Towards a Common API for Publish/Subscribe

Peter Pietzuch
Department of Computing
Imperial College
London, United Kingdom
prp@doc.ic.ac.uk

David Eyers Samuel Kounev
Computer Laboratory
University of Cambridge
Cambridge, United Kingdom
{dme26,sk507}@cam.ac.uk

Brian Shand
Clinical and Biomedical
Computing Unit
Cambridge, United Kingdom
Brian.Shand@cbcu.cam.ac.uk

ABSTRACT

Over the last decade a wide range of publish/subscribe (pub/sub) systems have come out of the research community. However, there is little consensus on a common pub/sub API, which would facilitate innovation, encourage application building, and simplify the evaluation of existing prototypes. Industry pub/sub standards tend to be overly complex, technology-centric, and hard to extend, thus limiting their applicability in research systems.

In this paper we propose a common API for pub/sub that is tailored towards research requirements. The API supports three levels of compliance (with optional extensions): the lowest level specifies abstract operations without prescribing an implementation or data model; medium compliance describes interactions using a light-weight XML-RPC mechanism; finally, the highest level of compliance enforces an XML-RPC data model, enabling systems to understand each other's event and subscription semantics. We show that, by following this flexible approach with emphasis on extensibility, our API can be supported by many prototype systems with little effort.

1. INTRODUCTION

Many different distributed applications benefit from using a publish/subscribe (pub/sub) system to handle communication between components. Research into pub/sub systems has led to a variety of different prototype systems [2, 17, 4, 9] that differ in terms of architecture, routing and matching algorithms, and event and subscriptions semantics. Other research has investigated services on top of pub/sub systems, including composite event detection, heterogeneous system federation, and access control models.

Unfortunately there is no agreement within the research community on a common *application programming interface* (API) for pub/sub systems. Consequently most research prototypes are incompatible with each other. This is an undesirable situation because it limits innovation within the community: applications built on top of pub/sub prototypes are tied to a particular system, there is no easy way to compare different prototypes, and each system must come with its own slightly different API description. Furthermore, research prototypes often "die" when students graduate.

Although several standards for industrial pub/sub systems exist [21, 11, 12, 27, 10], we argue that they are inappropriate for research systems. Commercial standards tend to be unambiguously specified to ensure maximum interoperability, making them heavy-weight, complex to understand, and difficult to implement. They also put emphasis on backwards compatibility and provide limited extensibility. In many cases, they are meant to push a given technology or cement the dominance of a particular commercial system. In contrast, a research standard should be light-weight, easy to comply with, encourage innovation through extensibility, and platform-independent.

Motivated by similar efforts in other areas [6], this paper proposes a common API for pub/sub systems that is tailored towards the needs of the research community. Our proposal explicitly acknowledges the tension between interoperability, extensibility, and ease-of-use by dividing our API into a *core* API to ensure interoperability and an *optional* API for supporting new features. Orthogonal to that, we define three *compliance levels* L1–L3. For L1-compliance we define abstract calls for pub/sub functionality without prescribing any particular implementation or data model. Our intention is that most existing systems come close to L1-compliance by default. This results in a common starting point and encourages the adoption of higher compliance levels. L2-compliance requires the use of XML-RPC calls, thus allowing communication between systems without necessary agreement on event and subscription models. Finally, L3-compliance requires XML-RPC calls to use an XML-RPC-based event and subscription model, leading to full interoperability.

The rest of the paper is organised as follows. §2 gives an overview of pub/sub systems and standards. After describing the design space for a common API in §3, we present our proposed API with its three compliance levels in §4. In §5 we demonstrate how our common API can be applied to existing systems. We conclude the paper in §6.

2. EXISTING SYSTEMS AND STANDARDS

Next we present an overview of current research prototypes, commercial systems and standards.

2.1 Research Prototypes

Depending on the subscription model, pub/sub systems

DEBS '07, June 20–22, 2007 Toronto, Ontario, Canada

System	APIs	Wire Protocols	Event Model	Subscription Model
Scribe	Java, C#	TCP	Unstructured data	Topic-based
Siena	Java, C++	TCP, UDP	Structured record (set of named, typed attributes)	Content-based, Conjunction of attribute filters
Hermes	XML, Java	XML over TCP	XML document	Type- and attribute-based, XPath queries
Gryphon	JMS	TCP	JMS message	Topic- and property-based, JMS selectors
Rebeca	Java	TCP	Structured record (set of named, typed attributes)	Content-based, Conjunction of attribute filters
XMessages	Java, C++	XML over SOAP	XML document	Channel- and content-based SQL-like queries
Narada-Brokering	Java, JMS, C++, WS-Eventing	TCP, UDP, HTTP, SSL, RTP, HHMS, GridFTP, etc.	Java object, JMS message, XML document	Topic- and content-based, Attribute filters, XPath queries, JMS selectors, etc.
ECho	CORBA, Java	NDR/PBIO, TCP, UDP, ATM	C-style structure	Channel- and content-based, Custom C-style filter functions

Table 1: Selected Pub/Sub Research Prototypes

can be broadly classified into *topic-based* (channel-based/ subject-based), *content-based* or *type-based*.

An example of a topic-based system is *Scribe* [3], developed at Microsoft Research. Scribe is a large-scale distributed event notification system built on top of Pastry, a peer-to-peer object location and routing substrate overlayed on the Internet. Pastry is used in Scribe to maintain topics and subscriptions, and to build efficient multicast trees. A rendezvous-based routing algorithm is used to route events from publishers to subscribers.

One of the first implementations of a distributed content-based pub/sub system is *Siena* [2]. *Siena* is based on a broker network architecture (deployed over a TCP/UDP transport layer) and uses a filtering-based routing algorithm. Advertisements are introduced to optimize the event routing process and avoid subscription flooding. *Siena* offers two APIs for client applications, one in Java and one in C++. Events in *Siena* are structured records (sets of named and typed attributes) and subscriptions are conjunctions of attribute filters. Another content-based pub/sub system similar to *Siena* is Rebeca [9]. Rebeca provides a generic routing engine that has been extended with several efficient filtering-based routing algorithms such as identity-based, covering-based, and merging-based routing.

Hermes [17] is an example of a type-based pub/sub system. Hermes is implemented as a network of event brokers deployed using an overlay network for efficient rendezvous-based routing and fault-tolerance. Unlike *Siena*, Hermes uses an XML-based event model in which event types are defined as XML Schema documents [26]. The subscription model is a typed variant of content-based pub/sub, called *type- and attribute-based pub/sub*. Hermes also offers some higher-level middleware services such as advanced security, composite event detection and congestion control. In addition to a Java interface, it provides an XML interface that automatically translates between XML and Java objects.

Further content- and/or type-based pub/sub systems include Jedi [4], Elvin [22], Gryphon [24], ECho [7], XMessages [28], Le Subscribe [16], WebFilter[15], Choreca [23], IndiQoS [1] and Cobra [19]. Recently, two pub/sub systems with dynamic reconfiguration and adaptation capabilities have been proposed, GREEN [20] and REDS [5]. Another mature prototype system is NaradaBrokering [14], developed at Indiana University. NaradaBrokering stands out with its support for a number of different client APIs and wire protocols as well as its flexible subscription model.

Table 1 compares a representative subset of the systems mentioned above in terms of their APIs, wire protocols, and event and subscription models. As evident from the table, the existing pub/sub research prototypes expose a wide range of different APIs and data models which are highly incompatible with one another.

2.2 Commercial Systems and Standards

In addition to the numerous research prototypes, commercial systems have appeared that support the pub/sub communication model. Some of the most popular ones include IBM WebSphere MQ (IBM MQSeries), TIB/RV, Progress SonicMQ, Sun Java System Message Queue, BEA WebLogic Server and Fiorano FioranoMQ. At the same time industry standards related to pub/sub systems exist, such as the Java Message Service (JMS) [21], the CORBA Event Service [11], the CORBA Notification Service [12] and the OMG Data Distribution Service [13].

JMS defines a vendor-agnostic Java API for message-oriented middleware. It supports two communication modes—point-to-point and pub/sub. The former is used for one-to-one communication through message queues managed by the JMS server. The latter provides a topic-based pub/ sub service with limited content-based filtering capabilities. Filters are specified using a subset of the SQL92 conditional expression syntax.

In the Web services arena, two standards have been proposed, the WS-Eventing [27] and WS-Notification [10]. Both of them define a mechanism that allows Web Services to exchange data through asynchronous messaging based on the pub/sub communication model. Efforts are underway to eventually consolidate these two standards into a single WS-EventNotification specification.

3. DESIGN SPACE

In this section we survey the design space for a common pub/sub API by discussing the main requirements.

Ease-of-use: The simplicity of adoption should be a major driving factor behind the API design. The implementation of a research prototype usually focuses on a novel contribution, which is why developers do not want to waste time achieving compliance with a hundred-page standard.

In general, research standards can be leaner for several reasons: (1) They do not need to provide the same level of detail because interoperability between systems without testing is not expected. This in contrast with industry,

where it is infeasible to validate the compliance of an implementation against all other existing systems. (2) Research standards are not encumbered by the requirement to be backwards compatible with existing technology and systems. (3) Research standards do need to make allowances for particular technologies or outside interests. As can be witnessed by the current division between the WS-Eventing and WS-Notification efforts [8], competing industry consortia can slow down the standardisation process and lead to feature-heavy compromises.

Interoperability: A common API should obviously aim to support interoperability between different systems. The API should be platform- and language-independent to reflect the variety of implementation environments. It should also hide implementation details such as its architecture or the matching and routing algorithms employed. Unfortunately, an API that is too abstract and generic is often not expressive enough to be useful in practise. We argue that a solution to this dilemma is the specification of multiple *compliance levels* for the API.

At a low compliance level, we define a range of RPC (remote procedure call)-style function calls that capture the essence of pub/sub communication without depending on a given language or implementation. This means that pub/sub systems can agree on externally-visible API calls and a common terminology, while maintaining a language-specific implementation, *e.g.,* using a Java Interface. Applications and benchmark suites can link directly against the implementation to get local call semantics. For remote API calls, any language features, such as Java object serialisation, can be used to obtain a rapid distributed implementation.

For higher compliance levels, we need to define a wire protocol to support interoperability between systems using different languages and platforms. Instead of inventing our own protocols, we acknowledge the dominance of existing web standards, though standards such as SOAP and XML Schema are cumbersome to implement and appear overly complex. Therefore we advocate a light-weight approach using the well-established HTTP and XML standards according to the five-page XML-RPC specification [29]. XML-RPC describes a mechanism for making RPC-style function calls over the network. Due to this simplicity, it has been used successfully by other research systems [18].

To achieve full compatibility, pub/sub systems must understand each other's event and subscription models. Data models for pub/sub systems are often domain-dependent and are still an area of active research. In order to balance interoperability with the potential for innovation, we only require pub/sub systems to support our data model at the highest compliance level.

For the *event model*, open web standards can provide the necessary platform-independence. However, we believe that the original XML standard is too constraining in its data types, whereas the XML Schema extension [26] is too powerful and complex. We choose a middle ground by defining events using the data model provided by XML-RPC. Similar to other specifications such as JMS, we divide events into a *filterable* and an *opaque* part to optimise processing during event matching and routing.

There is even less agreement on a common *subscription model* for pub/sub because such models are often tightly-coupled with event matching algorithms. To ensure compatibility with XML, we propose a subset of the XPath language [25] for stating subscriptions. XPath supports the matching of parts of XML documents through the evaluation of expressions, without suffering from the complexity of other specifications such as XQuery. By restricting ourselves to a portion of XPath, we can match its expressiveness with that of most other subscription languages in content-based pub/sub systems.

Extensibility: A common API should facilitate research by encouraging the development of novel pub/sub services and semantics. We support the extension of the API with optional API calls and additional data passed alongside events and subscriptions. Our hope is that this will create an ecosystem of different API extensions, with the successful ones becoming more wide-spread.

When supporting extensions, an advantage of using XML documents is that unknown parts related to optional extensions can be ignored by implementations that do not understand them. Naming is also an important issue so that conflicts between extensions proposed by different parties are avoided.

4. COMMON PUB/SUB API

In this section we describe the three compliance levels L1–L3 and their APIs.

4.1 L1: Abstract API

At L1 compliance, we define abstract function calls that capture the core of any pub/sub system, namely the subscription to and publication of events. L1 compliance is useful to broadly classify pub/sub systems and reach a common terminology for their external API. Any particular implementation of these calls in any language is acceptable. We also define a few optional API calls and provide an overview of areas for further optional extensions in pub/sub.

4.1.1 Core API

publish(event)

The *publish* call publishes an *event*. Events can be of any data type supported by the given implementation languages and may also contain meta-data.

subscribe(filter_expr, notify_cb, expiry) → sub_handle

unsubscribe(sub_handle)

The *subscribe* call takes a *filter expression* in any filtering language, a reference to a *notify callback* for event delivery, and an *expiry* time for the subscription registration. It returns a *subscription handle* that can be used to refer to this subscription registration in an *unsubscribe* call.

In a topic-based pub/sub system, the filter expression may just consist of a topic name. By including a lease duration in the call, we advocate system designs based on a soft-state approach for automatically removing stale state in the pub/sub system. If this is not supported, an expiry time of zero indicates an infinite lease. Note that the subscription handle may be used as an authentication token that identifies the original owner of a subscription.

notify_cb(sub_handle, event)

This function is called by the pub/sub system to deliver a matching event. The original subscription handle is included so that the client can determine the subscription that caused this notification.

4.1.2 Optional API

advertise(filter_expr, expiry) → adv_handle

unadvertise(adv_handle)

In many existing pub/sub systems, clients use an *advertise* call to announce their intention of publishing certain events. This is used by the system to create state or optimise routing. The advertise call is also lease-based with an *expiry* time and returns a *advertisement handle* for future referral.

renew_lease(adv|sub_handle, expiry)

This function call is for renewing a subscription or advertisement lease for systems that support this.

publish_ext(event, ext_data)

subscribe_ext(filter_expr, notify_cb, expiry, ext_data) → sub_handle

While some extensions will provide their own API calls, others will enrich the semantics of the publish and subscribe calls. Options that are tightly coupled with the event data will be passed alongside it; other settings may use optional *publish_ext* and *subscribe_ext* calls that each take an *extension data* parameter. This parameter can express advanced preferences, *e.g.,* delivery modes or reliability semantics.

4.1.3 Areas for extensions

Firewall traversal. Firewalls, or other network address translation (NAT) devices, allow hosts to initiate connections but not to receive them. This would prevent the core API from pushing event notifications to subscribers. For firewall traversal, a host could subscribe with a blank *notify_cb* and pull events with the following API extension:

notify_cb_wait(sub_handles, expiry)

Event types. Pub/sub systems that are tightly integrated with programming languages can benefit from using explicitly typed events. For example, the inheritance hierarchy of an object-oriented programming language can be mapped to a hierarchy of event types. Programmer convenience is increased (and error risks decreased)—explicit event (un)marshaling will not be required. Optional API calls will be needed to (de)register event types.

Quality-of-Service. Rather than best-effort service, persistent storage may be employed by a pub/sub system with transactional event semantics. Common *delivery guarantees* are: at least once, at most once, and exactly-once, and event ordering requirements. Extensions can also put constraints on *latency requirements*, *e.g.,* by respecting bounds on delivery latency. Finally, *event priorities* can ensure that important events move up queues.

Security. *Authentication.* Although pub/sub systems generally aim to route events agnostic to their source and destinations, some applications might require user, group or role *authentication* and add this provenance data to events. *Encryption and verification* can be used to protect the event in transit and to provide *access control* schemes.

Performance and statistics. Another area for extension particularly relevant to researchers relates to performance measurement and statistics gathering. "System-level attributes" could profile current delivery times and event routes. Standardising measurement APIs and a set of common performance metrics will hopefully promote the development of accepted performance benchmarks.

4.2 L2: XML-RPC API

At L2 compliance, we define the L1 API calls using XML-RPC. This means that different implementations share the same over-the-wire protocol. However, they may or may not share the same event and subscription model used in the calls. Error reporting is done using XML-RPC's *faultCode* tag. Simple authentication is supported through the standard HTTP mechanism. By using HTTP port 80, calls are also more likely to traverse firewalls and handle web proxies.

4.2.1 Core API

pubsub.core.**publish**(struct event)

The *publish* call takes a *struct*, which can contain any valid XML-RPC data types including *base64*-encoded proprietary binary formats. As for L1, the event may also include data used by optional extensions.

base64 pubsub.core.**subscribe**(string filter_expr, string notify_url, dateTime.iso8601 expiry)

pubsub.core.**unsubscribe**(base64 sub_handle)

For the XML-RPC calls, we use *base64*-encoded handles to refer to subscriptions (and advertisements), allowing them to be any length, *e.g.,* to guarantee global uniqueness. *Filter expressions* are strings in any language, and the *notification url* refers to a local XML-RPC endpoint. *Expiry* times are specified with a standard XML-RPC type.

pubsub.core.**notify**(base64 sub_handle, struct event)

4.2.2 Optional API

base64 pubsub.opt.**advertise**(string filter_expr, dateTime.iso8601 expiry)

pubsub.opt.**unadvertise**(base64 adv_handle)

pubsub.opt.**renew_lease**(base64 adv|sub_handle, dateTime.iso8601 expiry)

pubsub.opt.**publish_ext**(struct event, struct ext_data)

base64 pubsub.opt.**subscribe_ext**(string filter_expr, string notify_url, dateTime.iso8601 expiry, struct ext_data)

In the optional XML-RPC API, we define the same function calls as required for L1. *Extension data* for publish and subscribe calls is passed in a *struct*.

4.3 L3: XML-RPC API with Data Model

Level 3 compliant pub/sub systems use a common model for event data and subscriptions. This allows them to work together transparently, either as direct replacements or cooperatively in a federated event-based system.

The key contribution of L3 is that the same events, subscriptions and filter expressions can be used on multiple platforms, independently of how the events are managed internally within each pub/sub system.

4.3.1 Core API

In L3, an event must be represented as a valid XML-RPC parameter payload, in a *struct* with three parts: a filterable

```
<struct>
  <member><name>filterable</name>
    <value>Any valid XML-RPC data</value></member>
  <member><name>event data</name>
    <value>Any valid XML-RPC data</value></member>
  ...
  <member><name>pubsub.ext.extension_name</name>
    <value>Any valid XML-RPC data</value></member>
  ...
</struct>
```

Figure 1: The L3 compliant XML-RPC data format allows one `filterable` block, and any number of other data blocks. Extension data may be passed alongside events.

```
<struct>
  <member><name>filterable</name>
    <value><struct>
      <member><name>topic</name><value>news</value>
      </member>
      <member><name>date</name><value><dateTime.iso8601>
          2007-03-15</dateTime.iso8601></value></member>
      <member><name>title</name><value>
          Drama on the high seas</value></member>
    </struct></value>
  </member>
  <member><name>contents</name>
    <value><string>In the news today, ...</string>
    </value></member>
</struct>
```

Figure 2: Example of a Level 3 compliant event.

section, any extension-specific data, and the remainder of the event data. Subscription filters take the form of XPath queries and operate only on the filterable section of each event. Furthermore, L3 compliant systems must also satisfy the L2 properties above, and use the same method API.

Figure 1 illustrates the essential event structure conventions required for L3 compliance:

- The event must be an XML-RPC `struct`.
- The `filterable` block can hold any XML-RPC data. But, if the event has a topic, then the `filterable` block must also be a `struct`, with a `string`-formatted `topic` member containing the topic name.
- Pub/sub extension data is carried in blocks with names starting `pubsub.ext`.
- Subscription filters are expressed as XPath query strings returning a `boolean` result. `true` signals a match. For simplicity, and closer consistency with existing pub/sub systems, we restrict the required subset of XPath to exclude complex features such as sibling relationships or general arithmetic.

Figure 2 is a concrete example of a simple event representing a news story. This event could be matched against the following filter expression, which matches all news stories dated 1 April 2007:

```
boolean(/struct/member[name="topic" and value="news"])
and boolean(/struct/member[name="date" and
                  value/dateTime.iso8601="2007-04-01"])
```

This model is able to represent complex content-based filters, including parameter ranges in tuple spaces. For example, SIENA tuples could simply be transformed into the L3 event format, and SIENA subscription filters into corresponding XPath queries. Conversely, a broad range of L3 events and filters could efficiently be transformed into SIENA or Hermes equivalents. The transformation must be done only at the end points (at publishers and subscribers) and should not affect the internal mechanisms used for matching and routing events. The overhead of the transformation can be measured and taken into account when using the API to compare pub/sub systems.

In this example, both the event topic and a specific attribute are used in the filter. However, the filter cannot act on the non-filterable message contents. In a Python client, a subscriber might be notified of the above event via an XML-RPC call to `notify(sub_handle, event)` with `event = {'filterable' : {'topic':'news', 'date':DateTime( '2007-03-15'), 'title':'Drama on the high seas'}, 'contents' : 'In the news today, ...'}`. Thus appropriate language bindings enable pub/sub clients to use a range of L3 compliant systems effectively, without any alteration.

4.3.2 Optional API

Events carry blocks of extension-specific data to allow extensions to coexist. Each block is tagged with an unambiguous name. We advocate Java package style hierarchical naming for the extensions, *e.g.,* `pubsub.ext.event_priority`.

Ordinary event data and extension blocks have different names, allowing them to be merged into a single event object, or passed as separate parameters to the L3 API. This separation ensures that ordinary publishers and subscribers are never exposed to extension data in events, even when extensions are used in transit, *e.g.,* effecting expedited delivery through congested brokers' queues using the aforementioned priority extension.

While the XML-RPC interface provides excellent portability for event systems, some high performance applications may need to use a native API for speed, *e.g.,* for zero-copy support for large blocks of binary data. This section defines L3 consistency for a native API, enabling native and XML-RPC clients to operate consistently:

Native Events. The native API must provide functions for transforming the native event representation into a canonical XML-RPC form and vice versa.

Subscription Filters. For L3 compliance native and XML-specified filter expressions should have the same expressive power. All filters should respect the same concept of which parts of an event can be filtered.

In effect, the native and XML-RPC interfaces must have the same theoretical filter model, defined below. But this abstract model need not be implemented programmatically.

- Each event has three parts:
  event := (filterable_part, opaque_part, ext_part)
- Two functions are defined to extract the filterable part, and the topic name (if any):
  filterable(event) → filterable_part
  topic(filterable_part) → topic_name
- Then a subscription's filter **filter_expr** is effectively a binary decision function: **filter_expr(filterable_part) → match** where **match** ∈ {**Yes**,**No**}
- Extended subscriptions may also operate on the event's extensions **ext_part**. Topic-based subscriptions are just well-known filters.

This model has the advantage that it allows different pub/sub systems to have the same essential concept of event filtering, but with no restriction on how the data is represented internally. For example, the internal filter representation might be a pre-computed, optimised decision tree, with events stored in a custom binary format.

5. CASE STUDIES

This section demonstrates that a set of pub/sub systems can be brought to compliance in the sense of this paper.

Siena is L1 compliant. The methods below are from its Java interface's `siena.HierarchicalDispatcher` class.

Siena's `subscribe` method corresponds to the L1 API call with expiry set to zero. The `Filter` and `Notifiable` arguments correspond to the L1 filter expression and notify callback. This pair of Java objects taken together provides a handle for the subscription (as opposed to an explicit handle), as used by *Siena*'s L1 compliant `unsubscribe` method. *Siena* does not employ XML-RPC, so for L2 compliance its `Notifiable` and `Filter` arguments need to be serialised into XML. Also, an XML-RPC URL callback is required.

Siena defines a `Notifiable` interface with a `notify` method that corresponds to the notify call-back in the L1 API. *Siena* does not provide this method with a subscription handle, but in Java creating multiple instances of the `Notifiable` object achieves the same goal. *Siena*'s `publish` method is L1 compliant: `Notification` instances are L1 events. L2 compliance can be achieved by serialising events in the manner described above. *Siena* `Notification`s are sets of name/typed-value pairs. As mentioned in §4.3.1, L3 compliance will require a bridge between event and filter formats.

Hermes has an XML-based API with Java language binding (cf. §2). It is L1 compliant and could be made L2/L3 compliant with little effort. Hermes' `publish` method has three arguments: a publisher handle, security credentials and an event object. The first two parameters can be serialised into XML and packaged in an `extension data` struct passed to our L2 `publish_ext` method. Hermes event objects can be automatically translated into XML that can then be passed into the `event` parameter. The same approach could translate Hermes' `subscribe` operation into an L2 compliant XML-RPC. Subscriptions in Hermes are specified by providing an event type and content-based filter expression. Filter expressions are XPath queries, and thus are L3 compliant.

Scribe is L1 compliant. Its API has four methods: `create`, `join`, `leave` and `multicast`. The `create` method creates new multicast groups that correspond to a topic. Scribe's `join` method corresponds to our `subscribe` operation, with its `groupId`, and `messageHandler` arguments corresponding to our L1 filter expression and notify callback. The `leave` operation corresponds to L1's `unsubscribe`, and `multicast` can be mapped to L1's `publish`. Like Hermes, Scribe's API calls include credentials that can be passed through our extension mechanism. It would be straightforward to make Scribe L3 compliant by serialising method parameters into XML and encapsulating them into our L3 XML-RPC messages.

6. CONCLUSIONS

In this paper we presented a light-weight, flexible common API for pub/sub systems. By focusing on extensibility and ease-of-implementation, we want to encourage wide-spread adoption in research prototypes. Hopefully this will lead to increased application building on top of existing pub/sub systems and focus research on higher-level services. As future work, we intend to build wrappers for L3-compliance around a number of pub/sub systems to demonstrate the ease of the API and its performance implications. We will then conduct a study to compare the event matching and routing performance of these distributed pub/sub systems.

7. REFERENCES

[1] N. Carvalho, F. Araujo, and L. Rodrigues. Scalable QoS-Based Event Routing in Publish-Subscribe Systems. In *Proc. of NCA'05*, Washington, DC, USA, 2005.

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

[3] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A Large-scale and Decentralized App-level M/C Infrastruct. *IEEE JSAC*, 20(8), Oct. 2002.

[4] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI Event-Based Infrastructure and its Applications. *IEEE TSE*, 27(9):827–850, Sept. 2001.

[5] G. Cugola and G. P. Picco. REDS: a Reconfigurable Dispatching System. In *Proc. of SEM'06*, 2006.

[6] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proc. of IPTPS'03*, Feb. 2003.

[7] G. Eisenhauer, K. Schwan, and F. Bustamante. Publish-Subscribe for High-Performance Computing. *IEEE Internet Computing*, 10(1):40–47, January 2006.

[8] Gartner. WS-Notification Standard Ratified by OASIS Still Needs Work. ID: G00144177, Oct. 2006.

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

[10] OASIS. OASIS Web Services Notification. Specification, The World Wide Web Consortium (W3C), Mar. 2004.

[11] OMG. CORBA: Event Service, Version 1.0. Specification, Object Management Group (OMG), Mar. 1995.

[12] OMG. CORBA: Notification Service, V. 1.1. Specification, Object Management Group (OMG), Oct. 2004.

[13] OMG. Data Distribution Service for R-T Systems (DDS), V1.2. Spec., Object Management Group (OMG), Jan. 2007.

[14] S. Pallickara and G. Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In *Proc. of Middleware'03*, 2003.

[15] J. Pereira, F. Fabret, H. A. Jacobsen, F. Llirbat, and D. Shasha. WebFilter: A High-throughput XML-based Publish and Subscribe System. In *Proc. of VLDB'01*, 2001.

[16] J. Pereira, F. Fabret, F. Llirbat, R. P. Pietro, K. A. Ross, and D. Shasha. Publish/Subscribe on the Web at Extreme Speed. In *Proc. of VLDB'00*, 2000.

[17] P. R. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proc. of DEBS'02*, Vienna, Austria, July 2002.

[18] S. Rhea, B. Godfrey, B. Karp, et al. OpenDHT: A Public DHT Service and its Uses. In *Proc. of SIGCOMM'05*, 2005.

[19] I. Rose, R. Murty, P. Pietzuch, J. Ledlie, M. Roussopoulos, and M. Welsh. Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds. In *NSDI*, 2007.

[20] T. Sivaharan, G. S. Blair, and G. Coulson. GREEN: A Configurable and Re-configurable Publish-Subscribe Middleware for Pervasive Computing. In *OTM Conferences (1)*, volume 3760 of *LNCS*, 2005.

[21] Sun Microsystems. Java Message Service. Specification, Sun Microsystems, 2001. `http://java.sun.com/products/jms/`.

[22] P. Sutton, R. Arkins, and B. Segall. Supporting Disconnectedness - Transparent Inf. Delivery for Mobile and Invisible Comp. In *Proc. of CCGrid'01*, May 2001.

[23] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A Peer-to-Peer Approach to Content-Based Pub/Sub. In *Proc. of DEBS'03*, 2003.

[24] The Gryphon Team. Achieving Scalability and Throughput in a Pub/Sub System. Research report, IBM, Feb. 2004.

[25] W3C. XML Path Language Version 1.0 (W3C Recommendation), November 1999.

[26] W3C. XML Schema Part 0: Primer. W3C Recomm., World Wide Web Consortium, May 2001.

[27] W3C. Web Services Eventing (WS-Eventing). Specification, The World Wide Web Consortium (W3C), Aug. 2004.

[28] XEvents/XMessages: Application Events and Messaging Framework for Grid. Technical report, Extreme! Computing Lab, Indiana University, 2002.

[29] *XML-RPC Specification*. `http://www.xmlrpc.com/spec`.