Presumo

Java Message Service Implementation

High Level Design

Author: Dan Greff

Document version: 0.2

Introduction

When considering the design of a messaging system it is helpful to think about the basic problem that it must solve. Any basic communication system must get a message containing data from point A to point B. However, the JMS API demands much more than simply getting a message from point A to point B. It is important to understand the high level features of JMS since their implementation drives the overall design.

Asynchronous communication is the first major distinction between JMS and other communication mechanisms such as RMI. This means that when an application at point A needs to send a message to point B, it does not need to wait until point B receives the message to have a level of certainty it arrived. In JMS, the application at point A simply hands the message to JMS and assumes the middleware will achieve eventual delivery. One advantage to this style of communication is that the application at destination B, does not have to be running when point A sends the message, in order to receive the message. One could make a rough analogy of asynchronous messaging and two people sending email.

The JMS API decouples two communicating applications further through the use of intermediary targets (Queues and Topics) rather than having the applications talk directly to one another. Consequently, the application at point A does not have to know there is an application at point B. Instead, point A produces messages to an intermediary *Destination* that the application at point B may or may not be consuming messages from. This frees the application code at point A of this logic (although the designer still must coordinate the actions). The JMS implementation must receive the messages from any given application, discover where the messages needs to go to, and continually try to deliver the messages to the appropriate receivers. This is the basic problem all JMS implementations must solve. How the problem is solved is what gives each implementation unique characteristics although the outward functionality should be identical.

This document is a high level description of how Presumo solves this problem. It is intended to be a guide for developers on the project, but advanced users wishing to learn more about the Presumo architecture may find it interesting.

The first section contains the design mantras that servers as a list of what Presumo emphasizes and de-emphasizes in being a JMS implementation. The subsequent sections will describe the design of the various subsystems and major functional areas that compose Presumo.

Project Mantras

- Presumo must be small enough to be used within a single JVM by components that may wish to decouple their interactions, while being able to scale to massive distributed applications.

- Servers must be able to interconnect and communicate within scalable topologies (not fully connected clusters!!) in order to support large numbers of producers and consumers.

- Messages should rarely travel unnecessarily over a network link. This means that JMS' content-based filters will be evaluated as close to the publisher as possible.

- Presumo should be easy for developers to use with default configurations and application designers should have the ability to deploy at least small applications based on Presumo without their customers having to know it exists within the application.

- It must possess the capability to be truly embeddable within an application.

- Every effort should be made to reduce the amount of maintenance required. Features that will require end user maintenance will be ignored or de-emphasized. Application designers should be able to disable features that require end user maintenance.

- Presumo performance (in terms of message throughput) should be within 10% of the leading implementation.

- Logging should be used within the code to enable rapid debugging.

- Implementation details should be maintained with the source code in the form of javadocs.

- The communication layer must be easily adaptable to various network protocols.

- The server must be able to accept client and inter-server connections from different protocols simultaneously.

Publish/Subscribe Routing

The basis for the Presumo design is the implementation of the JMS' publish/subscriber API. One reason the pub/sub API is favored over the point-to-point API because it is more conducive to a truly distributed implementation. Most implementation favor point-to-point over pub/sub, and only put together kludged pub/sub solutions. Presumo will give JMS developers a well-designed pub/sub solution.

Presumo's distributed pub/sub design does not rely on any centralized controller. This contradicts the JMS API's reliance on JNDI since most JNDI implementations are centralized. For the first release applications will be encouraged to manually create their connections, queues and topics. Nothing will prevent application developers from creating the administered objects manually and storing them in a JNDI implementation.

Presumo is capable of many different network topologies. Figure 1 demonstrates the most simplistic topology. Here an application is using JMS to decouple communication between component *X* and *Y*. The key point with this "topology" is that no additional server is needed when an application uses Presumo's server client (client with routing logic). This allows designers to start using JMS early in the project while allowing for components to scale to different computers, as more scalability/flexibility might be needed in the future.

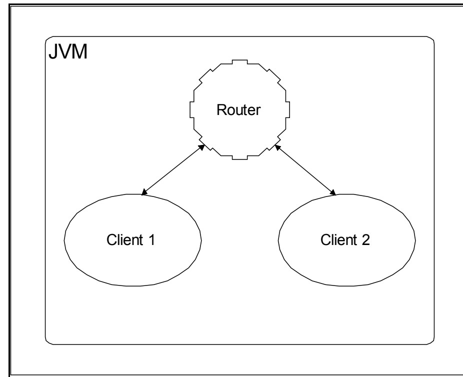

Figure 1 - Serverless "topology"

Figure 2 demonstrates Presumo in the classic client/server topology that is common with other JMS providers. A single server with multiple clients connected to the server. In Presumo the clients could be configured to communicate with the server with different protocols.

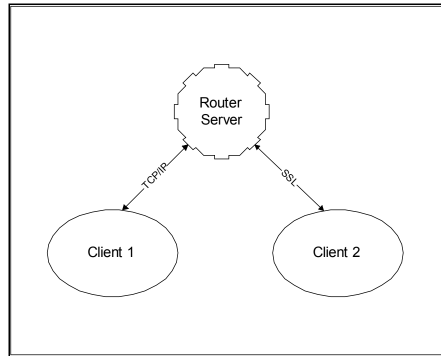

Figure 2 - Client/Server topology

While most JMS implementations can be configured in the client/server topology of figure 2, few can be configured to a more complicated topology represented in Figure 3. Here multiple Presumo servers are connected to each other with some servers having clients attached. The servers form a JMS cloud and all clients connected to any server in the cloud can talk to one another. The servers can be connected in any formation as long as a cycle is not formed. Presumo servers will be connected in any possible "spanning tree". Each server in the topology can have *n* neighbors where *n* is only limited by the performance of the machine. It is expected that large topologies will be deployed to suite the underlying network architecture. For example, servers may be placed on either side of a WAN, using a secure protocol to communicate with one another. Yet clients on more secure networks could use a more efficient protocols for communication.

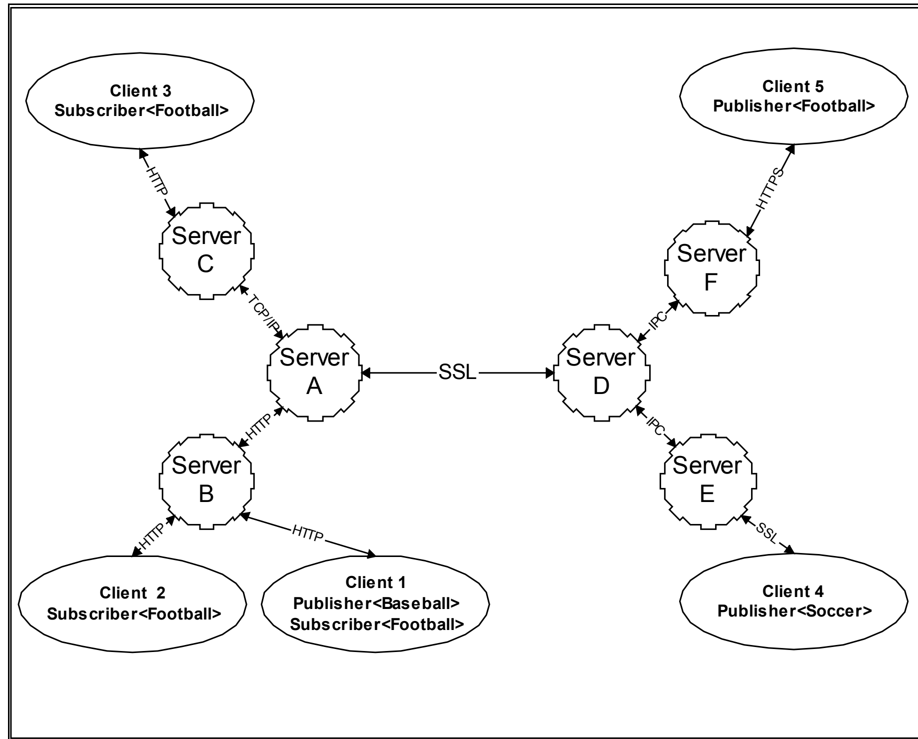

Figure 3 - Advanced topology

The Presumo servers must coordinate their activities in order to properly route JMS messages. As previously mentioned there is no centralized server instructing the publishers how to send their messages to subscribers. Instead a mechanism is employed where the routing information propagates from the subscriber to the publishers. Let us consider an example.

In Figure 3, there are three subscribers in the deployment. Server *B* has two clients, one with a subscriber to topic "Football", and another to topic "Baseball". Server *C* also has a client with a subscriber on the topic "Football". The other clients are only publishers.

After the subscribers have been created, server *B* would have told server *A* to send it any messages sent to topics "Football" and "Baseball" since it is servicing clients with subscribers on those topic. Server C would have told server *A* to send message to the one topic "Football". Server *A* would then need to tell server *C* and *D* that it needs messages from them on both topics since it is servicing clients (through one level of indirection) on those topics. Server *A* would also have to tell server *B* it needs messages on just "Football" but tell server *C* it needs messages on both topics since server *B* is serving clients with both topics. Finally, server *D* would instruct server *E* and *F* to forward messages on both topics.

> *This operation can be summarized by saying that for each server y which has a set of neighbors N, for every neighbor x in the set, a server must send a list of all topics needed by N – x (all members of the set minus x) to x.*

After the routing information has propagated, client 5 could publish a message to the topic "Football". Server *F* will know *D* is interested in the message and forward it. Server D will then forward it to server *A* but not server *E* since *E* is not servicing any subscribers on the topic at any level of indirection away from server *D*. Server *A* continues the process by

sending the message to server *B* and *C,* who finally send the message to the interested clients.

What happens when client 4 publishes messages to topic "Soccer"? Nothing. Once server *E* receives the message it is dropped immediately since nobody is interested. Another example a server making autonomous routing decisions is client 1 publishing messages to the topic "baseball". Here the messages are immediately routed the subscriber on topic "baseball" within client 2 with no help or information from other servers. This ability for servers to route messages and make decisions to drop messages with the information contained locally allows the routers to act independently in a distributed fashion and thus avoiding a potential centralized bottleneck.

Implementation Links

- Router source code

Filter Optimization

The previous section talked about how Presumo servers pass information about active subscribers on topics to determine where published messages should be routed. Presumo also performs the more complicated task of propagating and applying the JMS SQL-like content based filter in the same manner as the topic information. Actually, to simplify development the topic is actually changed to a filter (i.e. topicname = "Football") and is logically ANDed with the Subscriber's filter. This combined filter is then sent to represent the messages the Subscriber wishes to receive.

Routers are used for all routing communication. For example, when server *B* in the example based on figure 3 needs to tell server *A* what messages it needs, it logically ORs together the filters from clients 1 and 2. When server *A* needs to tell server *D* what message to pass, it ORs together the filters from servers *B* and *C*.

Persumo has a very efficient way of evaluating a message against many, or large filters. This subsystem is the main optimization. Most JMS implementations do not perform filtering on the server. Performance restrictions are cited as the main reason. This is a valid concern, consider a scenario where there is one server with 1000 subscriber's attached to it. If server side filtering is enabled, every message that passes through the server will have to be evaluated against each of the 1000 subscriber's filters. If the server hopes to maintain a throughput of 1000 messages per second, it would have to perform 1000 x 1000 (or 1 million) filter evaluations per second. This is certainly not possible and the numbers only become more impossible if you are talking about 10,000, 100,000 or 1 million subscribers.

Analysis of common use cases shows most applications use exactly the same filter or similar filters. If most of the filters are similar or identical, why evaluate each message against the same filter multiple times. Consider the previous example of 1000 filters. If these subscriptions are created by only 2 different applications running on 500 clients, and every filter created by the application is identical. We should only have to do 2 filter evaluations at the server to route a given message. As a result, to route messages at a rate of 1000 messages per second, we will only need to do 2000 filter evaluations a second.

This is certainly not impossible. In fact, the first prototype of Presumo was capable of performing over 10,000 filter evaluations per second. Presumo's filter evaluation mechanism ensures that not only common filters, but also common sub expressions within separate filters

are never evaluated more than once per message. As a result, applications with similar filters will also benefit from this optimization.

Implementation Links

- Filter implementation source code

Transport Layer

Today there are many ways a developer can send information over a network. To name a few TCP, UDP, Multicast, RMI, CORBA, HTTP, HTTPS and SSL. There is also the possible need and future unforeseen transportation mechanisms. During initial development it was not certain how this JMS implementation would be used, so the transport layer was purposely abstracted out. This will allow for the design to be used in various settings as well as allow for future improvements to the actual transportation mechanism without affecting the routing logic.

Another result of this abstraction is the ability of this JMS implementation to support multiple protocols simultaneously. This will be useful for backward compatibility and for situations that will require different protocols. Consider a multi-homed gateway sitting between protected and unprotected domains. The router could be able to accept SSL connections from the computers in the unsecured domain and TCP connections from computers within the protected domain.

The primary purpose of a pluggable interface is to allow for developers to work on different transportation protocols without worrying about the details of the routing logic. It would be possible for an application to supply its own transportation protocol. While possible it probably is not prudent to allow each application to implement and use separate transportation mechanisms. Rather multiple protocols should be built into the system, and the administrator should select the protocols the routers will use for communication.

Implementation Links

- Transport interfaces
- TCP/IP Transport implementation

Persistence

JMS supports a notion of persistent messages which are defined to be delivered once-and-only once. Presumo will accomplish reliability without requiring fault-tolerant hardware. The design favors surviving hardware failure by failing over to backup systems.

To survive temporary system outages, a JMS implementation must write *every persistent* message published to a transactional system. After the message is safely stored it may be routed to a destination. Once the destination has received and acknowledged the message, it must be deleted from the transactional system. Most RDMS cannot handle this amount of transactional deletes and writes since they are optimized for reads. This is why other transactional messaging systems wrote their own transactional storage mechanism and

is why this design will call for our own customized transactional queuing system optimized for highly granular deletes and writes with commits.

Furthermore, an asynchronous messaging API should be *asynchronous*. This means that an implementation must strive to allow for applications to publish messages without incurring any network I/O penalties. JMS implementations must somehow place a message safely in the system and return from the publish call, *then* route the message. This also means that applications can publish messages when the network is down, but the publish still returns successfully because the JMS implementation has queued the message in persistent storage for later delivery when the network is recovered. JDBC will not be used to store messages since it may incur network traffic for every message written or deleted.

To survive permanent system outages without relying on fault-tolerant hardware, Presumo must guarantee that a particular message is stored in separate nodes. In the event of a total system failure (the node can not reboot or has lost its physical media), Presumo must have enough state information stored in nodes that did not fail to determine who has and has not received a given message. Once you have done all of this work, it is possible to have a persistent message implementation that does not actually write messages to disk, but rather always guarantees that a message is stored in memory on separate machines.

While possible to implement Presumo as a system with “in-memory” persistence, this will not be done. Presumo will continue to store message on the disk within transactions in a methodology similar to store-and-forward, while still maintaining the same amount of state information necessary for “in-memory” persistence. This means Presumo should not loose a persistent message unless two adjacent routers fail completely within the time period required to recover the first failed system. Nothing will prevent administrators from place Presumo on HA boxes thus making the chance of two adjacent systems loosing their physical media almost simultaneously extremely small.

To reduce network traffic, acknowledgements should be batched, or piggyback on top of other messages. Depending on how efficient this mechanism is, all messages could follow use the same mechanism (including non-persistent messages). Persistent messages would have the additional fault-tolerance of being persisted to disk.

Implementation Links

- Persistent queue source code

Client

Eventually there will be two types of JMS clients in Presumo. The first type to be implemented will be a client that resides in the same JVM as the router. In other words the client will be the same code base as the server/router. The second type of client for future releases would be a true lightweight client with a minimal amount of functionality.

The server client will continue to exist even when there is a true lightweight client because there is a valid use case for it. For example, the server client could be embedded into an application using JMS for intra-JVM inter-component communication. The server client will be able to route messages without an external server.

The lightweight client's architecture should be language independent, so it can be implemented in other languages besides Java. The java version should be able to run on 1.1.x JVMs to enable its use within applets.

Implementation Links

- The server client source code

Point-to-Point and Durable Subscriptions

The point-to-point implementation in Presumo will be based on the pub/sub routing infrastructure. This is different than most implementations which implement pub/sub on top of their existing point-to-point routing infrastructure. Presumo is dedicated to being a high quality pub/sub implementation. The point-to-point implementation should be a quality product and perform well, but will not be emphasized as much as pub/sub.

Implementation Links

- Queue manager source code
- Client implementation

Logging and Internationalization

Presumo uses logging as the main mechanism of debugging and reporting errors. The logging mechanism is pluggable so that an application may substitute their own logger in if they wish. They should simply have to implement Presumo's Logging interface and specify which logger to use in a configuration file.

There are four levels of logging in Presumo.

1. Debug – For development debug messages (not internationalized/English)
2. Informational – Internationalized non-critical informational messages for the user.
3. Warning – Internationalized messages informing user of a potentially dangerous state (queue almost full).
4. Error – Internationalized messages informing the user that something went wrong (queue full and message dropped).

All messages that the end user might see should come from an internationalized resource bundle. All internationalized messages should have a message id placed in the translation string so that developers can reverse translate any given log.

While the importance of logging will be stressed, certain critical sections of code will not contain any logging statements for performance. These sections can usually be identified as methods that are called multiple times per message routed. Adding logging to these types of methods will affect message throughput, and therefore must be omitted. Code quality must be ensured in these sections with rigorous reviews.

Implementation Links

- Logging interface and implementation