

The SOA Magazine

Feature Article



Next-Generation Grid-Enabled SOA: Not Your MOM's Bus

by David Chappell and David Berry

Published: January 7, 2008 (SOA Magazine Issue XIV: January 2008, Copyright © 2008)

[Download this article as a PDF document.](#)

Abstract: In our previous article we discussed how SOA grids can be used to break the convention of stateless-only services for scalability and high availability (HA) by allowing stateful conversations to occur across multiple service requests, whether between disparate service boundaries or load-balanced groups of cloned service instances.

In this article we will challenge traditional applications of message-oriented middleware (MOM) for achieving high levels of quality of service (QoS) when sharing data between services in an enterprise service bus (ESB). We will further compare and contrast a state-based, in-memory storage and notification model, and investigate the intelligent co-location of processing logic with or near its grid data in large payload scenarios. Finally, we will also explain when to substitute an SOA Grid for existing MOM technologies as driven by the following question: "If you have an SOA grid that can reliably hold application state data and the necessary systems can access it, why continue to utilize conventional messaging?"

Introduction: Facing SOA Adoption Challenges with the SOA Grid

SOA initiatives are keeping their promise to deliver core, strategic benefits, including reduced cost and complexity, better alignment of IT with business needs, and an increased ability to consolidate, integrate, upgrade, and even replace IT systems. However, as early adopters work through initial SOA projects, there is a new set of recurring issues that can limit what they are able to achieve.

There are also architectural challenges, such as how to handle large XML payloads of tens and hundreds of megabytes, as well as latency and throughput issues as shared services are recombined and reused in ways that result in unexpected usage demands. These issues raise constant concerns about fulfilling the original expectations of service-level agreements.

Furthermore, from an application development perspective, tearing down application silos into decoupled services introduces challenges of its own, including, for example, how to share application context (or service state) between services that are recombined across boundaries, such as network separations. A related issue is how to share service state across multiple instances of the same kind of service in a load-balancing scenario.

As we collectively learn from our successes and mistakes and reach new plateaus of understanding of how best to adopt SOA, there are new opportunities to improve upon traditional practices. Now is also a good time to introduce new practices in the context of new technologies, which in the end make life easier for architects, developers, and IT operations, and also make systems more scalable and responsive.

In the previous article [REF-1] we introduced the concept of an SOA grid. Let's recap its basic capabilities:

- The SOA grid is a new approach to thinking about SOA infrastructure. It provides state-aware, continuous availability for service implementations, data, and processing logic. It's based on architecture that combines horizontally scalable, database-independent, middle-tier data caching with intelligent parallelization and an affinity of business logic with cache data. This enables newer, simpler, and more-efficient models for highly scalable service-oriented applications that can take full advantage of service virtualization and event-driven architectures.
- At the core of the SOA grid (Figure 1) is a group of interconnected software processes that cooperate together across a network with the sole purpose of storing and maintaining instance data for individual application objects. An application or service uses the grid through a familiar data structure, such as a Java HashMap or a .NET dictionary. When the application performs a put() to the HashMap, that operation is delegated to the grid, which then automatically elects a primary storage node and a backup storage node on another machine.
- The nodes on the grid are continually aware of the health and status of all other nodes. They automatically and immediately detect failures and then rebalance the data storage and workload among the remaining nodes. For example, if a primary storage node fails, the backup node becomes the primary and a new primary/backup relationship is established. Subsequent requests to update or fetch data will always get routed to the appropriate primary grid node for a given service object. This rebalancing happens in real time, and can occur seamlessly across multiple failures, even while updating the primary grid node. Through this mechanism, the grid can provide high-speed, in-memory access to continually available service state data without the need for disk persistence.

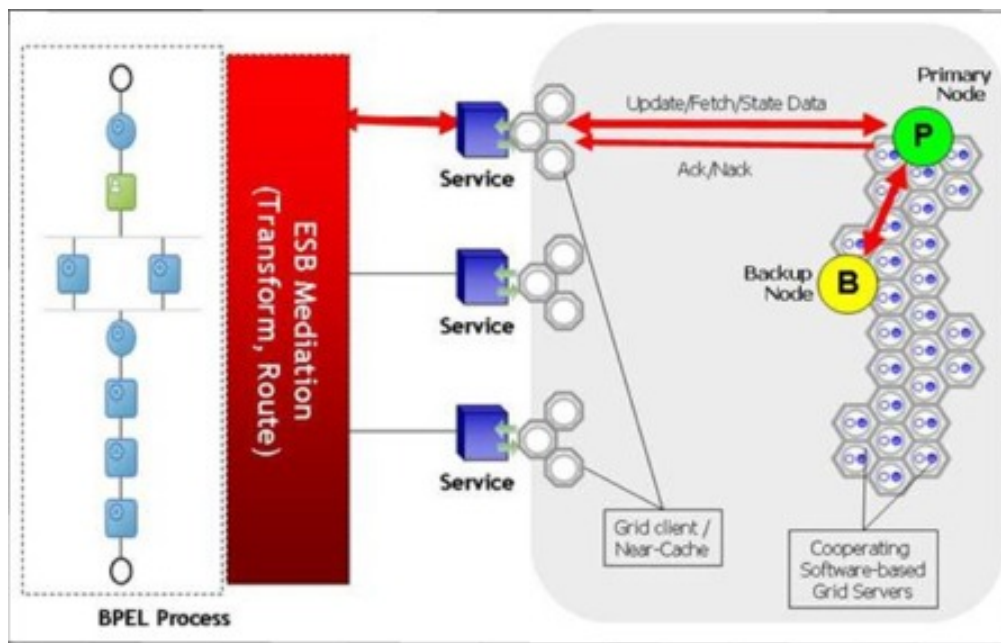


Figure 1: An SOA grid provides high-speed in-memory access to continually available service state data without the need for disk persistence.

MOM as Part of ESB Technology

Today's ESBs and related SOA infrastructures all use some kind of message-oriented middleware, whether based on proprietary MOM, JMS, or WS-RM, to achieve quality of service when sharing data between applications and services (Figure 2). The term "MOM" is used here generically to describe messaging techniques such as store-and-forward and message retries and redelivery to ensure that message data gets reliably transported from one place to another.

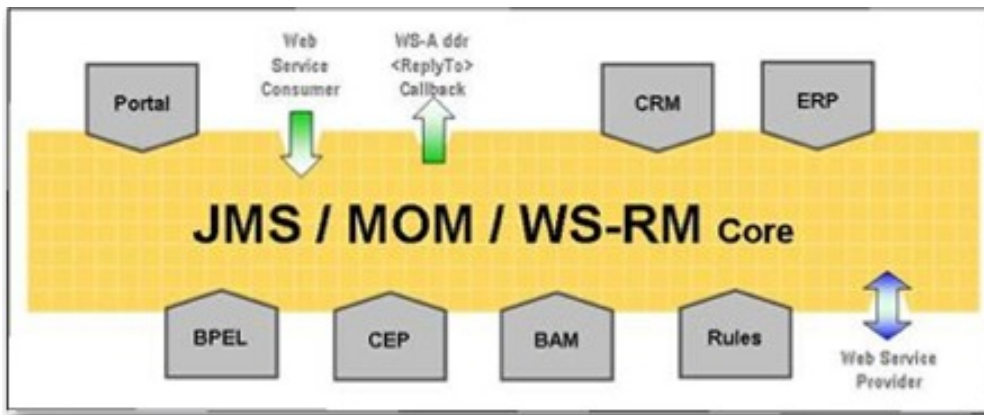


Figure 2: An example of conventional messaging for QoS where the MOM core of an ESB reliably transports data from one service to the next.

The "New" Bus

As we continue to enhance the SOA grid vision by addressing practical ramifications, some realizations help us to reevaluate assumptions about how to reliably get data from one place to another. For example, if you have an SOA grid that can reliably hold application state data, and the necessary systems can access it, why continue to utilize conventional messaging? Particularly for large XML data sets, why create large JMS or SOAP messages and send them across the wire through the ESB simply to take them out again? And, if the data in question is held reliably in the SOA grid and all interested parties can access it, why "send" it anywhere?

The next-generation service bus inherently has direct access to service state data using in-memory, grid-based, stateful HA, BPEL process orchestration capabilities. These are essentially today's notions of ESB capabilities such as data transformation and routing, service-level abstraction, and protocol and adapter support (Figure 3).

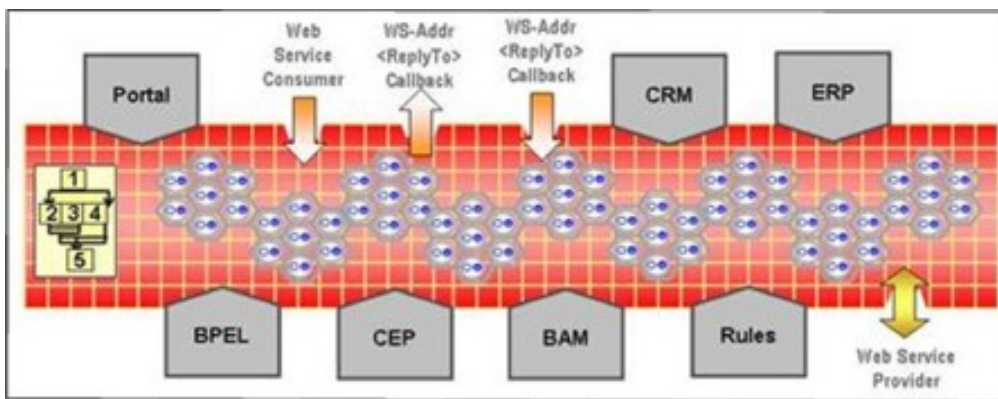


Figure 3: Why send it when it's already there? A new service bus based on an SOA grid combines in-memory access to HA service state, BPEL orchestration, event notifications, and conventional ESB mediation.

SOA Grid and State-Based Event Notifications

Among the advanced capabilities in new generation ESBs is an eventing mechanism based on state data changes, allowing listeners to use an observer pattern [REF-2] to monitor for state changes in the grid. As data is placed into grid nodes, notifications are emitted to all interested parties. As illustrated in Figure 4, this mechanism can be used to asynchronously notify and copy deltas in application state into a relational database or ETL environment.

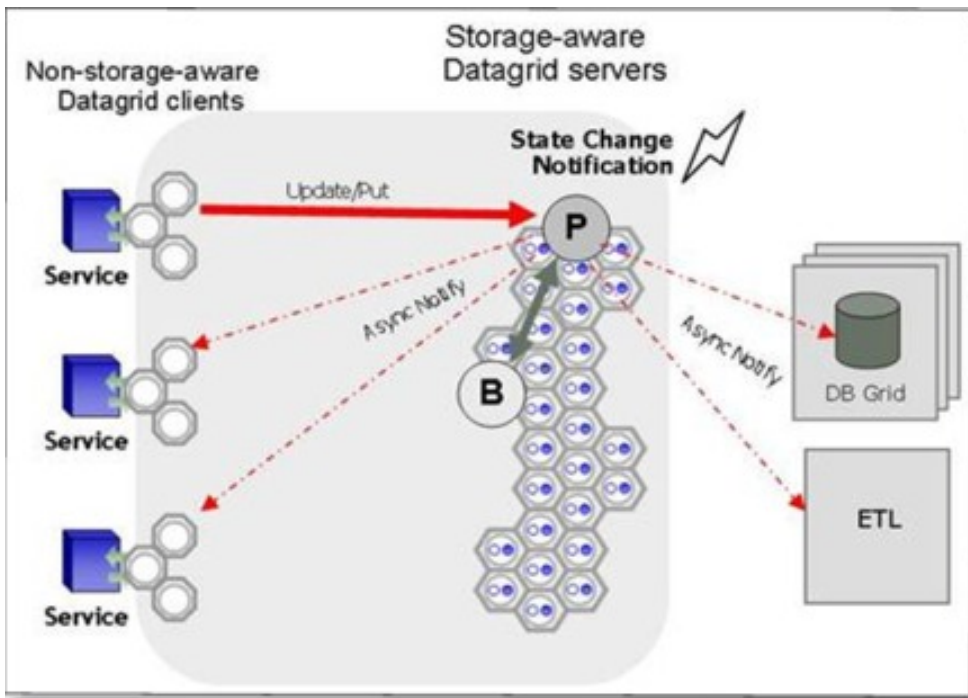


Figure 4: Notifications on state changes keep all interested parties informed of data changes in the grid.

Now Hold on a Minute! Does MOM Go Away?

There are still plenty of use cases for conventional messaging techniques in an ESB. For example, queuing semantics are still useful for in-order message delivery, and you can continue to use publish-and-subscribe for one-to-many broadcasts. Or, you may have existing JMS applications that you simply don't want to retool. Either way, an SOA grid should ideally support both MOM and direct, high-speed grid memory access.

A MOM implementation built on top of an SOA grid (Figure 5) would actually result in an intriguing architecture. In terms of JMS parlance, if all JMS connection, session, topic, and queue objects were stored in the grid, they would automatically enjoy the capabilities that the grid has to offer, such as predictable linear scalability and fault tolerance with near-in-memory access speeds using the non-persistent primary/backup contract between grid nodes. Likewise, a WS-RM implementation could use the grid for `<wsrm:Sequence>` management tasks, such as tracking gaps in sequenced message deliveries and managing sequences across stateless Web interactions (there's that state thing again).

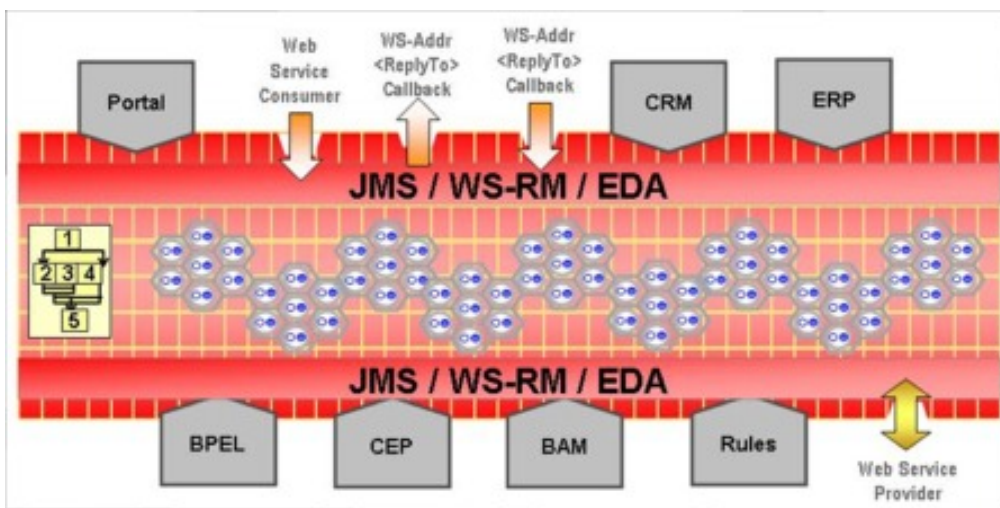


Figure 5: A hybrid architecture with MOM built on top of grid.

When to Use the Grid State Model

Ultimately, the choice between the grid state model and MOM boils down to architectural preferences, and a hybrid approach is the most flexible option. Table 1 enumerates the tradeoffs, but here are some basic guidelines.

- Don't put state into queues where it doesn't belong. We have seen an application use a queue simply as a place to put things between service requests.
- Don't send it when it's already there.
- Don't retool messaging applications unless you see a clear advantage for utilizing direct state management in the SOA grid.
- Use durable message queues and durable subscriptions for ordering and sequencing, particularly when all values are important for a given period of time. An SOA grid will always contain the most-recent value of a given piece of state data, but sometimes you need to view and process data in a sequential order and that represents a value that was meaningful relative to the state of a business transaction at the time it was emitted by the application.
- Use publish-and-subscribe messaging techniques for one-to-many data broadcasting, particularly if the messaging system allows hierarchical trees of subscription topics to be managed and maintained with wildcard subscriptions. In less sophisticated one-to-many relationships, you may want to consider an eventing mechanism that is built on an observer pattern of state changes. In fact a messaging layer that is built on top of an SOA grid could base its publish-and-subscribe implementation upon this pattern.

Usage	MOM	Grid
Stateful services	No	Yes
Legacy apps	Yes	Grid-enabled adapters may be an option for future consideration
Sequencing	Yes	Only if MOM is implemented on grid
Publish/subscribe	Yes	Observer pattern on state changes
Claim check	Event only	Payload
Distributed object cache	No	Yes
Reliable transport	Yes	Yes
Intra-application communications	Maybe	Yes
Eventing	Yes	Yes
Asynchronous request/response	Yes	Not likely, but this would be a great place to store correlation IDs
Large document handling	Event only	Document storage

Table 1: A comparison of MOM vs. SOA Grid usage tradeoffs.

Large document handling	Event only	Document storage
Content-based routing	Yes	Depends on how "routing" is defined

You may wonder about what's happening "under the covers." Aren't we still moving data? The simple answer is yes. Sometimes it's a matter of which architectural pattern is more elegant for the application—state-based eventing or message-based eventing. In large data scenarios, the SOA grid/state change observer model can be much more efficient in implementing a claim check/pass-by-reference pattern, which we will discuss in a little bit. Let's first consider the following.

Combining Data Grid and Compute Grid Capabilities

The SOA grid combines elements of a data grid and a compute grid in intelligent ways. Grid nodes can store data and host business logic. Performance optimizations, load balancing, and parallelization may be efficiently achieved by applying patterns of affinity between processing logic and grid storage. Grid nodes may be co-located on the same machine or in the same process space with service logic that uses the grid for data storage. This type of node might be used as an application's physical entry point into a grid, or as a local or "near" cache for data that is fetched from remote grid nodes.

Likewise, service logic might be co-located with primary grid storage nodes. This way, the service logic can be invoked across the grid and operate on its own data storage without incurring the network overhead of transferring that data between the primary storage-enabled grid node and the service logic that operates on that data. This is invaluable when dealing with large message payloads.

More generally, though, it breaks the mold of load balancing or even parallel processing conventions in that the execution of business logic (or in SOA parlance, dispatching of service requests) across the grid can always follow where the data has been stored. Grid storage is automatically balanced across as many nodes as necessary in a fashion that is horizontally scalable and that can be partitioned among specific banks of machines within the grid domain, based on configuration hints. As a result, the dispatching of service requests across the grid is naturally balanced.

Dealing with large data sets, common in XML messages, is a continual challenge in SOA-based applications. The accepted practice is to invoke services and serialize business data over the wire using a variety of transports. When the message payload becomes too large, a contention begins between the desire for a clean separation of loosely coupled services and the desire to avoid the boundary cost of serializing the message data.

An efficient solution for handling large messages is to use the claim check pattern [REF-3], which implements a pass-by-reference model in which the majority of the data is held in a stationary place, and a unique identifier that acts as a handle to the data is passed between services. This handle is analogous to the claim check ticket you receive when you hand over luggage to a doorman at a hotel or check your coat at a theater or restaurant. Each service that receives the claim check contacts the source of the data and retrieves what it needs from the stored message data to process it.

Figure 6 illustrates the Claim Check pattern as it applies to an SOA grid. There are three unique characteristics about this approach that are worth mentioning.

- The pattern uses a special grid-enabled ESB mediation service that inherently knows how to store the payload in the grid, create the claim check, and either feed it back to the BPEL process or send it as an event directly to the next service in the process flow (which, in this case, is the ERP system). Conversely the ERP system either uses a grid-enabled adapter that implements the claim check retrieval step or it is a service that implements the pattern directly.
- The pattern is enabled through the use of the grid's logic/data co-location capability to implement a filter that returns a partial XML fragment. This fragment is a subset of the message payload that is needed by this service instance.

- The pattern uses state change notifications on the grid node to trigger asynchronous updates to a relational database or an ETL environment. As downstream services make modifications to the stored grid data, the filter can be applied again to only send deltas in the database or ETL environment.

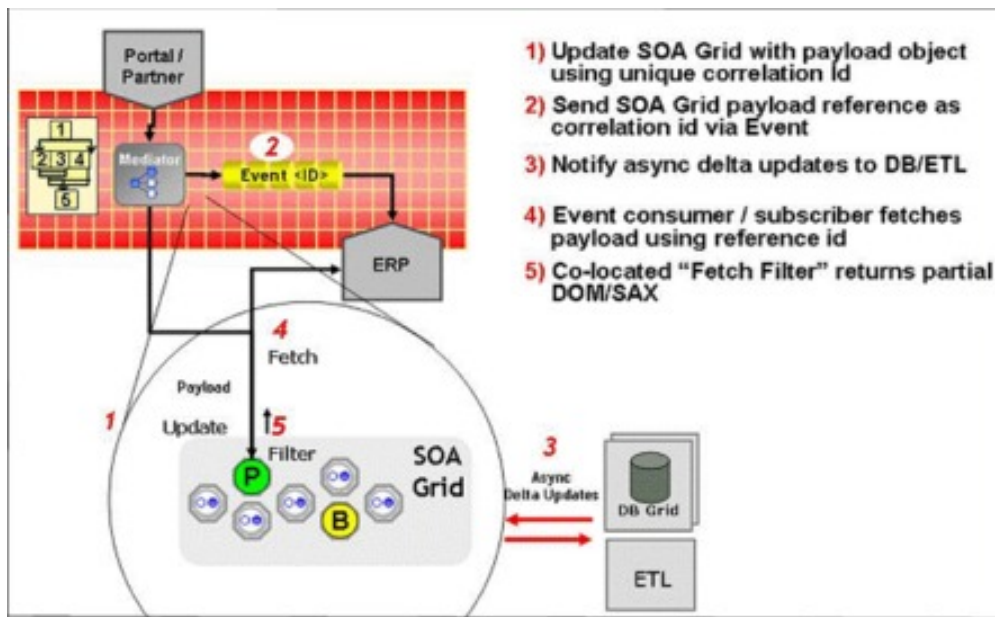


Figure 6: An SOA Grid mediation use case, where the Claim Check pattern is applied to a grid with Eventing and DOM fragment filtering.

Performance Expectations

We would like to leave you with a few closing thoughts on expectations using an SOA grid:

- In-memory speeds**—On simple deployments with one or two machines, there are some performance gains to be had by using the grid's in-memory cache mechanism in which the data is asynchronously written to disk at a later time. The amount of actual gain depends on the application, but generally it is similar to the benefits achieved by using in-memory JMS (which is typically about 10 to 20 percent).
- Parallel processing**—The SOA grid supports parallel interfaces, so the performance benefits depend on the extent to which the applications can leverage them. This will be the case even when running on different threads on the same node. It may be a bit of work to find the correct optimization points in your application, but it's well worth the effort.
- Scalability**—When you take the gains achieved on a single system with in-memory speeds and parallel processing and apply them to every node in a cluster, the true value of the grid becomes readily apparent.

Conclusion

An SOA grid transparently solves many of the difficult problems related to achieving high availability, reliability, scalability, and performance in a distributed environment. Service-oriented architectures can fully leverage such a grid to establish a QoS infrastructure far beyond the typical distributed service integration currently delivered by conventional SOA techniques.

References

- [REF-1] "[SOA - Ready for Primetime: The Next-Generation, Grid-Enabled Service-Oriented Architecture](#)" by David Chappell and David Berry (The SOA Magazine, Issue X, September 2007)
- [REF-2] "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma, Helm, Johnson, Vlissides (Addison-Wesley, 1994)

[REF-3] "Enterprise Integration Patterns" by Hohpe, Woolf (Addison-Wesley, 2004)

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Glossary](#) [SOA Books](#) [Legal](#) [RSS](#) Copyright © 2006-2007 SOA Systems Inc.
All Rights Reserved