# A Shared Global Event Propagation System to Enable Next Generation Distributed Services

Paul Brett, Rob Knauerhase, Mic Bowman, Robert Adams,
Aroon Nataraj, Jeff Sedayao, Michael Spindel
*Intel Labs, Intel Corporation*

**Abstract**

The construction of highly reliable planetary-scale distributed services in the unreliable Internet environment entails significant challenges. Our research focuses on the use of loose binding among service components as a means to deploy distributed services at scale. An event-based publish/subscribe messaging infrastructure is the principal means through which we implement loose binding. A unique property of the messaging infrastructure is that it is built on a collection of off-the-shelf instant messaging servers running on PlanetLab. Using this infrastructure we have successfully constructed long-running services (such as a PlanetLab node status service) with more than 2000 components.

## 1. Introduction

As many can attest, building a planetary-scale, distributed service differs substantially from building a traditional distributed service within a data center. A planetary-scale service must be architected for reliability even though it is built on highly unreliable components. It must adapt to a rapidly changing compute and communication environment. It must provide appropriate quality of service and performance for a globally distributed client community while also scaling to accommodate highly variable workloads.

Our research focuses on building planetary-scale services as a composition of many small, highly replicated component services. The components of a service are loosely bound together using an event-based, publish/subscribe messaging infrastructure called "Planetary-Scale Event Propagation and Routing" (PsEPR, pronounced "pepper"). Loose binding inverts the convention that a client initiates a connection to a service and waits for a response. Instead, in the spirit of Pandemonium [Selfridge59], a service publishes events to a communication channel with no *a priori* knowledge of the clients that are subscribed to or acting on the events. The communication channel isolates a service endpoint (defined by the channel abstraction) from a service instance (authenticated code that implements the specific service). Loose binding of components enables migration and replication of service instances, simplifies composition among services, and hides differences among multiple implementations of a particular service.

This paper focuses on the PsEPR messaging infrastructure, how it enables loose binding for planetary-scale services, and its implementation on an overlay network of commodity, off-the-shelf instant messaging servers deployed on PlanetLab [Peterson02].

## 2. Architecture of Planetary-Scale Services

### 2.1. Characteristics

While building and deploying services that monitor the health of PlanetLab nodes and audit traffic in PlanetLab [Trumpet03], we observed the following behaviors that differentiate planetary-scale services from traditional data-center services:

- Compute resources are unreliable.
- Network connections are unreliable.
- Bandwidth is variable.
- Latency is variable.
- Network routes are not guaranteed to be transitive.
- Network connections pass through multiple address spaces.
- Firewalls exist, and tunnel methodologies (if available) differ.
- Load (of the system overall, or of individual parts of the system) is highly variable.

These behaviors impact the design of planetary-scale services in many ways. Further compounding matters are the possibility of service instances having multiple versions, or being intermittently unavailable.

Loose binding among service components is an alternative approach to constructing distributed services that holds promise for addressing the problems listed above. We define loose binding as runtime discovery and utilization of independent (yet interoperating) components that can be assembled to provide a new, useful function, much as anticipated more than 30 years ago by Pandemonium.

Properties of loosely bound services include:

- Location (endpoint) independence – Services *rendezvous* with each other without needing to know the location of particular instances of the service. Further, many instances of a service may be connected to a communication channel simultaneously. Services may migrate between physical (and virtual) network locations without clients becoming aware of their migration.

- Service independence – Neither the publisher nor the subscriber of a service needs to know of the existence of the other, beyond whatever meta-information is (optionally) established by the application developers beforehand. Adding instances of a service (or alternative implementations of a service) is transparent to existing services and instances.
- Timing independence – The most basic form of ommunication among components of a service is asynchronous. Synchronous messages, remote procedure calls, transactions and other forms of interprocess communication are themselves implemented as service components.
- Protocol independence – Services assert interface specifications without the need for global agreement, making services essentially self-describing. Responsibility for interpreting the output of a service lies with the consumer of the service. Negotiation of interface specifications is yet another value-add service component.

Section 5 describes PLDB, a loosely-bound service that demonstrates these properties.

## 3. Planetary-scale Event Propagation and Routing

We built PsEPR, an event-based messaging abstraction with a form of publish/subscribe semantics, to support our research into (and our development of) loosely bound distributed services. Several design goals enable PsEPR to support scalability, adaptability, and reusability of services.

### 3.1. Events

A PsEPR event is an XML fragment that includes addressing information and a typed payload. Figure 1 shows an XML fragment for an event generated by a load average monitor running on PlanetLab. The monitor sends the event to the 'plab1.ee.ucla.edu' channel with a payload that identifies the current load average as 2.3.

```
<query xmlns='psepr:event'>
  <from>
    <service>psepr_source</service>
    <instance>psepr::source</instance>
  </from>
  <to>
    <channel>plab1.ee.ucla.edu</channel>
  </to>
  <event xmlns='psepper:payload:attribute'>
    <field>loadavg.15</field>
    <value>2.3</value>
  </event>
</query>
```

Figure 1 – a PsEPR event

### 3.2. Addressing

Services publish a PsEPR event on a "channel". Practically speaking, a channel is a name. Channel names are hierarchical, to enable prefix matches that enable subscription to a family of related channels. PsEPR does not enforce unique names for channels. To send an event to a specific destination, an address can optionally contain the name of a service (where "service" is an authenticated identifier) and the name of a specific instance of that service. The authenticated source service and instance are included with every event.

Channel-based addressing is the key to loose binding. A service can publish events to a channel with no information about the location of any subscribers (or even whether there are any subscribers) – or about the operations that will be performed on the events. A key implementation problem for PsEPR is to implement efficient routing for channels. One specific requirement is that the addition of new subscribers to a channel is completely transparent to existing publishers and subscribers on that channel. In this way, new service components can be added without affecting the behavior of the current collection of services.

### 3.3. Client API

The PsEPR client API provides operations for sending and receiving events:

- `Send`—Publish an event on a channel
- `Register`—Subscribe to events on a channel
- `Unregister`—Unsubscribe to events on a channel
- `Process`—Wait for an event to arrive

In PsEPR, the `Send` operation publishes events on a channel. The `Register` operation, used to subscribe to events on a channel, is implemented by sending a lease-request event to the 'psepr_registry' service on the channel. For example, a client interested in receiving events that describe the state of 'plab1.ee.ucla.edu' would send a lease request event to the registry service listening on the 'plab1.ee.ucla.edu' channel; Figure 2 shows the payload for this event.

```
<event xmlns='psepr:payload:lease'>
  <duration>180</duration>
  <identifier>psepr-1099501240</identifier>
  <type>register</type>
</event>
```

Figure 2 – a PsEPR event payload

In the current implementation of PsEPR, there is a single well-known registry service. However, since lease requests (and the corresponding responses) are just PsEPR events, it is possible to implement multiple registration services; we use this facility to experiment with alternative event routing algorithms – an example of loose binding extending PsEPR itself.

## 4. Composition of Loosely-Bound Services

We have used PsEPR to build and deploy loosely bound services on PlanetLab. Notable among these services is PLDB (the PlanetLab Database) which stores status information about PlanetLab nodes. Formerly a tightly-bound XMLRPC application, PLDB v2.0 is now a composition of many independent services. A set of monitors, a tuple-store service, and management components comprise the basic service. Loose binding enables the seamless addition of value-add components such as recommendation services, debugging services, and performance visualization services.

### 4.1. Monitors and Sensors

There are numerous PLDB monitors running on PlanetLab. Monitors running on each PlanetLab node observe properties like load average, currently installed packages, and kernel checksums. Properties are sent as PsEPR events to a channel that is named for the PlanetLab node described by the property. Currently more than 500 active monitors generate approximately 20,000 events daily.

### 4.2. Tuple-Store

A tuple-store service stores properties for PlanetLab nodes. Each instance of the service listens for events on a channel (recall that a channel is associated with each PlanetLab node) and saves the properties in an in-memory database. The tuple-store service also listens on the channel for queries and emits events satisfying the query. There are approximately 1500 instances of the tuple-store service currently running on PlanetLab ,each with a database of about 50 tuples.

### 4.3. Management Supervisor

Redundancy (for robustness, reliability, and high availability of the service) is implemented by replicating the tuple-store service across many diverse PlanetLab sites The management supervisor monitors health of the tuple-store service, dynamically starting and stopping instances to maintain availability. It also optimizes the distribution of tuple-store instances for network performance and client workload. Of course, the management supervisor service communicates with the tuple-store instances over PsEPR channels; the tuple-store service publishes events on a well-known, management channel to which the management supervisor is subscribed.

### 4.4. Value-Add Services

Since PLDB monitors and sensors use PsEPR channels for reporting information about PlanetLab nodes, other services besides the tuple-store can leverage this information. We have implemented several services that aggregate these data and rate the quality and usability of various PlanetLab nodes. The services publish recommendations on the channel that are therefore stored by the tuple-store service.

Another service we built was a PsEPR debugging service (both for debugging the system itself, and later for debugging services that use PsEPR). The debugging service simply listens on a set of channels and displays a timescale graph of traffic patterns, while also allowing inspection of event payloads themselves. None of the monitored components need even be aware of the debugger's existence.

Lastly, we built a PsEPR visualization service, both for demonstration of the system and as another debugging aid. This service subscribes to relevant channels, receives events from those channels, discards the payload in its entirety, and displays a connectivity graph of services, routers, and nodes.

## 5. PsEPR Implementation Details

PsEPR consists of two components: a message router and a registry service. The registry service receives lease-request events and maintains a map of subscriptions to service instances for PsEPR. In addition, the registry server informs the message router of clients who are interested in a particular channel.

The message router delivers events to interested local clients and implements the overlay network used to route events among PsEPR servers.

### 5.1. Overlay Routing

Client services can publish events to a channel with no information about the location of any subscribers. PsEPR constructs an overlay network to route events among subscribers on a channel. The use of overlay networks affords us great freedom to explore different mechanisms of routing within the system without affecting the operation of services on the edges.

#### 5.1.1. Client to PsEPR connections

PsEPR services may be composed of many distributed components. a number of factors including network attributes (throughput, latency, etc.) and application constraints (information location, diverse routing requirements, etc) determine each component's optimal entry point into the PsEPR network. We choose to make the client libraries primarily responsible for establishing and maintaining the most efficient connection to the PsEPR network. PsEPR message routers send route-advice events to clients indicating alternate entry points based on the router's information about the environment.

Client entry points are computed by offline analysis of the PlanetLab network, based on "all-pairs-pings" data collected for PlanetLab [Stribling04] augmented with local site information. This produces a small set of candidate PsEPR routers topologically proximate to a given application instance.

### 5.1.2. PsEPR to PsEPR connections

PsEPR message routers assume the responsibility of transparently and efficiently moving events to the correct endpoint(s).

Initially we implemented a simple n-to-n broadcast in which each PsEPR message router forwards events to a dynamic list of servers. This algorithm was easy to implement and allowed us to develop the service API, client libraries, and some sample services without waiting for a complete routing scheme. The broadcast method scaled surprisingly well to as many as 100 routers and more than a half-million events per day. However, as with any $n^2$ algorithm, the n-to-n broadcast quickly breaks down as either offered message load or number of PsEPR message routers increases.

We also implemented a routing algorithm based on a static minimal spanning tree. With this approach, when a PsEPR message router forwards an event to neighbors it knows are interested. A message router is "interested in" an event if it has PsEPR clients who are subscribed to the channel, or if it has neighbors who are "interested in" the event (recursively). Unlike the broadcast algorithm, the spanning tree approach accommodates non-transitive routes (e.g. connections from the commodity Internet to Internet2 nodes) and asymmetric routes through a NAT or firewall. Our spanning tree is generated using information on network latency taken from "all-pairs-pings" data.

We are currently investigating variations on this algorithm. Perhaps most importantly is dealing with partitioning of the tree. A PsEPR message router can cease responding for many reasons – software crash, machine crash, intermittent unreachability due to network congestion, etc. When this happens, a new routing tree must be generated. Our algorithms include tunable parameters for when and how often to regenerate the routing trees, but we have not yet thoroughly determined the settings that work best for our anticipated workloads.

Currently, we maintain just one spanning tree and send events to or through neighbors who are interested in the event. Because our channel abstraction allows us to know, dynamically, which listeners are on which channels, we are evaluating the use of multiple spanning trees, to more efficiently (in terms of latency, hop-count, etc.) route events, e.g. avoiding PsEPR servers which do not have direct clients interested in a set of events. One disadvantage of this scheme is that the overhead of generating, maintaining, and distributing multiple trees may exceed the gain from more efficient distribution; as our research progresses, we hope to be able to quantify this for optimal, normal, and pathological workloads.

## 5.2. Instant Messaging as Foundation

We elected to build our PsEPR prototype on top of an instant-messaging (IM) system. IM has long been popular for chatting among end users in home and business settings. Importantly, the ubiquity of IM infrastructure (existing servers, corporate firewall holes, etc.) has brought this technology to the attention of researchers [Knauerhase03] as a viable mechanism for new types of application-to-application communication.

Our system is based on the open-source Jabber [Jabber04] project. The choice to build on Jabber results in some obvious and subtle trade-offs. For example, using Jabber's XMPP (eXtensible Messaging and Presence Protocol) not only provides us with a foundation of code (client libraries, server infrastructure, etc.), but also the potential to capitalize on its notions of presence for determining the health and availability of individual PsEPR components.

Other benefits of building atop IM include the ability to tunnel through firewalls, network-address translation (NAT) systems, and so forth; as IM clients, we are able to fully exploit this connectivity to our advantage. By mapping our concepts into Jabber constructs (e.g. "JIDs" and "resources" play the role of PsEPR services and instances), we gain the benefit of object-level addressing, enabling our channel abstraction and our location-independence features (e.g. endpoints are no longer fixed to an IP address or a particular socket). In addition, we reuse Jabber's authentication mechanisms for PsEPR, freeing us from having to design and implement our own solution. Lastly, since XMPP is based on XML, it lends itself well to our goals of self-describing events and communication.

The PsEPR registry and message router components are implemented as IM clients. This offers several advantages. For example, we can interoperate with existing (non-PlanetLab) servers and we can run multiple versions of PsEPR components (with different addresses) connected to the same IM server. As our research matures, we plan to explore the performance and security benefits of integrating PsEPR directly into an IM server, or possibly implementing our own stripped-down “integrated PsEPR server” with only the IM functionality required by the PsEPR system.

There are, of course, disadvantages. PsEPR performance is limited both by the performance of the IM servers themselves, and by the IM client connection architecture. Additionally, XMPP’s heritage in XML implies a verbosity that incurs significant overhead in the encoding and transmission of small events.

The PsEPR client libraries hide details of the underlying IM system. From the distributed service’s point of view, one simply requests a connection, and begins communicating. Client libraries are available for Java, Perl, and Python.

### 5.3. Experimental Results

Through the course of development, we have been interested in measuring the performance of our system. Our results, while preliminary, are encouraging. Primarily, we wished to show that PsEPR was fast enough for general use (assuming a reasonable set of client components). Beyond that, we were concerned with scalability – both in terms of capacity, and in performance under global load. While our current implementation has not been optimized for performance, we wished to establish whether the system could provide a useful service on PlanetLab.

We created a test network of ten routers, with *n* event sources and *n* event sinks attached to each router. All event sinks subscribed to a single channel, to which the sources transmitted simple events. The use of a single channel results in a worst case configuration in which every event must be transmitted to every client (and therefore every router). Event sources transmit events into the system as quickly as possible, and the total time it takes for a stream of 100 events to arrive at each of the sinks is measured. An average response time can then be computed from the timing of the event arrivals.

| N (Sources/ Sinks) | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Routers | 10 | 10 | 10 | 10 |
| Total Events | 2,000 | 6,000 | 12,000 | 20,000 |
| Avg Response Time (s) | 0.77 | 0.94 | 2.22 | 3.45 |

**Table 1 - PsEPR Performance Measurements**

Initial results (Table 1) indicate that loads estimated to be an order of magnitude greater than that generated by PLDB 1.0 could be sustained on a single channel.

## 6. Related and Future Work

### 6.1 Other Publish/Subscribe message systems

XMPP Pub/Sub[Millard04] relates closely to PsEPR because it uses Jabber and XMPP as a foundation. A commercial implementation exists from PubSub.com (http://www.pubsub.com). This protocol does not specify routing between IM servers; the assumption is that all clients connect to one server.

The Web Services community has developed a family of specifications collectively called WS-Notification [Graham04]. These specifications define XML schema for standardized notification (event delivery) from “notification broker service providers”, as well as mechanisms by which publishers can deliver events to the provider. WS-Notification is integrated with other web service protocols (e.g. for security and addressing).

Many other publish/subscribe mechanisms exist, from the Java Message Service [Happner02] to TIBCO Corporation’s Rendezvous [Tibco04]. The Internet Indirection Infrastructure (i3) [Stoica02] has many similar ideas to PsEPR and can support late bindings for functions like load balancing and server selection. PsEPR differs from these mechanisms by abstracting message routing functionality and hiding it from publishers and subscribers. Similar to i3, our system allows use of different overlays and routing schemes as needs and research interests require. COBEA [Ma98] infrastructure attempts additional functions such as defining proxies and implementing alarms – PsEPR has a simpler interface and leaves these functions to higher-level services.

Most of the systems described in this section differ from PsEPR in that they assume just one message server, and they have a “connection-based” approach to subscription/registration. While i3 shares PsEPR’s ideas about object naming, it too does indirection based on connections, rather than on loose channel bindings.

### 6.2 Next steps

Work is currently underway to continue developing our debugging and visualization toolsets for PsEPR applications, which would further assist in development of applications based on our channel messaging abstraction.

For simplicity, we implement Jabber’s store of id/password with a centralized user database that feeds a copy at each PsEPR node. This central canonical store and synchronization will undoubtedly limit the future scalability of the system. Additionally, having a single central authority for authentication may present organizational limitations as more diverse users join the system.

The messaging infrastructure currently contains no mechanism to apply flow control to prevent a single application from flooding the network with events. Local policies can be established on a per router basis to prevent abuse of the network in the short term, but setting and enforcing appropriate global policies remains unsolved.

### 6.3 Future research

We plan to continue exploring what features the event-propagation system should provide, and how best to expose those in a general way to maximize utility to loosely bound distributed applications. Similarly, we have begun thought about how to "component-ize" PsEPR itself, for example, to formalize the layering of a low-level transport service supporting multiple (possibly competing) higher-level routing and event-management services.

We believe there are nontrivial questions in the determination and maintenance of an overlay network in an environment like PlanetLab. As nodes go up and down (or, often, suffer load that makes them *worse* than down, in that they are active but excruciatingly slow), we would like our service to be much more adaptive, responding more quickly and more intelligently.

Lastly, there also exist many interesting problems in finding the right mix between topologies for our overlay (e.g. bus, star, spanning tree, DHT, and combinations of the above) that provide acceptable reliability while not generating excessive traffic, again complicated by network load, CPU load, and failures.

## 7. Summary

Our research supports our assertions that a loose binding methodology supports scalability, adaptability, and robustness in distributed systems. In support of this methodology, we built a planetary-scale event propagation and routing system (PsEPR) that allows composition of services from components. In our system, services publish self-describing events to a communication channel rather than waiting to be contacted by clients.

To validate our system, we converted the PlanetLab Database (PLDB) and Trumpet sensors/monitors from a tightly-coupled monolithic service to a collection of loosely-bound components. We also developed other interesting services that interact and interoperate with the data generated by our sensors, providing useful functionality without requiring any change to the original components.

Our message system is constructed by setting up an overlay network of commercial-grade Instant Message servers. We validated the performance and scalability of the system both by running PLDB over PsEPR and by measuring performance and scalability under pathological (worst-case in traffic and in un-optimized code). We plan to continue developing the system, in exploration both of overlay routing and of techniques to develop loosely-bound distributed services.

## 8. References

[Graham04] Steve Graham et al., "Web Services Base Notification 1.2", http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf

[Happner02] Mark Happner et al., "Java Message Service 1.1", http://java.sun.com/products/jms/docs.html

[Jabber04] Jabber Software Foundation, "What is Jabber?", http://www.jabber.org/about/overview.php

[Knauerhase03] Rob Knauerhase and Krystof Zmudzinski, "Mobilized Instant Messaging", http://www.mobilizedsoftware.com/developers/showArticle.jhtml?articleId=17100270

[Ma98] Chaoying Ma and Jean Bacon, "COBEA: A CORBA-Based Event Architecture," 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe, 1998.

[Millard04] Peter Millard, "Jabber Enhancement Proposal JEP-0060", http://www.jabber.org/jeps/jep-0060.html

[OpenIM04] OpenIM project website, http://open-im.net/en/

[Peterson02] Larry Peterson, Tom Anderson, David Culler, Timothy Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet", in Proceedings of First ACM Workshop on Hot Topics in Networking (HotNets), October 2002

[Selfridge59] Oliver Selfridge, "Pandemonium: A Paradigm for Learning," Proceedings of Symposium on the Mechanization of Thought Processes, 511-29.

[Stribling04] Jeremy Stribling, "PlanetLab All Pairs Pings" data available from http://www.pdos.lcs.mit.edu/~strib/pl_app/

[Stoica04] I. Stoica, D. Adikins, S. Zhuang, S. Shenker, and S. Surana. "Internet Indirection Infrastructure", SIGCOMM, 2002.

[Tibco04] TIBCO Corp., "TIBCO Rendezvous", http://www.tibco.com/software/enterprise_backbone/rendezvous.jsp

[Trumpet03] Intel Corp., "Trumpet User Documentation", http://jabber.services.planet-lab.org/php/docs/users.php