

# PubSubBus: A scalable, message-oriented middleware for distributed, mobile applications

Debangsu Sengupta ([debangsu@cs.stanford.edu](mailto:debangsu@cs.stanford.edu))

## Abstract

PubSubBus is an example of a scalable, message oriented middleware targeted towards mobile applications. We present an initial prototype of the system, explain the design considerations and describe the key features of the implementation. Interesting design and implementation choices include smart proxies, asynchronous message passing, events and notifications, attribute oriented interfaces, and optimizations like notification coalescing, and message queues.

## 1.0 Introduction

Message-oriented middleware is software infrastructure that uses asynchronous message sending as a paradigm to simplify building distributed systems. Specifically, it provides an abstraction layer to the distributed application developer and encapsulates the details of interoperability across different networks and operating systems. Asynchronous message passing (in contrast to request-response) among client and server nodes is often used to propagate state across the system.<sup>1</sup>

PubSubBus provides a lightweight framework that utilizes asynchronous message passing and smart proxies to enable development of distributed systems that have the following properties: predictable latency, high availability, and eventual consistency. It provides the abstraction of a distributed service bus using smart proxies on application and worker nodes, along with a centralized directory service. By using a few high performing worker nodes, it is able to support the publish-subscribe pattern that enables application node to share state across the system in an asynchronous manner.

In section 2, we discuss related work. In section 3, we present the design and implementation of the PubSubBus system. In section 4, we discuss the pros and cons of the system. In section 5, we propose future improvements to the project, and conclude in section 6.

## 1.1 Use case

PubSubBus was motivated from development on the PrPI system<sup>2</sup>, a decentralized personal data management system that relies on personal servers that create and maintain a unified index of a person's data. Various applications, including ones on mobile devices use the PrPI system to share and access one's data across disparate sources. Building client-server applications on mobile clients that relied on RPC exposed well-known challenges.

Mobile Applications: Today's smartphones, such as the iPhone or the Motorola Droid (Android) have sufficient local computation/storage that native applications can benefit from. However, such applications often need to deal with unreliable, fluctuating wireless network characteristics when calling remote methods, such as via RPC<sup>3</sup>. RPC has well-known challenges in such an environment, namely that procedure calls have a large variance in their return time, ranging from several milliseconds in the case of cached calls to tens of seconds while waiting for a timeout. Further, the timeout may occur due to network or remote node issues.

We find that by relaxing consistency and network latency requirements in favor of available local computation/storage on the device, this framework can deliver useful properties to applications, such as predictable latency, and some offline capabilities.

Specifically, the common-case communication pattern is as follows. A number of application nodes (e.g. mobile/desktop/web application nodes) wish to communicate and exchange state with each other on a given topic/channel. Each application node contacts and registers with the directory service, which points them to a high-performance bus node that is assigned to the authoritative node for the information. The application node loads a proxy object from the bus node that reflects the current state of the topic. It subscribes to the topic and starts sending and receiving updates using messages. The bus node bears the responsibility to share the updates with all other application nodes registered to the topic.

## 2.0 Related Work

**PubSub protocols / patterns / systems** are encountered in operating systems, Networks (multicast), programming languages (e.g. Linda<sup>4</sup>), to today's web applications (e.g. PubSubHubBub<sup>5</sup>). Our focus is to study the known approaches and provide a messaging layer suitable for asynchronous/disconnected mobile communications

**Java Message Service (JMS)**<sup>6</sup> is a popular, heavy-weight enterprise class, message oriented middleware API. It relies on a (J2EE) application server that runs on the client nodes. It utilizes the Java Native and Directory Interface (JNDI) extension to perform discovery and lookup of directory information. The JMS API provides a standard approach for J2EE clients to create and exchange messages with each other. It provides loosely coupled, asynchronous communication, along with reliability. Applications either run in J2EE server's context, or communicate with it using network adapters. In contrast, our framework is lightweight and designed to run in-memory on a per application basis. There is no requirement for the complexity and performance penalties of communicating with a standalone J2EE process such as in JMS. A PubSubBus application writer merely needs to be aware of the "topic" paradigm

(although even this could be hidden using framework extensions) via which multiple instances can communicate. Our directory service is quite simple – we have implemented features that are necessary for our system. In future, a JNDI extension can be built, but was not found to be necessary.

**The Extensible Messaging and Presence Protocol (XMPP)**<sup>7</sup> is an open, XML based protocol geared towards near real-time instant messaging and presence applications. In addition to chat, it can also be used towards building Message Oriented Middleware. In contrast, our target application simply needs a messaging and routing layer. Since it did not need presence features, we built and used minimal features and did not require the complexity of a full XMPP implementation.

**SQLite**<sup>8</sup> is an relational database management system written in a C library that is linked in with application. It provides a light-weight persistent database that is popular among mobile applications. It is orthogonal to the messaging/routing service that we provide. Instead, it addresses the persistent storage aspect of MOM. Currently, our implementation is in-memory and has no persistence built-in. In order to implement durability on mobile devices, SQLite appears to be a good option.

**Remote Procedure Calls (RPC)** is a form of inter-process communication where the application program calls a function that executes in a different address space, usually over the network. Similar to our system, RPC encapsulated the communications layer details from the application programmer. RPC is popular in web services via the XML-RPC and JSON-RPC variants. Proxies are a complementary approach, since both RPC and Proxies can be configured to provide asynchronous and blocking semantics. In the common case, RPC's provide synchronous, blocking, request-response communication. An RPC call provides a local function call interface to the application layer. However, the call return time varies from milliseconds to tens of seconds in the case of a time-out. In such a situation, the application programmer needs to deal with the issue. Proxies are a more natural way of implement systems that require calls to return within predictable delay, and can tolerate stale data.

### 3.0 Design and Implementation

#### 3.1 Design considerations

The following are some of the design considerations while designing PubSubBus:

- **Predictable latency:** The system needs to provide predictable latency to the client application. We solve this using client proxies, in contrast to the common RPC use case where the function call may return several seconds later due to a remote timeout and the application needs to handle the case.

- **High availability / eventual consistency:** The bus needs to be highly available to the mobile applications. This is provided by a central directory service (which may be replicated as needed), along with several high performance bus nodes that service topics in a load-balanced manner. Bus Workers are load-balanced using sharding.

- **Throughput:** The bus workers and proxy bus workers need to efficiently exchange state information with each other. We solve this using message queues at each bus worker that provide optimization in the form of coalescing duplicate notifications. Our design emphasizes the latest state of topics.

- **Record – playback / archival:** The system provides an ability for Proxy Topics to request archived state of the topic. Topics record a limited archive of past Topic State Msgs, which are delivered on request. The Proxy can play back the messages in scenarios where the ordering of updates is important.

- **Tradeoff 1:** The system trades network latency in favor of local computation and storage. The observation is that mobile applications have sufficient resources that can be leveraged to provide a better experience to applications. This is done by creating and performing actions on local state whenever possible.

- **Tradeoff 2:** The system trades strict consistency in favor of predictable latency and availability. It is an example of a BASE<sup>9</sup> design, in contrast with ACID principles. By relaxing the consistency requirements, we are able to build a system that provides predictable latency via local method calls, and high availability even during network disconnects. The eventual consistency mechanism is provided using asynchronous message updates.

#### 3.1 Design

The service bus abstraction is provided by a combination of a central directory service, application nodes (proxy bus nodes), and high performance/authoritative bus nodes.

**Bus:** The service bus abstraction is provided by a central directory service (Dir Svc), and bus nodes (Bus Workers). The Dir Svc is the only necessarily centralized/distinguished component in the design. It serves as registrar and provides discovery and lookup services to all other components in the system, such as the proxy directory service, proxy bus workers, and bus workers.

**Proxy Bus:** The proxy bus node represents the application node, and typically runs in the context of a client application. It consists of a proxy directory service (Proxy Dir Svc), and a proxy Bus Worker. It provides a local interface to the application and encapsulates the management of distributed state and communication layer. The Proxy Dir Svc connects to the central directory service

and performs registration, discovery and lookup services on behalf of the application.

**Bus Worker / Proxy Bus Worker:** The Bus Worker has several functions such as establishing and managing connections, message processing, message routing, and managing topics. A Bus Worker is the authoritative source of a topic. The Proxy Bus Worker is the client instance of the BusWorker, which is loaded on the (mobile) application node. It communicates with the Bus Worker to create, send and receive updates for a topic. While the Bus Worker and Proxy Bus Worker have several identical characteristics, they differ in the following important ways.

- A Bus Worker maintains the authoritative version of a topic, while a proxy bus worker operates on a cached version.
- Routing rules: A bus worker communicates with all proxy bus workers subscribed to a topic. A proxy bus worker only communicates with the bus worker for the topic.
- Hardware and Network characteristics: Due to the communication pattern described above, a bus worker needs to be addressable, and able to service higher processing, storage, and network requirements.

**Load balancing:** Bus Workers allow the system to provide horizontal scaling by using the sharding concept and partitioning topics. By having  $m$  Bus Workers, serving  $n$  topics each, the system has a capacity of  $m * n$  topics. Additional capacity can be obtained by adding more bus workers to the system. This is an example of horizontal scaling.

**Topic / Proxy Topic:** In PubSubBus, application nodes exchange state info over a topic abstraction. A topic is identified by a unique ID, and has associated data, and metadata (producer and consumer lists, authoritative bus worker etc). The authoritative version of a topic is maintained by a bus worker, while cached proxy topics are instantiated on proxy bus worker nodes. Applications read/write messages to a proxy topic, which sends updates to the authoritative topic via messages. Topics can be long-lived or short-lived according to application needs, and can provide optional archiving and persistence capabilities.

### 3.2 PubSubBus Scenario

Let us describe the key features of PubSubBus using a detailed use case.

- Bus bootstrap:** The central Dir Svc starts and starts accepting connections.
- Bus Worker registration:**  $1...m$  BusWorker nodes start and register with the Dir Svc. They provide information about the topics that they are currently serving. Each bus worker serves up to  $1...n$  topics thereby providing a system

capacity of  $m * n$  topics.

- Proxy Bus registration:** An application instantiates a ProxyBus node, which consists of a Proxy Dir Svc and a Proxy Bus Worker. The Proxy Dir Svc establishes connection with the central Dir Svc for discovery and lookup of BusWorkers.

- Create/join topic:** On application request, the Proxy Bus Worker creates a Proxy Topic named "topic1." The Proxy Dir Svc requests the Dir Svc for the topic's authoritative bus worker. The Dir Svc provisions a Bus Worker for the topic and returns it to the Proxy Bus Worker. If the bus worker already exists for the topic, the Dir Svc simply returns its URI.

- Load balancing:** The Dir Svc has a collection of available Bus Workers that have registered. It provisions one of them using round-robin or more sophisticated load balancing techniques.

- Catch up with topic / Load proxy object:** The Proxy Topic exchanges Topic State Msg's with the remote Topic via the bus workers to load the latest state of the topic. After initial loading cost, the application operates on the cached/proxy version of the topic.

- Subscribe to topic:** Proxy Topic adds self to the topic's Producers/Consumers lists. Topic State Msg's are used by the Proxy Bus Worker to communicate the state change to the Bus Worker, which updates its authoritative topic object.

- Pub-Sub:** The Proxy Bus Worker sends a single message to the Bus Worker containing state change information. The bus worker updates itself and forwards the state information to all other consumers of the topic. This establishes the publish-subscribe pattern and is an example of multicast via unicast messages. The consumers (other Proxy Bus Workers) update their Proxy Topics on receipt of the message.

### 3.3 Implementation

PubSubBus is written in Java and comprises approximately 3000 SLOC including test modules. This section implements choices and also describes the major components.

**Attribute-oriented interfaces:** PubSubBus makes use of attribute oriented public interfaces. This corresponds to a RESTful interface design<sup>10</sup>, and simplifies interface maintenance. Both simple as well as composite attributes are used. Composite attributes provide better abstraction, and more flexibility than inheritance and correspond to the adapter design pattern<sup>11</sup>.

**Events and notifications:** They are used at multiple levels in the system. At a high level, asynchronous messages parallel a system receiving asynchronous notifications of

interesting events to which state updates are performed. On a code level, the (Proxy) Topic objects update themselves on receiving notification from the (Proxy) BusWorker about new Topic State Msgs. Finally, the application registers for notifications to specific attributes changing (such as Topic Data or Topic Subscribers) using the observer pattern<sup>12</sup>, a form of event-notifications.

**Smart Proxy:** Each application instantiates a Proxy Bus object, which promises of a Proxy Dir Svc and a Proxy Bus Worker objects. A Proxy Bus Worker manages many Proxy Topics. The Proxy Topic objects are an example of smart, or stateful, proxies, and communicate with their authoritative counterparts on the network using the Bus Worker.

A smart proxy object is loaded at startup from the remote location. This one-time startup cost is amortized over subsequent accesses, which are local to the application.

**Message based routing:** We use an asynchronous message passing paradigm to share state updates across application nodes using the bus nodes for efficiency. This is done via Topic State Msg objects that are sent across the network. The Topic and Proxy Topic objects update their state from these msg objects. This is in contrast with blocking request-response paradigm common in RPC.

**Routing semantics:** Publish – subscribe is one of many message routing modes present in message oriented middleware. Examples include (synchronous/asynchronous) request-response, flooding / broadcast, multicast. We focussed on publish-subscribe as that best fit the communication patterns for ad-hoc distributed mobile applications.

While publish-subscribe is a good choice for sharing updates among topics, there are situations where a unicast request-response model is better suited. In such a case, the sender and destination information in the msg's can be used to support unicast replies. For example, when a Proxy Topic requests update to a topic's state, the Bus Worker does not send the messages to all existing servers. Other examples include Bus Worker registration, request for archives or missing messages.

**Marshalling, custom deserializer to support sub classes:** In order to communicate across the network, messages need to be marshalled and un-marshalled. We chose to serialize the messages to XML, and later optimized it to JSON<sup>13</sup> objects in order achieve more compaction. Farther optimization is possible using binary serialization formats such as Google's Protocol Buffers or Facebook's Thrift. Compression such as gzip, are another approach to reduce network data, while incurring processing cost at endpoints. All msg types in the system (e.g. Topic State Msg) subclass the Msg class. Regular Java deserializing loses subclass information. We use Google's GSON library for marshal/unmarshal. To handle subclasses, we implemented a custom deserializer module that uses Java introspection

and/or a MsgType field hint to deserialize into the concrete class type.

Finally, a NetMsg object contains information about message tpe, length, sender, receiver, and message state (pre and post marshalling).

**Bus Worker / Proxy Bus Worker:** These modules encapsulate most of the complexity in the PubSubBus system. The Bus Worker registers with the Dir Svc and serves as the authoritative source of a topic. The Proxy Bus Worker object is a client copy of the Bus Worker. In our current implementation, each ProxyBus (application node), has 1 Proxy Dir Svc, and 1 Proxy Bus Worker. The Proxy Bus Worker communicates with the Proxy Dir Svc for lookup, along with up to n Bus Workers that serve the n topics it is interested in.

The Bus Worker functionality is delivered by the following components: Msg Processor, Msg Router, Topics Manager, and Transport Manager (todo).

**a. Msg Processor:** Msg Processor modules reside inside a (Proxy) BusWorker module. Each (Proxy) BusWorker object maintains incoming and outgoing message queues, which are serviced by a thread each. Incoming, serialized messages are placed in a (Proxy) BusWorker's In message queue. The In-MsgProcessor thread deserializes the msg before passing to the correct topic via the Topics Manager.

Post processing, outgoing messages are placed in the (Proxy)BusWorker's out message queue. The Out-MsgProcessor thread retrieves the message, serializes, and sends to the recipient(s) via the Msg Router.

The queues are of fixed size. Once they become full, then the choices are to either drop new messages, or evict old messages. Our assumption is that applications are recency-sensitive and therefore, we choose to evict older messages using LRU. The eviction rule can be provided as a configurable parameter. The implementation uses a Linked Hash Map to implement the queue, with each message's hash code serving as a key into the map.

**b. Msg Router:** The msg router module sends the message to the appropriate recipient according to the routing scheme (pub-sub). In the case of a BusWorker, it sends Topic State Msg objects to all topic consumers except for the sender. In the case of a ProxyBusWorker, it sends to the super consumer (BusWorker) for the topic.

All other messages in the system are control messages. There fore, unicast request-response is appropriate e.g. registration. (Note: To be implemented) The current implementation runs within a single process and uses TopicStateMsgs for all topic related updates. However, it directly uses the objects for registration using a layer of indirection to simulate the network. It uses an object ID -> object decoupled mapping so that it is designed for the

forthcoming Transport Manager.

**c. Topics Manager:** The Topics Manager maintains a collection of topics served by the BusWorker. The ProxyBusWorker's Topics Manager maintains a collection of Proxy Topics, which are client/cached versions of the topic, sharing the same Topic ID. The Proxy Topic maintains a mapping to the authoritative Topic object's ID/URI.

**d. Transport Manager:** It is inefficient for the (Proxy) Bus Worker to incur TCP connection set up overhead in order to send frequent state update messages. Therefore, we need to maintain a set of persistent HTTP connections among the Bus Worker and Proxy Bus Worker objects. This will be implemented in the future. Initial prototyping shows that xSockets and asyncWeb frameworks may be a good match, since they provide asynchronous, non-blocking IO via Java's NIO library.

**e. Load balancing:** Bus Workers are load-balanced by the Dir Svc using simple round-robin bus allocation for topics.

**f. Routing Super Consumer:** The ProxyBusWorker tracks the BusWorker for a given topic, which acts as the super consumer for the proxy topic. In order to consume network resources, all TopicStateMsg's for a given topic are exchanged between the Proxy Bus Worker and its Bus Worker as unicast messages. Each ProxyTopic maintains a list of the topic's consumers - direct communication among ProxyBus nodes is possible.

**Topic / Proxy Topic:** PubSubBus applications communicate using Proxy Topics and its authoritative Topic which share a Topic ID. Each (Proxy) Topic object consists of a unique ID, data, metadata (producers and consumers lists, along with timestamp information). It also includes limited history.

**a. Record – playback:** Each Topic object maintains a limited history of past TopicStateMsg objects, which are sent to ProxyTopic objects on request. The TopicStateMsg objects are time stamped and thus can be played back by the Proxy Topic to "catch up" and update itself.

**b. Fixed size:** The archive queue is fixed size. On becoming full, messages are evicted per LRU scheme. The implementation internally uses a LinkedHashMap using unmarshalled message hash codes as keys.

**c. Event-notifications:** The eventing paradigm is used by Topic and ProxyTopic objects to update themselves. After msg processing (unmarshalling), the Topic or ProxyTopic object is notified of the new message so that it can update itself and possibly generate new messages in response.

On the client application side, each ProxyTopic object acts as the notifier. The application registers itself and a callback is used to notify the application of an update e.g. topic data/

subscriber change.

**d. Notification coalesce:** The archive queue is of fixed size. We have implemented the ability to coalesce multiple identical messages. For example, a Topic may receive multiple messages with the same data. In the topic's history, this can be coalesced to the same message. During playback, time based order needs to be preserved.

The notification coalesce is a better fit for the Msg Processor queues because we assume the common case of only tracking the latest state of a Topic. Thus, multiple Topic Data updates can be coalesced into a single message on the processing queue. However, the Msg Processor incoming queue currently contains serialized msgs, which reduces the applicability of the system to simply removing identical messages, which are rare because their time stamps typically differ. It is more interesting to coalesce messages when their significant fields are common. After a future refactoring, one can use coalescing in the incoming queues. On the outbound queue, coalesce is not needed since the ProxyTopic will typically not output duplicate, identical messages.

### 3.4 Code Snapshot

SLOC	Directory	SLOC-by-Language (Sorted)
1034	Common	Java=1034
816	Client	Java=816
601	Test	Java=601
537	Server	Java=537
24	Userapp	Java=24
Total: 3012		Java=3012

Figure: PubSubBus code snapshot

### 3.5 Traces

1. Pub-Sub: 1 BusWorker (Topic1), 2 ProxyBusWorkers (PTopic1, PTopic2( exchange messages over a single topic ("topic1")). Applications sit on top of the ProxyBusWorkers.

16:23:31,202 INFO BusWorkerImpl:268 - Started thread 430693591.InMQThread for Queue:IN

16:23:31,206 INFO BusWorkerImpl:268 - Started thread 430693591.OutMQThread for Queue:OUT

**16:23:31,207 INFO DirSvcImpl:136 - BusWorkerAdd: Added busWorker with Id: -1155484576. New size: 1**

16:23:31,210 INFO DirSvcImpl:246 -

ProxyBusWorkerAdd: Added pBusWorker with Id: -1676573746. New size: 1

**16:23:31,211 INFO ProxyBusWorkerImpl:54 - Started ProxyBusWorkerImpl with Id: -1676573746**

16:23:31,217 INFO ProxyBusWorkerImpl:328 - Started thread -1676573746.PInMQThread for Queue:IN  
16:23:31,220 INFO ProxyBusWorkerImpl:328 - Started thread -1676573746.POutMQThread for Queue:OUT  
**16:23:31,220 INFO SingleProcTestsJunit:353 - Ptopic1 subscribes for topic: topic1**

16:23:31,233 INFO DirSvcImpl:246 - ProxyBusWorkerAdd: Added pBusWorker with Id: -1333558320. New size: 2

**16:23:31,234 INFO ProxyBusWorkerImpl:54 - Started ProxyBusWorkerImpl with Id: -1333558320**

16:23:31,234 INFO ProxyBusWorkerImpl:328 - Started thread -1333558320.PInMQThread for Queue:IN

**16:23:31,235 INFO SingleProcTestsJunit:360 - Ptopic2 subscribes for topic: topic1**

16:23:31,235 INFO ProxyBusWorkerImpl:328 - Started thread -1333558320.POutMQThread for Queue:OUT

16:23:31,378 INFO ProxyBusWorkerImpl:307 - transportSend: Sending msg from :-1333558320 to BusWorker: -1155484576

16:23:31,395 INFO ProxyBusWorkerImpl:307 - transportSend: Sending msg from :-1333558320 to BusWorker: -1155484576

16:23:32,223 INFO ProxyBusWorkerImpl:307 - transportSend: Sending msg from :-1676573746 to BusWorker: -1155484576

16:23:32,226 INFO ProxyBusWorkerImpl:307 - transportSend: Sending msg from :-1676573746 to BusWorker: -1155484576

16:23:33,212 INFO BusWorkerImpl:248 - transportSend: Sending msg from :-1333558320 to ProxyBusWorker: -1676573746

16:23:33,217 INFO BusWorkerImpl:248 - transportSend: Sending msg from :-1333558320 to ProxyBusWorker: -1676573746

16:23:33,220 INFO BusWorkerImpl:248 - transportSend: Sending msg from :-1676573746 to ProxyBusWorker: -1333558320

**16:23:33,223 INFO TestProxyTopicNotiffee:68 - Proxy Topic changed:**

ProxyBusWorker: -1676573746,

TopicId: topic1,

TopicState Msg:

Id: topic1,

Data: null,

Producers:

-1333558320,

Consumers:

-1333558320, -1676573746,

Origin: REMOTE,

Sender: -1333558320,

Timestamp: Thu Dec 10 16:23:32 PST 2009,

Hashcode: -868033807,

16:23:33,224 INFO BusWorkerImpl:248 - transportSend: Sending msg from :-1676573746 to ProxyBusWorker: -1333558320

**16:23:33,226 INFO TestProxyTopicNotiffee:68 - Proxy Topic changed:**

ProxyBusWorker: -1676573746,

TopicId: topic1,

TopicState Msg

Id: topic1,

Data: null,

Producers:

-1333558320,

Consumers:

-1333558320, -1676573746,

Origin: REMOTE,

Sender: -1333558320,

Timestamp: Thu Dec 10 16:23:31 PST 2009,

Hashcode: -868033807,

**16:23:33,239 INFO TestProxyTopicNotiffee:68 - Proxy Topic changed:**

ProxyBusWorker: -1333558320,

TopicId: topic1,

TopicState Msg:

Id: topic1,

Data: null,

Producers:

-1676573746,

Consumers:

-1333558320, -1676573746,

Origin: REMOTE,

Sender: -1676573746,

Timestamp: Thu Dec 10 16:23:32 PST 2009,

Hashcode: -868033807,

**16:23:33,242 INFO TestProxyTopicNotiffee:68 - Proxy Topic changed:**

ProxyBusWorker: -1333558320,

TopicId: topic1,

TopicState Msg:

Id: topic1,

Data: null,

Producers:

-1676573746,

Consumers:

-1333558320, -1676573746,

Origin: REMOTE,

Sender: -1676573746,

Timestamp: Thu Dec 10 16:23:31 PST 2009,

Hashcode: -868033807,

**16:23:35,235 INFO SingleProcTestsJunit:370 - Ptopic2 enters topic data: ptopic2: modified topicdata c**

16:23:35,399 INFO ProxyBusWorkerImpl:307 -

transportSend: Sending msg from :-1333558320 to

BusWorker: -1155484576

16:23:37,227 INFO BusWorkerImpl:248 - transportSend:

Sending msg from :-1333558320 to ProxyBusWorker:

-1676573746

**16:23:38,234 INFO TestProxyTopicNotiffee:68 - Proxy Topic changed:**

ProxyBusWorker: -1676573746,

TopicId: topic1,

TopicState Msg:

Id: topic1,  
Data: ptopic2: modified topicdata c,  
Producers:  
-1333558320, -1676573746,  
Consumers:  
-1333558320, -1676573746  
Origin: REMOTE,  
Sender: -1333558320,  
Timestamp: Thu Dec 10 16:23:31 PST 2009,  
Hashcode: 1962186989,

**16:23:40,236 INFO SingleProcTestsJunit:379 -  
test\_TopicPubSubMsgTwoProxiesCallback: Final states:**

16:23:40,236 INFO SingleProcTestsJunit:380 -  
test\_TopicPubSubMsgTwoProxiesCallback: **PBusWorker:**  
**-1676573746** PTopic1 state: TopicState Msg:  
Id: topic1,  
Data: ptopic2: modified topicdata c,  
Producers:  
-1676573746, -1333558320,  
Consumers:  
-1676573746, -1333558320,  
Origin: LOCAL,  
Sender: -1676573746,  
Timestamp: Thu Dec 10 16:23:31 PST 2009,  
Hashcode: 1962186989,

16:23:40,237 INFO SingleProcTestsJunit:384 -  
test\_TopicPubSubMsgTwoProxiesCallback: **PBusWorker:**  
**-1333558320** PTopic2 state: TopicState Msg:  
Id: topic1,  
Data: ptopic2: modified topicdata c,  
Producers:  
-1333558320, -1676573746,  
Consumers:  
-1333558320, -1676573746,  
Origin: LOCAL,  
Sender: -1333558320,  
Timestamp: Thu Dec 10 16:23:31 PST 2009,  
Hashcode: 1962186989,

16:23:40,237 INFO SingleProcTestsJunit:389 -  
test\_TopicPubSubMsgTwoProxiesCallback: **BusWorker:**  
**-1155484576** BusWorker state: TopicState Msg:  
Id: topic1,  
Data: ptopic2: modified topicdata c,  
Producers:  
-1333558320, -1676573746,  
Consumers:  
-1333558320, -1676573746,  
Origin: LOCAL,  
Timestamp: Thu Dec 10 16:23:31 PST 2009,  
Hashcode: 1962186989,

16:23:40,243 INFO ProxyBusWorkerImpl:197 - Shutting  
down thread -1333558320.PInMQThread for Queue:IN  
16:23:40,252 INFO BusWorkerImpl:128 - Shutting down  
thread 430693591.InMQThread for Queue:IN

16:23:40,399 INFO ProxyBusWorkerImpl:197 - Shutting  
down thread -1333558320.POutMQThread for Queue:OUT  
16:23:41,228 INFO ProxyBusWorkerImpl:197 - Shutting  
down thread -1676573746.POutMQThread for Queue:OUT  
16:23:41,233 INFO BusWorkerImpl:128 - Shutting down  
thread 430693591.OutMQThread for Queue:OUT  
16:23:41,235 INFO ProxyBusWorkerImpl:197 - Shutting  
down thread -1676573746.PInMQThread for Queue:IN

**2. Notification Coalesce in Topic Archive.** The Topic  
receives 3 notifications - 2 identical and 1 different. It  
coalesces the identical msg and overwrites using a linked  
hash map.

19:57:32,916 INFO SingleProcTests:453 -  
test\_TopicNotificationCoalesce  
19:57:32,921 DEBUG DirSvcImpl:57 - Instantiating DirSvc  
19:57:32,922 DEBUG BusWorkerImpl:40 - Instantiating  
BusWorkerImpl  
19:57:32,927 INFO DirSvcImpl:136 - BusWorkerAdd:  
Added busWorker with Id: -1155484576. New size: 1  
19:57:32,929 DEBUG TopicImpl:39 - <init>: TopicId:  
topic1

**19:57:32,944 DEBUG TopicImpl:208 -  
TopicDataArchiveAdd: TopicState Msg: Id: topic1,  
Data: topicdata 1, Producers: Consumers: Origin:  
NONE, Timestamp: Wed Dec 09 19:57:32 PST 2009,  
Hashcode: -1105714599,  
19:57:32,944 DEBUG FixedSizeLRUMapQueue:62 -  
enqueue: Enqueued. New size: 1**

**19:57:32,945 DEBUG TopicImpl:208 -  
TopicDataArchiveAdd: TopicState Msg: Id: topic1,  
Data: new topicdata 2, Producers: Consumers: Origin:  
NONE, Timestamp: Wed Dec 09 19:57:32 PST 2009,  
Hashcode: 1274873818,  
19:57:32,946 DEBUG FixedSizeLRUMapQueue:62 -  
enqueue: Enqueued. New size: 2**

**19:57:32,947 DEBUG TopicImpl:208 -  
TopicDataArchiveAdd: TopicState Msg: Id: topic1, Data:  
topicdata 1, Producers: Consumers: Origin: NONE,  
Timestamp: Wed Dec 09 19:57:32 PST 2009, Hashcode:  
-1105714599,  
19:57:32,947 DEBUG FixedSizeLRUMapQueue:62 -  
enqueue: Enqueued. New size: 2**

19:57:32,948 INFO SingleProcTests:533 - Archive list:  
19:57:32,948 INFO SingleProcTests:537 - TopicState Msg:  
Id: topic1, Data: topicdata 1, Producers: Consumers: Origin:  
NONE, Timestamp: Wed Dec 09 19:57:32 PST 2009,  
**Hashcode: -1105714599,**

19:57:32,949 INFO SingleProcTests:537 - TopicState Msg:  
Id: topic1, Data: new topicdata 2, Producers: Consumers:  
Origin: NONE, Timestamp: Wed Dec 09 19:57:32 PST  
2009, **Hashcode: 1274873818**

## 4.0 Discussion

Messaging Oriented Middleware includes message routing, persistence, and message transformation. Pub Sub Bus focusses on message routing. While it offers in-memory archival, there is no durability in the system. For this, SQLite on mobile applications, and a SQL or key-value based store on Bus Workers appear to be a good choice. Message transformation is useful for enterprise service buses with heterogeneous senders and receivers. Here, we assume that all senders and receivers share state via common Topic State Msg objects serialized into JSON format.

We have implemented an initial prototype that uses asynchronous messages and message queues to send out tens of messages per second. It performs adequately on our test system – a laptop running all the components described. We have already seen performance improvements from the first round of optimizations – namely JSON encoding of messages over XML, notification coalescing in message and archive queues. However, meaningful performance evaluation and tuning will be carried out after the integration of a Transport Manager (see Future Work).

In order for the system to be usable by mobile applications, one needs to solve the addressability issue – mobile devices often operate behind private data networks, and therefore unreachable by external connections. They will need to maintain persistent outbound connections through their firewalls. Or, for direct connections, STUN/TURN protocols can be used to navigate firewalls and NAT devices.

## 5.0 Future Work

**Scalability and optimization improvements:** There are opportunities to make PubSubBus more scalable at multiple levels.

a. **Directory Service** is a centralized component and must serve a large number of short requests. The main writers are the (Proxy) Bus Workers registering and the Bus Workers responding to topic allocation requests, while the main read requests come from Proxy Bus Workers' lookup requests for Bus Workers. Some options include replicating the Directory Service, or splitting it into Read and Write servers.

b. **Application level Multicast** can be made more scalable to support a much larger number of hosts. By using a multi-level multicast tree, a Bus Worker can support many more Proxy Bus Workers. The separation of read and write servers apply here as well. The ProxyBusWorker objects can write to a small number of Write servers, which will propagate updates to Read Servers. The Read Servers can be arranged as a multicast tree as mentioned above to support a larger fanout. The Write Servers will be able to support higher throughput, while the Read Servers will support high availability at the cost of slightly increased

latency.

**BusWorker Decentralization:** In the current implementation, the Bus Worker is a distinguished node. In reality, only the Dir Svc needs to be centralized for convenience. However, as described earlier, from the implementation perspective, the BusWorker and ProxyBusWorker modules are essentially the same, differing in roles (specifically routing roles). A future refactoring will make this remove the distinguished role of the BusWorker module.

**BusWorker Selection:** Currently, Bus Workers register with the Dir Svc, which allocates a Bus Worker for a given topic. Once Bus Workers are no longer distinguished nodes, there emerge more possibilities for allocating Bus Workers. For example, the first (Proxy) Bus Worker to start the topic can attain the authoritative BusWorker for it. Once the topic gains subscribers, the Proxy Bus Workers can execute a leader elect protocol to elect a new authoritative source based on criteria such as bandwidth, uptime etc. Such a protocol will allow high availability in the event of Bus Worker failure.

**Load balancing:** There is opportunity to improve the load balancing from the simple round-robin mechanism that the Dir Svc picks. For example, a new Topic request can be served by a Bus Worker with the lowest number of subscriber (open connections), assuming near uniformity in Bus Worker processor and bandwidth capabilities.

**Transport Manager:** As mentioned above, the current prototype works within a single process. The next logical step is to implement a transport manager that maintains persistent HTTP connections between Bus Worker and Proxy Bus Workers.

**Streaming Support:** The current system operates on objects updating state. It would be interesting to extend it to support byte streams, and examine suitability for streaming applications.

**Addressability:** The system does not address the issues of addressability; most mobile applications connect to the data networks from behind firewalls and private networks. Therefore, they are not addressable. By using the transport manager, the Proxy Bus Workers will open and maintain persistent HTTP connections to the Bus Workers, which will be able to contact them. Direct inbound contact among Proxy Bus Workers will require the use of STUN/TURN protocol and servers.

**Authentication / Authorization:** The system allows nodes to self declare addresses and publish. Basic authentication / authorization will be necessary to prevent system wide abuse by rogue nodes.

**Performance studies:** While we have built a usable prototype, meaningful performance studies and tuning will

be possible after the Transport Manager is integrated into the system. At that point, we can get meaningful information about the number of connections, traffic characteristics, frequency of nodes entering and existing the system and so on.

**Integration with mobile client application:** A useful study will compare a PubSubBus enabled mobile application with a standard one relying on XML-RPC/JSON-RPC and evaluate the applicability of our model in a real world test.

## **6.0 Conclusion**

We have successfully implemented a functional pub-sub bus, which can function as a message-oriented middleware for mobile applications. This can be a light-weight alternative to today's existing options. With a few additional components like the Transport Manager, the framework can be usable in mobile/desktop distributed applications that can benefit from a lightweight, asynchronous messaging system with useful latency, and availability properties.

## **7.0 Acknowledgements**

Many thanks go to Professors Peter Danzig, and Pei Cao for their guidance in this project, Professor Monica Lam for feedback and encouragement, Professor David Cheriton for introducing me to many of the topics from CS249a/CS244b material, and Michael Chan for his help, troubleshooting and feedback on the design.

## 8.0 References

- 1 David Chappell, Enterprise Service Bus, O'Reilly Media, Inc., 2004
- 2 Seong et al, The Architecture and Implementation of a Decentralized Social Networking Platform, Seong et al. October 2009.
- 3 A. L. Ananda , B. H. Tay , E. K. Koh, A survey of asynchronous remote procedure calls, ACM SIGOPS Operating Systems Review, v.26 n.2, p.92-109, April 1992
- 4 Nicholas Carriero , David Gelernter, Linda in context, Communications of the ACM, v.32 n.4, p.444-458, April 1989
- 5 PubSubHubBub: <http://code.google.com/p/pubsubhubbub/>
- 6 Richard Monson-Haefel , David Chappell, Java Message Service, O'Reilly & Associates, Inc., Sebastopol, CA, 2000
- 7 Saint-Andre, P., Smith, K., and Tronon, R. 2009 Xmpp: the Definitive Guide Building Real-Time Applications with Jabber Technologies. O'Reilly Media, Inc.
- 8 Michael Owens, Embedding an SQL database with SQLite, Linux Journal, v.2003 n.110, p.2, June 2003
- 9 Pritchett, D. 2008. BASE: An Acid Alternative. *Queue* 6, 3 (May. 2008), 48-55. DOI=<http://doi.acm.org/10.1145/1394127.1394128>
- 10 Pautasso, C., Zimmermann, O., and Leymann, F. 2008. Restful web services vs. "big" web services: making the right architectural decision. In *Proceeding of the 17th international Conference on World Wide Web* (Beijing, China, April 21 - 25, 2008). WWW '08. ACM, New York, NY, 805-814. DOI= <http://doi.acm.org/10.1145/1367497.1367606>
- 11 MacDonald, S. 1996. Design patterns in enterprise. In Proceedings of the 1996 Conference of the Centre For Advanced Studies on Collaborative Research (Toronto, Ontario, Canada, November 12 - 14, 1996)
- 12 Batov, V. 2001. Callbacks made easy with the observer/mediator design patterns. *C/C++ Users J.* 19, 2 (Feb. 2001), 38-45.
- 13 Michael Mahemoff, Ajax Design Patterns, O'Reilly Media, Inc., 2006
- 14 Attribute oriented interfaces: <http://www.stanford.edu/class/cs249a/coursereader/ch2.pdf>
- 15 Event-notifications: <http://www.stanford.edu/class/cs249a/coursereader/ch3.pdf>
- 16 Stateful Proxy: <http://www.stanford.edu/class/cs244b/coursereader/ch3.pdf>
- 17 Yongqiang Huang , Hector Garcia-Molina, Publish/subscribe in a mobile environment, *Wireless Networks*, v.10 n.6, p.643-652, November 2004
- 18 Baldoni R., Virgillito A. Distributed event routing in publish/subscribe communication systems: a survey. In: Technical Report TR-1/06. mDipartimento di Informatica e Sistemistica, nUniversità di Roma 'La Sapienza' (2005)