation over the use of a separate aggregation system; and
- evaluate a mechanism for deploying new aggregation functions to be attached to joinpoints in brokers. We also discuss the supported flexibility in terms of the different forms of aggregation that are implementable in ASIA without changes to the main broker code.

2. BACKGROUND

After outlining the generic publish/subscribe model that we assume in this paper, we derive the requirements of a mechanism for additional information flow.

2.1 Publish/Subscribe Model

Several publish/subscribe models have been proposed in the literature [15] (cf. [8]). We adopt a mixed topic/content-based publish/subscribe model, which provides a good compromise between performance and expressiveness in practice. It enables ASIA to be applied both to pure topic-based and pure content-based publish/subscribe systems.

Subscribers issue subscriptions to express interest in a topic, and may additionally use a predicate filter to further narrow their subscriptions to specific content. Publishers send advertisements to indicate that they will publish messages to a specific topic. Publication messages are then routed from publishers to interested subscribers.

We consider clients and brokers to be interleaved to form an overlay network, as shown in Figure 2(a). For ease of presentation, we assume the overlay network to be free of cycles. The brokers (b1 to b5) provide the publish/subscribe service to clients. Brokers directly connected to clients (b{1,3,5}) are termed edge brokers. We further assume each client to be connected to one and only one broker. Each broker state includes information (i.e., advertisements and subscriptions) on directly connected clients. Furthermore, the broker state stores information on neighboring broker nodes and their respective (transitive) advertisements and subscriptions.

Broker nodes propagate events—each of which pertains to a topic τ—from publishing nodes (i.e., publishers of τ events) to client nodes that have expressed interest in receiving τ events (i.e., subscribers to τ with content-based filters). A broker b{1} propagates events for each topic τ along its neighbor links to other brokers or clients via lower-level protocol messages (event messages); analogously a broker propagates advertisements (advertisement messages) and subscriptions (subscription messages).

To illustrate the routing of messages in the broker network, Figure 2(a) shows a sample sequence, in which client c1 first advertises publications for a topic (step 1); clients c3 and c4 subscribe to the advertised topic (steps 2 and 3); and finally client c1 publishes an event of that topic (step 4).

![Distributed publish/subscribe system with messages](image)

Figure 2: Distributed publish/subscribe system with messages

Algorithm 1 A basic publish/subscribe client algorithm. The client is instantiated with an edge broker

```
1: init
2: b
3: for all published topics τ do
4: SEND(pub, τ) to b
5: to PUBLISH(e) on topic τ do
6: SEND(pub, τ, e) to b
7: to SUBSCRIBE(φ) to topic τ do
8: SEND(sub, τ, φ) to b
9: to UNSUBSCRIBE(φ) from topic τ do
10: SEND(unsub, τ, φ) to b
11: upon RECV(pub, τ, e) do
12: DELIVER(e)
```

2.2 Broker Routing Algorithms

The routing of messages in a broker network is conducted according to routing algorithms. We now briefly introduce how these algorithms work before presenting our corresponding integrated aggregation mechanisms in the following sections.

Algorithm 1 shows the API utilized by clients and the corresponding broker routing algorithm is shown in Algorithm 2. The to ... do clauses describe how to achieve a desired function (e.g., publish a message) using lower layers in the network stack. In contrast, the upon ... do clauses indicate the operations that occur when lower layers in the network stack react to messages they have received. In both algorithms, we assume that nodes communicate by pair-wise reliable channels offering primitives SEND and RECV. For simplicity, a node p acts either as a client c or as broker b, and advertisements are only on topics and do not include value ranges. Unsubscriptions are elided for brevity.

In our algorithm listings operations such as insertion of a predicate filter (INSERT()), removal (DELETE()) or matching (MATCH()) on partially ordered sets—which store subscriptions using subsumption [4]—are just shown as procedure calls. Subscribers (subs), subscriptions (partially ordered set P), and publishers (pubs) are stored by topic τ; links is the array of neighbor brokers.

We assume that brokers act as publishers and subscribers towards their respective downstream and upstream neigh-
Algorithm 2: Algorithm for publish/subscribe event processing with subscription summarization. $\mathcal{P}[\tau]$ is the predicate poset ordered by $\preceq$. $\text{pubs}[\tau]$ stores the advertising peers. $\text{subs}[\tau]$, stores peers that subscribe with $\phi$. $\text{subs}[\tau][\phi]$ avoids the need to duplicate $\phi$ in $\mathcal{P}[\tau]$, if more than one peer subscribes with $\phi$.

Publish/subscribe broker algorithm. Executed by broker $b_i$:

1: init
2: $\text{pubs}[] \{\text{Process sets indexed by topic } \tau\}$
3: $\mathcal{P}[] \{\text{Posets indexed by topic } \tau\}$
4: $\text{subs}[] \{\text{Process sets indexed by } \tau \land \phi\}$
5: links $\{\text{Neighbor brokers } p_i\}$
6: upon $\text{recv}(\text{ad}, \tau)$ from $p_i$ do $\{\text{Receiving advertisement}\}$
7: if $p_j \notin \text{pubs}[\tau]$ then
8: for all $b_k \in \text{links}(\{p_i\})$ do
9: $\text{send}(\text{ad}, \tau)$ to all $b_k$
10: $\text{send}(\text{sub}, \tau, \text{LUB}(\mathcal{P}[\tau]))$ to $p_j$
11: $\text{pubs}[\tau] \leftarrow \text{pubs}[\tau] \cup \{p_j\}$
12: upon $\text{recv}(\text{sub}, \tau, \phi)$ from $p_j$ do $\{\text{Receiving subscription}\}$
13: $\text{subs}[\tau][\phi] \leftarrow \text{subs}[\tau][\phi] \cup \{p_j\}$
14: $\phi^{\text{old}} \leftarrow \text{LUB}(\mathcal{P}[\tau])$
15: if $\text{subs}[\tau][\phi] = 1$ then
16: insert$(\mathcal{P}[\tau], \phi)$
17: $\text{upd}(\tau, \phi^{\text{new}}, \text{LUB}(\mathcal{P}[\tau]), p_j)$
18: upon $\text{recv}(\text{pub}, \tau, e)$ from $p_j$ do $\{\text{Receiving event}\}$
19: $\text{matches} \leftarrow \text{match}(\mathcal{P}[\tau], e)$
20: for all $\phi \in \text{matches}$ do
21: for all $p_k \in \text{subs}[\tau][\phi] \{p_j\}$ do
22: $\text{send}(\text{upd}(\tau, \phi, e))$ to $p_k$
23: upon $\text{recv}(\text{unsu}, \tau, \phi)$ from $p_j$ do $\{\text{Receiving unsubscription}\}$
24: $\text{subs}[\tau][\phi] \leftarrow \text{subs}[\tau][\phi] \setminus \{p_j\}$
25: $\phi^{\text{old}} \leftarrow \text{LUB}(\mathcal{P}[\tau])$
26: if $\text{subs}[\tau][\phi] = 0$ then
27: delete$(\mathcal{P}[\tau], \phi)$
28: $\text{upd}(\tau, \phi^{\text{old}}, \text{LUB}(\mathcal{P}[\tau]), p_j)$
29: procedure $\text{upd}(\tau, \phi^{\text{old}}, \phi^{\text{new}}, p_j)$ $\{\text{Update poset and neighbors}\}$
30: if $\phi^{\text{old}} \neq \phi^{\text{new}}$ then
31: for all $b_k \in \text{pubs}[\tau]\{p_j\}$ do
32: $\text{send}(\text{sub}, \tau, \phi^{\text{new}})$ to $b_k$
33: $\text{send}(\text{unsu}, \tau, \phi^{\text{old}})$ to $b_k$

3. AN ASPECT-ORIENTED PUBLISH/SUBSCRIBE BROKER

In this section we present an aspect-oriented event broker algorithm for publish/subscribe systems that supports reflection by exposing joinpoints.

3.1 Overview

Algorithm 3 outlines a reflective broker algorithm exposing joinpoints as an evolution of that of Algorithm 2. The joinpoints refer to points in the execution, at which we may want to customize the algorithm’s behavior. Execution “jumps” from a given joinpoint $X-Y$ to a corresponding advice $\text{advise} X-Y$, which is said to advise joinpoint $X-Y$.

In ASIA, this advice is used to invoke code that effects the computation of an aggregation function within a given broker—a simple example would be a function that maintains a count of subscribers to a particular topic—down a subtree reachable from this broker. We discuss the aggregation computations in more detail in §3.3 shortly. Every joinpoint involves a specific set of arguments passed to its advice, which represent volatile computational state of the broker for the current action being performed.

3.2 Joinpoints

Table I summarizes the joinpoints used in ASIA and their arguments. Besides arguments, code for advice $\text{advise} X-Y$ is constrained by a type of value that must be returned. Corresponding advice are thus able to substitute some of the volatile current computational broker state. For every joinpoint/advice, there is a default behavior, which corresponds to the logic of the basic, non-reflective, broker algorithm.

In a custom advice $\text{advise} X-Y$, this code can be optionally invoked explicitly via $\text{procdo} X-Y$. The primitive $\text{procdo} X-Y$ has the same formal argument and return types as the corresponding advice $\text{advise} X-Y$. In ASIA, if no advice is specified for a given joinpoint $X-Y$, a call to $\text{procdo} X-Y$ is automatically performed ($\text{advise} X-Y(\text{args}) = \text{procdo} X-Y(\text{args})$). Thus in the absence of custom advice, Algorithm 3 has the same semantics as Algorithm 2.

In addition to joinpoints, ASIA uses subtype polymorphism (subtyping) for easy expression of code for aggregation. Attributes of broker protocol messages can be augmented by appending to them (denoted $\oplus v$ to append a value $v$) for piggybacking information. This operation refers to an implicit subtype of the original attribute type, augmenting its super-type with an attribute of the type of $v$. We also allow aggregation features to add their own specific reactions (upon clauses) or even tasks, as well as custom message types issued and consumed via $\text{send}$ and $\text{recv}$, respectively. Due to space limitations, we elide the presentation of aggregated information to clients.

3.3 ASIA Model

While joinpoints and advice capture which code to inject into brokers for the sake of aggregation and where, they do not exactly say what such code should perform. This section thus provides an overview of the ASIA aggregation model, before presenting a concrete example in §3.4.

An aggregation function $f$ is evaluated by aggregators that operate at each broker in the distributed publish/subscribe system. Aggregators are pieces of code that are installed using advice. They can access the state at each broker, for example to include current aggregation computations, or to store their own state. By adding some restrictions on the type of aggregation function that can be computed, we can ensure that the results of the aggregation computations at each broker can be combined. Thus we can compute a global aggregation result by repeatedly combining the partial results generated within each broker.

In this paper we focus on aggregators using additive func-
In a distributed broker network, a client that registers interest in an aggregation function at a broker \( b_i \) causes the formation of a spanning tree rooted at \( b_i \). The tree, which we call the aggregation tree, contains all of the other brokers \( b_j \) that can contribute relevant aggregation results. To maintain aggregation of data, brokers exchange messages along the aggregation tree. For efficiency, these messages may be piggybacked onto existing publish/subscribe messages rather than being sent separately. Brokers that provide a broker \( b_i \) with data to aggregate are considered below \( b_i \). A broker that receives more aggregated data is considered to be above \( b_i \), and is closer to the root of the tree.

Consider our online advertising scenario from before. Figure 2(b) shows the aggregation tree formed when \( c_1 \) requests the count of subscribers from \( b_1 \). From the perspective of \( b_1 \), the subscriber count is 2—i.e., \( b_i \)'s descendants \( c_3 \) and \( c_4 \). But \( b_4 \) can determine this value by learning the counts from its immediate children, \( b_3 \) and \( b_2 \), namely one each, and then applying \( f \) to these counts. The contents of the messages indicating updates to the aggregation value are shown on Figure 2(b) as sets propagating up the aggregation tree.

As a consequence, computing an overall \( f \) result can be broken down into computations of \( f \) at individual brokers \( b_i \) within the distributed publish/subscribe system. To enable this distributed computation in our model, every broker \( b_i \) stores the most recent evaluation of \( f \) so as to track the \( f \) that will be passed to its parent in the aggregation tree: we denote this cached value \( v_i \). In addition, every broker stores the value of \( v_k \) for its direct children in the aggregation tree. All of these \( v \) values are part of the broker state.

When \( v_i \) changes, a message is sent by broker \( b_i \) to the broker above it in the tree, triggering an update of its \( v \) value. So in Figure 2(b), if client \( c_3 \) were to unsubscribe from \( \tau \), then \( b_3 \) would recompute \( v_3 \), and would need to inform \( b_1 \) in order for \( v_4 \) to be recomputed. This would continue up the tree to the root broker \( b_1 \), which would invoke client \( c_1 \)'s registered aggregation callback. \( c_1 \) would thus learn the number of subscribers as requested—and in particular that this number had recently changed.

### 3.3.2 Precision of Distributed Aggregation

In this above approach, every change in the aggregation value \( v_i \) at any broker causes aggregation value update messages to be sent up to the root of the aggregation tree, which limits scalability. Therefore, similar to other scalable aggregation models [29], we relax the precision of \( v_i \) that is propagated upwards in the aggregation tree by having each broker \( b_i \) maintain \( v_i \) privately, which we term \( \bar{v}_i \). Updates to \( \bar{v}_i \) are sent upwards in the tree only if the previous values of \( \bar{v}_i \) sent up the tree would otherwise be more than some bound away from \( b_i \)'s most recent \( v_i \). The bound on the precision with which \( \bar{v}_i \) approximates the true evaluation of \( f \) (i.e., \( v_i \)) is denoted by \( \delta \in \mathbb{R} \). If \( R \) is the range of an aggregation function \( f \), we require \( d : R \times R \to R \) to be a distance metric on \( R \). For our example scenario, \( d \) is simply the positive integer difference between any two numbers of subscribers.

Clients control the quality (i.e., precision) of the aggregation data, by setting the \( \delta \) parameter when they register interest in an aggregation function. When a broker \( b_i \) first constructs an aggregation tree, it divides the \( \bar{v}_i \) that it has been given between itself and the subtrees, at which its immediate children are roots. For a broker with \( v \) children, a simple heuristic is to set \( \bar{v}_i = f^{-1}(v_i) \). For aggregators that perform summation, this means setting \( \bar{v}_k = \frac{\sum_{j} v_j}{v+1} \) for each child \( k \), and using the same bound over its directly connected clients.

To illustrate this approach using our running example, consider \( \bar{v}_1 = 2 \) and broker \( b_2 \) having set \( \bar{v}_2 = 1 \) and \( \bar{v}_3 = 1 \). Because of subscriber \( c_3 \), \( v_3 = 1 \). If \( c_3 \) unsubscribes, now

### Table 1: Joinpoints X−Y

<table>
<thead>
<tr>
<th>Joinpoint description (X−Y)</th>
<th>Arguments</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advertiser addition (AD-ADD)</td>
<td>( \tau, ) new advertiser ( p_j )</td>
<td>-</td>
</tr>
<tr>
<td>Subscriber addition (SUB-ADD)</td>
<td>( \tau, ) new subscription ( \phi ), new subscriber ( p_j )</td>
<td>old LUB ( \phi_{old} )</td>
</tr>
<tr>
<td>Advertiser removal (AD-DEL)</td>
<td>( \tau, ) old advertiser ( p_j )</td>
<td>-</td>
</tr>
<tr>
<td>Subscriber removal (SUB-DEL)</td>
<td>( \tau, ) old subscription ( \phi ), old subscriber ( p_j )</td>
<td>old LUB ( \phi_{old} )</td>
</tr>
<tr>
<td>Publication matching (PUB-MATCH)</td>
<td>( \tau, ) publication ( e )</td>
<td>matching subsr.</td>
</tr>
<tr>
<td>Publication sending (PUB-SEND)</td>
<td>( \tau, ) publication ( e ), upstream publisher ( p_j )</td>
<td>-</td>
</tr>
<tr>
<td>Subscription sending (SUB-SEND)</td>
<td>( \tau, ) new own subscr. ( \phi ), new subscriber ( p_j )</td>
<td>-</td>
</tr>
<tr>
<td>Unsubscr. sending (UNSUB-SEND)</td>
<td>( \tau, ) old own subscr. ( \phi ), old subscriber ( p_j )</td>
<td>-</td>
</tr>
<tr>
<td>Advertisement sending (AD-SEND)</td>
<td>( \tau, ) new own advertisement ( p_j )</td>
<td>-</td>
</tr>
<tr>
<td>Unadvert. sending (UNAD-SEND)</td>
<td>( \tau, ) new own advertisement ( p_j )</td>
<td>-</td>
</tr>
</tbody>
</table>
Algorithm 3 A reflective broker algorithm. Advice ADVS\(X-Y\)-abstract core broker functionalities. When not implemented by an aggregation feature, ADVS\(X-Y\)(args) is implemented by a direct call to PROC\(D-Y\)(args). When installing a custom ADVS\(X-Y\) the corresponding default behavior in PROC\(D-Y\) can be invoked explicitly.

Reflective publish/subscriber broker algorithm. Executed by broker \(b\).

1: init
2: pubs[] \{Process sets indexed by topic \(\tau\)\}
3: \(\Pi[]\) \{Posets indexed by topic \(\tau\)\}
4: sub[][] \{Process sets indexed by \(\tau \& \phi\)\}
5: links \{Neighbor brokers \(p_i\)\}
6: upon recv\((ad, \tau)\) from \(p_i\) do \{Receiving advertisement\}
7: if \(p_i \notin pubs[\tau]\) then
8: \(\text{ADVS}_{\tau}[\tau, p_i]\)
9: \(\text{ADVS}_{\tau}[\tau, p_i]\)
10: \(\text{ADVS}_{\tau}[\tau, p_i]\)
11: upon recv\((pub, \tau, e)\) from \(p_i\) do \{Receiving event\}
12: match \(\leftarrow \text{ADVS}_{\tau}[\tau, e]\)
13: for all \(\phi \in \text{match}\) do
14: \(\text{ADVS}_{\tau}[\tau, \phi]\)
15: upon recv\((subs, \tau, \phi)\) from \(p_i\) do \{Receiving subscription\}
16: \(\phi_{\text{old}} \leftarrow \text{ADVS}_{\tau}[\tau, \phi]
17: udp\((\tau, \phi_{\text{old}}, \text{lub}(\Pi[\tau]), p_i)\)
18: upon recv\((unsubs, \tau, \phi)\) from \(p_i\) do \{Receiving unsubscription\}
19: \(\phi_{\text{old}} \leftarrow \text{ADVS}_{\tau}[\tau, \phi]
20: udp\((\tau, \phi_{\text{old}}, \text{lub}(\Pi[\tau]), p_i)\)
21: procedure udp\((\tau, \phi_{\text{old}}, \phi_{\text{new}}, p_i)\) \{Update poset and neighbors\}
22: if \(\phi_{\text{old}} = \phi_{\text{new}}\) then
23: \(\text{ADVS}_{\tau}[\tau, \phi_{\text{new}}]\)
24: \(\text{ADVS}_{\tau}[\tau, \phi_{\text{new}}]\)
25: procedure proc\(\text{adv}_{\tau}(\tau, p_i)\) \{Default advice for new adv\}
26: pubs[\tau] \(\leftarrow \text{pubs}[\tau] \cup \{p_i\}\)
27: function proc\(\text{adv}_{\tau}(\tau, p_i)\) \{Def. adv. for new subscript.\}
28: \(\text{subs}[\tau][\phi] \leftarrow \text{subs}[\tau][\phi] \cup \{p_i\}\)
29: \(\phi_{\text{old}} \leftarrow \text{lub}(\Pi[\tau])\)
30: if \(\text{subs}[\tau][\phi] = 1\) then
31: insert\((\Pi[\tau], \phi)\)
32: return \(\phi_{\text{old}}\)
33: function proc\(\text{adv}_{\tau}(\tau, p_i)\) \{Def. adv. for unsubscription\}
34: \(\text{subs}[\tau][\phi] \leftarrow \text{subs}[\tau][\phi]\}\{p_i\}\)
35: \(\phi_{\text{old}} \leftarrow \text{lub}(\Pi[\tau])\)
36: if \(\text{subs}[\tau][\phi] = 0\) then
37: delete\((\Pi[\tau], \phi)\)
38: return \(\phi_{\text{old}}\)
39: function proc\(\text{adv}_{\tau}(\tau, e)\) \{Def. adv. for event forwarding\}
40: return match\((\Pi[\tau], \phi)\)
41: procedure proc\(\text{adv}_{\tau}(\tau, e)\) \{Def. advice for new event\}
42: for all \(p_j \in \cup\{\text{sub}[\tau][\phi]\}\{p_i\}\) do
43: send\((pub, \tau, e)\) to \(p_j\)
44: procedure proc\(X_{\tau}(X, \tau, p_j)\) \{Def.\} for \(\text{adv. for \{un\}subscription}\}
45: for all \(p_j \in \text{pubs}[\tau]\) do \{adv. for \{un\}subscription\}
46: send\((X, \tau, \phi)\) to \(p_j\)
47: procedure proc\(X_{\tau}(X, \tau, p_j)\) \{Def. adv.\}
48: for all \(b_i \in \text{links}[\tau]\) do \{for \{un\}advertisement\}
49: send\((X, \tau)\) to \(b_i\)

For our advertising scenario, we require maintenance of a subscriber count—an internal variable \(c\) (line 2), which stores subscription count ranges for subscribers (i.e., \(v_j\) for each broker \(b_j\)).

Subscriptions are augmented (\(\oplus\)) with count ranges when new subscriptions are issued. Upon reception of a subscription, the call to ADVS\(\text{ub}_{\text{ADV}}\) (line 10 in Algorithm 3) triggers the code at line 3 in Algorithm 4. Thereafter, the default behaviour is invoked via PROC\(\text{ub}_{\text{ADV}}\) at line 3. Subtype polymorphism allows us to append an actual count value (representing \(\tau\)) to any such subscription. The appending of this value is illustrated on line 13 in Algorithm 4.

Now, if the LUB has not changed, but the subscriber count—which necessarily changed—has moved outside the count range held by the broker \(p_k\) (as calculated by the implementation of \(f\) at line 7), we send to the appropriate neighboring brokers a message to update the aggregate value (abbreviated here as UAV) at line 9 before returning the result of PROC\(\text{ub}_{\text{ADV}}\). In case the LUB has changed, the advice for subscriptions invoked in upd propagates the new count range along with the new subscription. Advising of subscription message reception proceeds analogously, except...
that unsubscriptions need not be augmented with counts as the information is present at respective brokers.

If a broker receives a message requesting it to update an aggregate value (line 22), it updates the local count ranges and recursively updates the neighbor brokers’ count information, if their current count ranges do not cover the new count information. As mentioned, we must also advise the sending of new subscriptions in the context of UPDs, as subscriptions must be augmented with respective counts (line 13).

3.5 Dynamic Aggregation Function Changes

The capability to change broker behaviour at runtime is powerful, but potentially unsafe, if the change of broker behaviour leads to inconsistent processing of events that are in transit.

While we assume the correctness of the implementation of the new behaviour and do not explore generic mechanisms for ensuring safe modification of broker behaviour, we provide an intuition of how the problem can be tackled for the specific case of distributed evaluation of aggregation functions. As already described, given a function \( f \) evaluated over an aggregation tree \( t \), each broker \( b_i \) receives aggregate updates from its children in \( t \) and forwards updates towards the root of \( t \). By propagating a new aggregation function \( f_{new} \) from the root of an aggregation tree, down to the leaves, we ensure that a generic broker \( b_i \) has obtained \( f_{new} \) before it starts to receive updates for \( f_{new} \) from brokers below it. Indeed, \( b_i \) can receive updates only from its children in the tree, which obtain \( f_{new} \) from (and, thus, after) \( b_i \).

While this ensures safe deployment of new, independent functions, the generic case in which functionalities are substituted or modified is more complex. It potentially requires every broker \( b \) to (i) modify its internal state and adapt it to new functionalities, and (ii) support transition phases in which \( b \) receives updates from its children that are generated by an old version of the communication protocol. We leave this for future work.

4. IMPLEMENTATION

This section describes our prototype implementation of ASIA in the REDS publish/subscribe system [10]. REDS was chosen due to its modularity, which gives us the flexibility needed to implement our joinpoint model.

4.1 Overview of REDS

REDS provides a framework of Java classes and defines the architecture of a generic broker through a set of components with well-defined interfaces. The current release of REDS offers concrete implementations for each component, thus limiting our implementation effort to a small number of classes. Fig. 3 shows the REDS architecture. REDS is composed of two layers: an overlay layer and a routing layer. The former manages the connections with other brokers and has mechanisms for sending and receiving packets. The latter defines the routing strategy. The routing layer communicates with the overlay layer through the Overlay component, which relies on a TopologyManager component to establish and monitor connections with other brokers, and on a Transport component to receive and send packets.

REDS packets are composed of a Subject and a Payload (any Serializable object). Developers can register different traffic classes on the Overlay: for each traffic class, the Overlay creates a separate queue for storing incoming packets coming from the Transport. Developers can specify the maximum size of each queue, and how packets are associated with traffic classes based on their subjects.

The Overlay delivers packets to the Router, using the method notifyDataArrived. Based on the packet’s subject, the Router selects the PacketForwarder in charge of processing it (if any). The actual routing strategy is implemented within the various PacketForwarder components installed on top of the Router. In processing an incoming packet, a PacketForwarder can: (i) modify its payload; (ii) forward it to one or more brokers; or (iii) generate and send new packets.

4.2 Implementing ASIA in REDS

Our implementation of ASIA in REDS requires fewer than 9,000 new lines of code, and no changes to existing code. We exploit the standard REDS Overlay layer, which adopts TCP for communication between brokers. In the routing layer, we use the standard Router component, limiting the scope of our implementation to the definition of two new PacketForwarder components: a TreeBuilder, as well as an ASIAPacketForwarder.

The TreeBuilder creates one forwarding tree on top of the existing overlay network. ASIA assumes an acyclic topology: the TreeBuilder creates it by electing a leader node \( n_L \) and then running a protocol that creates the shortest path spanning tree rooted at \( n_L \). The ASIAPacketForwarder is responsible for processing both packets for event propagation, and packets for aggregation. With reference to [3] the ASIAPacketForwarder implements both the protocols for offering a publish/subscribe service (e.g., protocols for advertisements, subscriptions, and publications forwarding), and the aggregators for evaluating aggregation functions.

Besides implementing a new PacketForwarder, we define new types of packets for propagating aggregation requests and updates, and modify the packets used for publish/subscribe communication so that they can piggyback aggregation data, using subtype polymorphism (see §3.2).

4.2.1 Aggregation Mechanisms

For each aggregation function \( f \) (see §3.3), an aggregator in each broker needs to perform the following steps:

- it creates a data structure to store (i) the local view of the value of the aggregation function \( \bar{\tau} \); (ii) the last value sent to each neighbor \( k (\bar{v}_k) \); (iii) the maximum imprecision tolerated by each neighbor \( k (\bar{v}_k) \);
- it starts monitoring the information that may change its local view of \( \bar{\tau} \) (e.g., a new subscription);
• every time \( \forall \) changes, it checks, if it needs to send updates to its neighbors by applying \( f \) and determining if the result falls beyond the boundary value for imprecision—i.e., a range check for additive aggregators;
• it determines if the updates can be piggybacked using existing messages.

Within this approach, defining and deploying a new aggregator is straightforward. Developers need to specify (i) what triggers an update in the value of the aggregation function, i.e., which specific data has to be monitored; and (ii) how to compute the value of the aggregation function. Functions currently accept and return simple Java Objects.

To evaluate ASIA, we implemented several aggregators, including subscriber and publisher counts, rate of subscriptions and publications (over a time-based sliding window), and active publishers (in a time-based window). We present some of the results based on these aggregators in the following section.

### 4.2.2 Runtime Distribution of Aggregation Functions

To allow for maximum flexibility, we support distributing new aggregation functions at runtime through clients supplying additional advice. Clients that want to supply a new, custom aggregation function write a corresponding class (implementing the common interface for all aggregation functions) and compile it locally. They then disseminate the compiled bytecode over the broker network. We make use of the existing publish/subscribe system for dissemination of code across all brokers.

Once a broker receives a message with a new aggregation function, it takes the bytecode from that message and uses a custom classloader to load that class at runtime. Classes are then instantiated using reflection, adding the new aggregation function to the respective joinpoints. While there is a risk of naming conflicts—two clients could use the same class name—this is no different to potential name collisions when using multiple libraries of any sorts, and is easily avoided by following Java naming conventions.

Weaving is simplified by assuming that different advice (represented by different classes) are independent. If this does not hold, programmers need to deal with the interaction explicitly knowing that different advice for a same joinpoint are executed in a non-deterministic order. Piggybacked information is unambiguously assigned to advice by conveying all such information inside a hashmap indexed by advice name—this is no different to potential name collisions when using multiple libraries of any sorts, and is easily avoided by following Java naming conventions.

Weaving is simplified by assuming that different advice (represented by different classes) are independent. If this does not hold, programmers need to deal with the interaction explicitly knowing that different advice for a same joinpoint are executed in a non-deterministic order. Piggybacked information is unambiguously assigned to advice by conveying all such information inside a hashmap indexed by advice name—this is no different to potential name collisions when using multiple libraries of any sorts, and is easily avoided by following Java naming conventions.

Weaving is simplified by assuming that different advice (represented by different classes) are independent. If this does not hold, programmers need to deal with the interaction explicitly knowing that different advice for a same joinpoint are executed in a non-deterministic order. Piggybacked information is unambiguously assigned to advice by conveying all such information inside a hashmap indexed by advice name—this is no different to potential name collisions when using multiple libraries of any sorts, and is easily avoided by following Java naming conventions.

Weaving is simplified by assuming that different advice (represented by different classes) are independent. If this does not hold, programmers need to deal with the interaction explicitly knowing that different advice for a same joinpoint are executed in a non-deterministic order. Piggybacked information is unambiguously assigned to advice by conveying all such information inside a hashmap indexed by advice name—this is no different to potential name collisions when using multiple libraries of any sorts, and is easily avoided by following Java naming conventions.

Weaving is simplified by assuming that different advice (represented by different classes) are independent. If this does not hold, programmers need to deal with the interaction explicitly knowing that different advice for a same joinpoint are executed in a non-deterministic order. Piggybacked information is unambiguously assigned to advice by conveying all such information inside a hashmap indexed by advice name—this is no different to potential name collisions when using multiple libraries of any sorts, and is easily avoided by following Java naming conventions.

### 5. EVALUATION

Our experimental evaluation has two goals:

1. We investigate the costs and benefits of introducing the ASIA model within a publish/subscribe middleware. For this purpose, we (i) compare ASIA with the REDS publish/subscribe system and measure the overhead introduced by the aggregation mechanism; (ii) compare ASIA with REDS and an external aggregation system running side by side.

2. We measure the cost of dynamically deploying new aggregate functions in the broker network at run-time. We monitor the behavior of brokers while injecting new functions supplied by the clients.

#### 5.1 Experiment Setup

For our experiments, we define a reference scenario with the parameters shown in Table 2. We consider a network of 16 brokers, each connected to 3 other brokers and serving 100 clients. Each client publishes 1 event per second on average and is subscribed to 5 topics out of 500. Since subscriptions select events based on their topics only, each client receives 1% of the published events on average. Topics of publications and subscriptions are normally distributed, and each client issues one subscription (or unsubscription) every 10 seconds.

We use 32 Intel Core i7 nodes, each with 8 cores at 3.4 GHz and 8 GiB of RAM, running Linux version 3.0.3. Each broker is deployed on a separate node. An additional 16 nodes host the clients that produce network traffic; all clients connected to a given broker are hosted on the same node. When measuring the delay of packets, we deploy all the clients on a single physical node, but connected to different brokers. This allows us to measure time in a precise way, based on the unique clock of the host machine. We also reduce the overall number of clients running concurrently to 8 to have sufficient resources to run them in parallel.

Depending on the specific test, we adopt the subscriber count or the publisher count functions. Each client requests a subscriber (publisher) count for three event topics. When stressing the system, we configure it towards a maximum aggregate imprecision of zero, i.e., updates are always propagated. In §5.2.2, we explore the effect of increasing the imprecision. We use REDS with two different traffic classes, one for delivering events and one other to distribute subscriptions, aggregate updates, and new functionalities in terms of deployable code.

All the experiments presented below have been repeated five times, with different seeds for generating events and subscriptions. Each point plots the average value measured.

#### 5.2 Overhead and Benefits

We evaluated the overhead of ASIA compared to a publish/subscribe-only middleware and studied the advantages of integrating the aggregation services with the publish/subscribe system.

##### 5.2.1 Investigation of the Costs of Aggregation

We investigate the costs of the aggregation mechanisms introduced in ASIA. To do so, we compare it against the REDS middleware, which only provides a publish/subscribe service with no support for aggregation. Both systems are

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of brokers</td>
<td>16</td>
</tr>
<tr>
<td>Average subscriptions per client</td>
<td>5</td>
</tr>
<tr>
<td>Number of connections</td>
<td>3</td>
</tr>
<tr>
<td>Number of different topics</td>
<td>500</td>
</tr>
<tr>
<td>Average link latency</td>
<td>0.2ms</td>
</tr>
<tr>
<td>Subscription/unsubscription ratio</td>
<td>50% / 50%</td>
</tr>
<tr>
<td>Average publication rate (per client)</td>
<td>1 ev/s</td>
</tr>
<tr>
<td>Average subscription rate (per client)</td>
<td>0.1 subsc./s</td>
</tr>
<tr>
<td>Number of clients per broker</td>
<td>100</td>
</tr>
<tr>
<td>Number of clients (delay tests)</td>
<td>8</td>
</tr>
<tr>
<td>Average aggregation requests per client</td>
<td>3</td>
</tr>
<tr>
<td>Maximum imprecision</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: Parameters used in the reference scenario
configured to build and use the same overlay topology, and the same protocol for distributing subscriptions and events.

In this experiment, we configured all clients to publish events at their maximum rate, and we measure the overall number of events received by subscribers per second. In ASIA, we also configured publishers to ask for subscriber counts. Since this value is affected by the number of subscriptions generated by clients, we repeated the experiment while changing the number of subscriptions generated for every publication.

Figure 4 shows the maximum throughput we measured for the two systems. As the results show, the overhead of aggregate update computation and distribution is negligible: ASIA exhibits a maximum throughput that is almost identical to the unmodified publish/subscribe system. This is true even when we push the system to the extreme case in which subscriptions/unsubscriptions are generated at the same rate as publications.

5.2.2 Advantages of Integration

We studied the advantages of integrating the publish/subscribe and aggregation services, as proposed in the ASIA model. To do so, we compare ASIA with a solution that adopts two separate systems to provide the publish/subscribe and aggregation services, deployed side by side on the same brokers. We refer to this solution as PS+Agg.

Figure 5 compares the overall traffic generated by ASIA and PS+Agg when changing the maximum imprecision accepted by clients. As a baseline, we also plot the traffic generated by a traditional publish/subscribe system that does not implement any aggregation mechanism. Notice that we do not consider the traffic of events, which is identical for all the systems under analysis. As Figure 5 shows, the overhead introduced by the aggregation mechanisms of ASIA is limited, and decreases with the maximum imprecision tolerated by clients. At the same time, Figure 6 highlights the advantages of ASIA with respect to the PS+Agg solution. By piggybacking aggregation updates, ASIA is capable of significantly reducing the network traffic.

We conducted several other experiments that we omit due to space constraints. Interested readers can refer to the ASIA project webpage\footnote{http://www.dvs.tu-darmstadt.de/research/events/asia/} where we investigate in greater detail the performance of ASIA in terms of throughput, network traffic, and delay for delivering events and aggregate updates.

Based on these results, we conclude that the ASIA aggregation model is a practical augmentation of distributed publish/subscribe middleware to provide components interacting via implicit invocations with aggregated crucial information that can be computed in a cheap, distributed manner.

5.3 Dynamic Function Loading

Having demonstrated the benefits of an integrated aggregation mechanism for publish/subscribe systems, we now investigate the cost of real-time deployment of new (aggregation) functions into the broker network.

We configure clients to periodically send special DL Packets (Dynamic Loading Packets), which include the Java bytecode of new functions. When a broker receives a DL Packet, it extracts the bytecode, loads the new functionality, and
starts executing it, thus weaving the new code into the broker as outlined in Section 1.2.2. It then forwards the packet to other brokers through the overlay network. This happens until all brokers receive the new function. For our experiments, each new function implements some mathematical computation that is executed only once at each broker.

In realistic settings, we expect the deployment of new functions to occur rarely. To stress the system, however, we performed all our tests in the extreme situation in which new functions are deployed at high rate. We investigate the overhead of function deployment when changing the generation rate of DL Packets from 1 to 1000 per second.

Initially, we measure how the presence of dynamic loading impacts on the delay for delivering events. During our experiments, clients use ASIA both as a publish/subscribe system and as an aggregation system, to collect information about the total count of publishers.

Figure 6 shows the average delay measured by clients and its 95th percentile. We compare the results we collect with a baseline system that does not offer dynamic loading. Interestingly, both the average delay and the 95th percentile oscillate around the values collected from the baseline system. We do not measure any significant overhead. Moreover, increasing the rate of DL Packets does not significantly increase the measured delay.

Considering the same experiment, Figure 7 shows the delay for propagating aggregate updates. While our implementation uses a separate traffic class for events, aggregate updates and DL Packets share the same class. We thus expect a higher impact of dynamic loading. Indeed, as shown in Figure 7, the 95th percentile is higher when dynamic loading is introduced. Moreover, it slightly increases when the rate of DL Packets becomes extremely high (over 100 DL Packets per second). Interestingly, however, even when dynamic loading occurs at high rate, the impact on the average delay remains negligible.

As a final experiment, to better understand the computational effort of dynamic loading, we monitored the CPU load of the machines hosting the brokers. The results are shown in Figure 8. Independently from the rate at which DL Packets are generated, the average CPU load remains almost constant and well below 1%. For this reason, we also plot the maximum load. In all cases, it never reaches 100%; moreover, when considering dynamic loading, this value only increases marginally.

6. RELATED WORK

Implicit Invocations.

Several programming languages support implicit invocations inherently through asynchronous “event” methods or similar constructs. Examples include ECO [21], Java/ES [13], EventJava [14], Ptolemy [58], and EScala [19]. None of these provide any other features than propagation of implicit invocations; focusing on centralized deployments, Ptolemy or EScala could easily provide information on the number of publishers per subscriber or vice-versa. In contrast, languages based on Actors (e.g., Erlang [12] and inspired by the Join Calculus [18] (e.g., Polyphonic C#—now integrated with C#), or combining the two (e.g., Scala Actors [22], Erlang Joins [44, 37]) focus on asynchronous invocations on single destinations, and thus do not support implicit invocations.

Distributed Aspects.

In general, AOP is sometimes viewed itself as a form of event-based programming which is implicit, as opposed to the explicit events mentioned above. Several authors have proposed extensions to AOP specifically for asynchronous distributed systems, including event-based aspect-oriented programming (EAOP) [11], aspects with explicit distribution (AWED) [34], and distributed aspects for distributed objects (DADO) [16]. These focus on advising one-to-one remote object invocations and thus on a different distributed programming abstraction than implicit invocations, which are specifically designed for multicast (one-to-many) interaction. Together with a design pattern, for instance based on the use of dummy proxy objects, one could certainly use any of these approaches to implement some form of implicit invocations. Their applicability for implementing the aspects presented herein is less obvious.

The specific contribution of EAOP, AWED, or DADO is to be able to advise remote joinpoints in application components. In contrast, the joinpoints introduced herein are focused on broker logic which is internal to a publish/subscribe-based application; broker networks are precisely introduced to streamline propagated information via aggregation, and thus to avoid references and dependencies between application components. Similarly, conspects [24] focus on advising application components—in order to add contextual arguments to implicit invocations—rather than on the components constituting the infrastructure implementing the implicit invocations.

Publish/Subscribe & Group Communication Systems.

Centralized publish/subscribe middleware systems, such as ActiveMQ [42], are used as a foundation for distributed applications that involve many participants, potentially joining and leaving at runtime [23]. To gain scalability, distributed publish/subscribe middleware such as Padres [17] have been developed, in which publishers and subscribers are decoupled in time, space and flow [15] by a network of brokers.

Researchers have considered additional information that can be provided by publish/subscribe middleware. Behnel et al. [1] describe various metrics that are of interest in such systems. For example, ActiveMQ publishes advisory messages providing information about the state of the server. Traditional group communication systems have a membership service that manages a list of active members for each group [16, 32, 40]. Group communication systems typi-
cally provide strong delivery guarantees that are aligned with changes in membership views. However, in contrast to ASIA, these approaches do not scale to large system sizes.

**Distributed Aggregation.**

Efforts on distributed aggregation for scalable monitoring deal with challenges in terms of robustness, scalability, and dynamism. The spectrum of aggregated information ranges from state information to a quantification of system stability.

Astrolabe is an early system for monitoring the state of distributed resources and provides summarization aggregations based on user-defined aggregation functions. Astrolabe uses a single logical aggregation tree on top of an unstructured peer-to-peer gossip protocol. The authors describe how it could support a topic-based publish/subscribe model but, in contrast to ASIA, it is unclear how aggregation would be affected by changes in the topology.

Yalangandula and Dahlin extend distributed hash tables (DHTs) into a scalable distributed information management system (SDIMS). SDIMS performs hierarchical aggregation based on attribute types and names through aggregation functions associated with a certain attribute type. Jain et al. introduce network imprecision (NI), a consistency metric for large-scale distributed systems. NI allows a system to quantify its stability in terms of currently (un)reachable nodes and number of updates that might have been repeatedly processed due to network failures. The STAR protocol adaptively sets the precision constraints for processing aggregate queries and is used by NI. However, such generic aggregation systems cannot leverage specific properties of publish/subscribe systems, such as overlay topologies or exchanged messages. Consequently aggregation trees do not necessarily match routing trees, resulting in inefficiency and delayed adaptation to system changes.

Other proposed aggregation solutions specialize on certain goals. For example, Cheung and Jacobsen propose an algorithm that probabilistically traces publication messages through replies with data aggregation in order to find the best broker for connecting a new publisher. Migliavacca and Cugola provide an approach for handling replies in publish/subscribe communication. REMO builds many optimized routing trees for different monitoring tasks, taking available resources into account and allowing for efficient data aggregation towards the root of the tree. Adam2 estimates the distribution of a value by using a gossip-based algorithm, in which nodes exchange aggregation instances. All of these systems cannot support the application-specific aggregation requirements of generic applications.

The paradigm of stream processing (SP) bears certain resemblances with aggregation as described herein. While SP systems also make use of distributed overlay networks, they pursue a different goal: every node performs a different operation, and the end result is not inherently multicast.

**Reflective Middleware.**

Middleware systems have long been designed with reflective features in mind. Object Request Brokers (ORBs) have been a particular early target of such design. ORBs are analogous to our event brokers, except that they focus on remotely accessible objects and explicit invocations thereon whereas the latter implement implicit invocations. Examples of reflective ORBs include the DynamicTao and Open ORB systems. Similar to more recent and generic work on adaptive middleware and composable systems relying on reflection features, these approaches aim at improving the internal design and other non-functional characteristics (e.g., adaptivity, extensibility, and QoS) of a software system. Preservation of external interfaces is a declared goal. In contrast the goal of ASIA is the addition of features for individual applications or families of applications, thus extending the external interfaces of a system.

**7. CONCLUSIONS**

Publish/subscribe systems for implicit invocations enable decoupling among the communication components in a distributed system. In practice, components often require some information on others, and rely on ad-hoc middleware services to obtain such information. Progressing from this premise, we introduced ASIA, a model for augmenting a distributed publish/subscribe system with integrated mechanisms to collect aggregated information.

Since different applications may require different information about interacting components, our solution does not provide a predefined set of aggregated functions. Rather, it provides domain-specific joinpoints which developers can program advice for that compute application-specific aggregation functions.

In terms of future work, we are investigating a more formal definition of the conditions required for safe modification of broker behavior at run-time. In addition, we are working on an implementation of ASIA for the ActiveMQ publish/subscribe system. Last but not least, we are considering leveraging the well-matured AspectJ tool chain, by compiling ASIA advice to AspectJ advice, and exploiting existing dynamic weaving support for runtime deployment of aggregation functions.

**8. REFERENCES**
