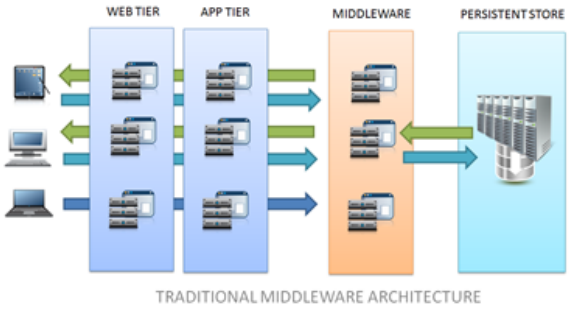
Web 2.0 Killed the Middleware Star

Lori MacVittie, 2011-26-07

Pondering the impact of cloud and Web 2.0 on traditional middleware messaging-based architectures and PaaS.

It started out innocently enough with a simple question, "What exactly *is* the model for PaaS services scalability? If based on HTTP/REST API integration, fairly easy. If native middleware… input?" You'll forgive the odd phrasing – Twitter's limitations sometimes make conversations of this nature … interesting.

The discussion culminated in what appeared to be the sentiment that middleware was mostly obsolete with respect to PaaS.



TRADITIONAL MIDDLEWARE ARCHITECTURE

THE OLD WAY

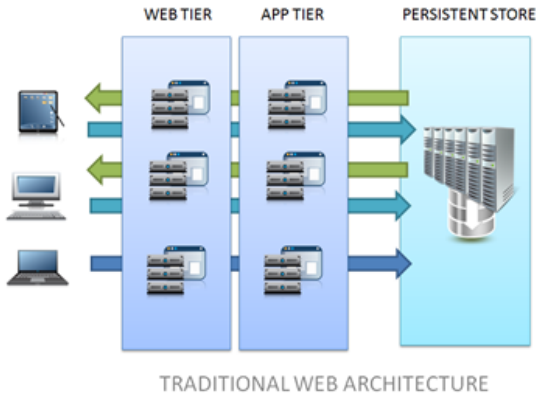
Very briefly for those of you who are more infrastructure / network minded than application architecture fluent, let's review the traditional middleware-based application architecture.

Generally speaking, middleware – a.k.a. JMS, MQ Series and most recently, ESB – is leveraged as means to enable a publish-subscribe model of sharing data. Basically, it's an integration pattern, but no one really likes to admit that because of the connotations associated with the evil word "integration" in the enterprise. But that's what it's used for – to integrate applications by sharing data. It's more efficient than a point-to-point integration model, especially when one application might need to share data with two, three or more other applications. One application puts data into a queue and other applications pull it out. If the target of the "messages" is multiple applications or people, then the queue keeps the message for a specified period of time otherwise, it deletes or archives (or both) the message after the intended recipient receives it.

That pattern is probably very familiar to even those who aren't entrenched in enterprise application architecture because it's similar to most social networking software in action today. One person writes a status update, the message, and it's distributed to all the applications and users who have subscribed (followed, put in a circle, friended, etc…). The difference between Web-based social networking and traditional enterprise applications is two-fold:

- **First** – web-based applications were not, until the advent of Web 2.0 and specifically AJAX, well-suited to "polling for" or "subscribing to" messages (updates, statuses, etc…) thus the use of traditional pub-sub architectures for web applications never much gained traction.
- **Second** – Middleware has never scaled well using traditional scalability models (up or out). Web-based applications generally require higher capacity and transaction rates than traditional applications taking advantage of middleware, making middleware's inability to scale problematic. It is unsuited to use in social networking and other high-volume data sharing systems, where rapidity of response is vital to success.

Moreover as James Urquhart noted earlier in the conversation, although cloud computing and virtualization appear capable of addressing the scalability issue by scaling middleware at the VM layer – which is certainly viable and makes the solution scalable in terms of volume – this introduces issues with consistency, a.k.a. CAP, because persistence is not addressed and thus the consistency of messages across queues shared by users is always in question. Basically, we end up – as pointed out by James Saull - with a model that basically kicks the problem to another tier – the scalable persistence service. Generally that means a database-based solution even if we use the power of virtualization and cloud computing to address the innate challenges associated with scaling messaging middleware. Mind you, that doesn't mean an RDBMS is involved, but a data store of some kind and all data stores introduce similar architectural and technologically issues with consistency, reliability and scalability.



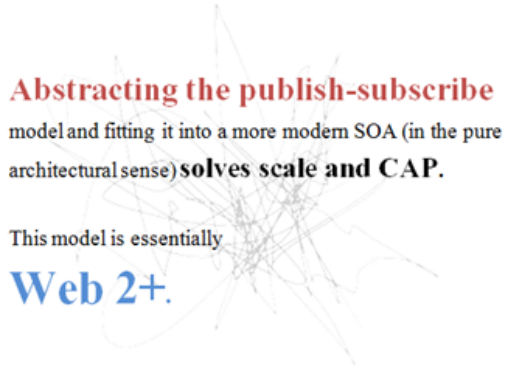
TRADITIONAL WEB ARCHITECTURE

THE NEW WAY

Now this is not meant to say the concept of queuing, of pub-sub, is absent in web applications and social networking. Quite the contrary, in fact. The concept is seen in just about any social networking site today that bases itself on interaction (integration) of people. What's absent is the traditional middleware as a means to manage the messages across those people (and applications). See, scaling middleware ran into the same issues as stateful applications – they required persistence or a shared-nothing architecture to ensure proper behavior. The problem as you added middleware servers became the same as other persistence-based issues seen in web applications, digital shopping carts and even today's VDI implementations. How do you ensure that having been subscribed to a particular topic that you actually manage to get the messages when the load balancing solution arbitrarily directs you to the next available server?

You can't. Hence the use of persistent stores to enable scalability of middleware. What you end up with is essentially 4 tiers – web, application, middleware and database. You might at this point begin to recognize that one of these tiers is redundant and, given the web-based constraints above, unnecessary. Three guesses which one it is, and the first two do not count.

Right. The middleware tier. Web 2.0 applications don't generally use a middleware tier to facilitate messaging across users or applications. They use APIs and web-based database access methods to go directly to the source. Same concept, more scalable implementation, less complexity. As James put it later in the conversation, "*the "proven" architecture seems to be Web2, which has its limitations*."



BRINGING IT BACK to PaaS

So how does this relate to PaaS? Well, PaaS is Platform as a Service which is really a nebulous way of describing developer services delivered in a cloud computing environment. Data, messaging, session management, libraries; the entire application development ecosystem. One of those components is messaging and, so it would seem, traditional middleware (as a service, of course). But the scalability issues with middleware really haven't been solved and the persistence issues remain.

Adding pressure is the web development paradigm in which middleware has traditionally been excluded. Most younger developers have not had the experience (and they should count themselves lucky in this regard) of dealing with queuing systems and traditional pub-sub implementations. They're a three-tier generation of developers who implement the concept of messaging by leveraging database connectivity directly and most recently polling via AJAX and APIs. Queueing may be involved but if it is, it's implemented in conjunction with the database – the persistent store – and clients access via the application tier directly, not through middleware.

The ease with which web and application tiers are scaled in a cloud computing environment, moreover, meets the higher concurrent user and transaction volume requirements (not to mention performance) associated with highly integrated web applications today. Scaling middleware services at the virtualization layer, as noted above, is possible, but reintroduces the necessity of a persistent store. And if we're going to use a persistent store, why add a layer of complexity (and cost) to the architecture when Web 2.0 has shown us it is not only viable but inherently more scalable to go directly to that source?

At the end of the day, it certainly appears that between cloud computing models and Web 2.0 having been forced to solve the shared messaging concept without middleware – and having done so successfully – that middleware as a service is obsolete. Not dead, mind you, as some will find a use case in which it is a vital component, but those will be few and far between. The scalability and associated persistence issues have been solved by some providers – take RabbitMQ for example – but that ignores the underlying reality that Web 2.0 forced a solution that did not require middleware and that nearly all web-based applications eschew middleware as the mechanism for implementing pub-sub and other similar architectural patterns. We've gotten along just fine on the web without it, why reintroduce what is simply another layer of complexity and costs in the cloud unless there's good reason.

Viva la evolution.

- The Inevitable Eventual Consistency of Cloud Computing
- Let's Face It: PaaS is Just SOA for Platforms Without the Baggage
- Cloud-Tiered Architectural Models are Bad Except When They Aren't
- The Database Tier is Not Elastic
- The New Distribution of The 3-Tiered Architecture Changes Everything
- The Great Client-Server Architecture Myth
- Infrastructure Scalability Pattern: Sharding Sessions
- Infrastructure Scalability Pattern: Partition by Function or Type
- Applying Scalability Patterns to Infrastructure Architecture
- Sessions, Sessions Everywhere

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com

©2016 F5 Networks, Inc. All rights reserved. F5, F5 Networks, and the F5 logo are trademarks of F5 Networks, Inc. in the U.S. and in certain other countries. Other F5 trademarks are identified at f5.com. Any other products, services, or company names referenced herein may be trademarks of their respective owners with no endorsement or affiliation, express or implied, claimed by F5. CS04-00015 0113