Abstract: Whether you are learning about service-oriented architecture or already working with it, you will no doubt have had some anxiety about building a supporting IT infrastructure capable of meeting its runtime demands. Issues such as intermediate layers, large XML payloads, reliability, and scalability across heterogeneous systems have raised concerns among many practitioners.

In this article, we’ll cover some of the key technologies that have emerged to address common SOA scalability requirements, such as mid-tier caching, load balancing, and high availability (HA) through service-level grid enablement in support of building out high performance SOA implementations. Collectively, these infrastructure components are referred to as the “SOA grid,” providing enterprise IT professionals with much-needed support for enabling enforceable service-level agreements (SLAs) across entire service portfolios, including Web services, messaging, custom enterprise system applications, and legacy mainframes.

Introduction: SOA Performance Challenges

Chances are the SOA-based projects you are working on involve building automated processes that combine multiple services into composite applications. These can then be exposed to an unpredictable amount of traffic that originates from external Web users (most likely your customers). As SOA projects become more successful, subsequent projects naturally begin to build upon one another, and new parts of the organization can start realizing the benefits of reuse by tapping into existing services and leveraging composite processes to achieve additional functionality. Even though this can result in an alarming increase in overall traffic both within and outside of the enterprise, the throughput requirements of your business processes cannot be compromised.

Automated trading is but one example where even a few milliseconds can make a big difference in the aggregated throughput and performance of the entire system. This section describes different types of technologies that impact this latency as a prelude for the discussion on how an SOA grid can help solve these problems.

Scalability, Throughput, and Points of Congestion

As you begin to increase traffic in a service-oriented computing environment, the bottlenecks tend to occur in one of three places:

- *Shared intermediary services* - Such services perform common integration tasks such as data transformation, content based routing, and filtering.

- *The services themselves* - That is, application code exposed as a service and invoked by other services on the network, whether directly or through an orchestration engine.
The SOA Magazine

- **SOA infrastructure operations** - These operations ensure quality of service (QoS), such as the persistence of Business Process Execution Language (BPEL) process state through a “dehydration store,” or the store-and-forward techniques used to ensure “exactly once” delivery of messages via Java Message Service (JMS) or Web Services Reliable Messaging (WSRM) implementations within delivery channels.

In most cases, the scalability bottlenecks across these SOA components are caused when disk I/O, memory, or CPU saturation levels are reached.

### Large Payloads and Memory Constraints

Over-sized payloads (10+ MB), especially if delivered in high-frequency bursts, can certainly slow down end-to-end response times. If a given amount of demand on a service results in exhausting the memory limitations on the machine the service is hosted on, then the typical solution is to deploy that service on multiple machines that are dedicated for that purpose - and - load balance requests across those multiple instances of the service.

Streaming, de-batching, claim checks, and pass-by-reference are also often used to process large documents, but benefits are generally limited to when documents are subsequently processed in the same container. Offloading the transient memory storage (combined with intelligent load balancing and co-location of processing logic across a smart distributed caching cluster) is a much-preferred approach for memory-intensive services, such as those that need to process large-sized documents.

In a distributed environment, anything that causes file I/O can slow things down and thereby lessen the chances of attaining a truly scalable architecture. In general, stateless services are easiest to scale because they never have to write or persist the service state to a file or database. However, not all applications or services can be written to be completely stateless.

### High Availability and Reliability

High availability (HA) in SOA, as in any major architecture, is generally defined as the constant availability of a service regardless of the status of the hosting or dependent servers on which it runs. Server redundancy is the classic solution to this problem, where a group of servers typically employs peer-to-peer technologies to share “awareness of health” across the cluster—if any one server goes down, services are automatically processed on a backup, usually in an “active-active” failover fashion.

Most clusters providing HA and reliability depend on static partitioning, where a single backup server is pre-assigned to service requests from a failing server. This approach requires significant advance configuration of many servers in order for them to be fully prepared for system outages. Typical hardware clustering solutions need all the server machines in the cluster to be identical machines with identical OS versions, right down to the patch level. Applying a security patch at the OS level can therefore require bringing down the whole cluster.

In some cases, it can take up to 15 minutes for the hardware clustering to fail over to the new machine, and for the relevant SOA infrastructure to notice that failover has happened. At this point it will attempt to re-establish all connections and sessions, which may or may not be successful. In a high-volume, continuously available business, 15 minutes of downtime can seem like an eternity, particularly if the service-oriented solution includes a customer-facing portal. Your clientele may simply decide to go somewhere else to conduct their business while your systems are offline.

A better approach is to have a smart cluster, one that is specifically designed for memory access, load balancing, and HA for an SOA environment. This type of cluster works across heterogeneous, low-cost commodity hardware and can dynamically assign one or more backup systems in the cluster as needed.

### Making Your SOA Bulletproof

An SOA grid is the critical enabler of an architecture for true linear scalability and maximum availability, as highlighted by the following data points:

- 100 percent active-active server failover
• zero single points of failure
• automatic service load distribution
• full range of QoS levels for stateful services and service orchestrations with in-memory access speeds
• increased throughput (typically a 3x–10x improvement, with some extreme cases showing a 1000x improvement in throughput)
• self-healing management and SLA enforcement

An architecture for continuous availability is accomplished by removing the complexity of designing the optimal HA architecture with self-healing management and proactive load balancing of services, based on real-time complex service load analysis. When nodes of the grid go down, others take over. When new nodes start up, they take on their requisite share of the load and increase total throughput. In short, the SOA grid empowers administrators to actively monitor and respond to operational issues that are impacting customer service levels.

Mid-tier Caching of Stateful Services

A critical part of a grid-enabled SOA environment is a middle-tier caching layer. This layer provides a JCache-compliant, in-memory, distributed data grid solution for state data that is being used by services in a service-oriented solution.

The middle-tier caching layer offloads the memory storage of a service instance to other machines across the grid. This effectively provides a distributed shared memory pool that can be linearly scaled across a heterogeneous grid of machines (which can include both high-powered, big-memory boxes, and other lower-cost commodity hardware).

In a grid-enabled, stateful service-oriented environment (one that makes use of this middle-tier caching layer), all the data objects that an application or service puts in the grid are automatically available to and accessible by all other applications and services in the grid, and none of those data objects will be lost in the event of a server failure. To support this, a group of constantly cooperating caching servers coordinate updates to shared data objects using cluster-wide concurrency control.

The key to this approach is in ensuring that each piece of data is always managed by a primary owner and an additional backup server. This data can be anything from simple variables to complex objects or even large XML documents. From the perspective of the developer, the service is simply performing operations against a programmatic interface to a collection, such as a Java map or a .NET dictionary.

As shown in Figure 1, the request to put data to the map is taken over by the SOA grid and transported across a highly efficient networking protocol to the grid node P, which owns the primary instance data. The primary node in turn copies the updated value to the secondary node B for backup, and then returns control to the service once the proper acknowledgments are handled.
Conversely, when the instance data needs to be accessed by a service, the SOA grid automatically locates the primary node which owns that instance data, and routes the data back to the service or business object that is requesting it. A wide range of operations are supported in this manner, including parallel processing of queries, events, and transactions. In a more advanced implementation, an entire collection of data can be put to the grid as a single operation, and the grid can disperse the contents of the collection across multiple primary and backup nodes in order to scale.

**Continuous Availability with In-Memory Access Speeds**

The reason primary and backup nodes hold onto the instance data is for high-availability and failover purposes. If that primary node fails, the SOA grid can automatically detect that condition and immediately route subsequent data access requests to one of the backup nodes (this effectively makes that node the new primary), while a new backup node is then elected.

For this to all work properly, the SOA/data grid must address reliability issues that need to be built into the network access protocol for a cluster based on sophisticated positive and negative acknowledgments (ack/nack). Locking and concurrency measures must be taken to avoid collisions on multi-node access and updates, and each node in the grid must be aware of the existence and status of other nodes in the grid. These and additional, more advanced configurations beyond the primary and backup nodes are discussed here to help deliver continuous, predictable performance, regardless of load or server outages.

**What About Database Updates?**

All the data caching we have been discussing so far has been in the form of high-speed, in-memory storage and access with continuous availability. These approaches don’t need to rely on the use of traditional database technology, but what if you want to support database updates anyway? For example, you may need to generate regular reports against live data, or you might just want to have an extra layer of reliability and longer-term storage.

The whole point of in-memory access is to avoid the overhead of database persistence, which is known to become a bottleneck under peak processing loads. To address this issue, an SOA grid can make use of asynchronous write-behind queues, which eventually update to a database (see Figure 2). We stress the words “asynchronous” and “eventually” here because the operation of updating the database should not interfere or block the update of the grid that is holding the state data for the application or service.
Load Balancing Stateful Services

A common means of scaling up an SOA to support increased demand is to deploy multiple instances of a service and use a load balancer to dispatch requests and spread out the service request traffic. Typically this approach works well if the services are written to be completely stateless. By this we mean that there is no “stickiness” between service invocations because there is no local data or service state that needs to be held by the service from one invocation to the next.

But what if the state of a service is required to live beyond a single invocation? Where do you put that state data? The answer generally lies in providing service affinity by making the service request itself “sticky.” This means ensuring that all subsequent requests within the same logical conversation (such as callback operations) get routed to the same instance of a service, perhaps even isolating that service from other consumer requests. Alternatively, it may require that all possible state values get packaged up in one service request and passed around on the wire. Both approaches can prove to be costly and cumbersome to implement.

In an SOA grid, as the load balancer invokes arbitrary instances of a particular service, the service always locates and accesses the most up-to-date values for the instance data it is managing as its transient state, regardless of which service the load balancer happens to delegate the request to. As illustrated in Figure 3, the enterprise service bus (ESB) is performing the task of load balancing the requests between an instance of Service A and Service A1. The request to Service A updates a data object that is transparently maintained by the SOA grid. The subsequent request to Service A1 accesses (again, transparently) that same piece of data.
Continuous Availability of Stateful Services

Likewise, if a particular service instance fails, the state data for that service is still available and up-to-date when an alternate service instance is invoked. In Figure 4, Service A1 is the backup for Service A. When Service A fails, the ESB automatically routes to Service A1.
Because the SOA grid is designed to maintain a single version of the “true” instance data, the backup Service A1 does not have to recover from anything. The state data is already there!

**Optimizing SOA Infrastructure and the BPEL Dehydration Store**

So far we have been talking mostly about what the grid can do for your services. But what about the SOA infrastructure itself? Parts of the SOA infrastructure, such as the ESB and BPEL servers, can also be leveraged to enhance performance throughput, scalability, and availability. Let’s look at BPEL dehydration as an example.

A well-known issue with any business process is the slowdown it is subject to when it takes a checkpoint of its current state to a database. This is typically required for recoverability purposes and during outbound asynchronous invocations. As a result, the more complex the process, the more checkpoints are required, and thus the higher the overall latency of the process execution. An SOA grid can be substituted to store the checkpoint data to provide the following advantages:

- faster storage and retrieval times of the process state
- easier distribution of the state if callback or recovery occurs on a separate node of the cluster

**An Example: Data Grid Infrastructure in Action**

The following is a basic but powerful use case demonstrating how data grid infrastructure can provide fast and reliable distributed memory caching for a service-oriented solution.

A BPEL engine orchestrates the interaction between multiple service invocations. Depending on what is described in the BPEL process definition, the BPEL engine might apply flow constructs to sequentially invoke services, or it might lie dormant and listen for an event from an external party. In BPEL parlance, this includes receive, invoke, wait, and onAlarm.

During these periods, the BPEL engine will temporarily rest (dehydrate) the process, by writing its context out to a
persistent storage, referred to as a “dehydration store.” This process context might include the message payload of the service requests that the process is currently orchestrating, which can be quite large. In addition, the process context will include the values of the following:

- any state variables global or local scopes that might have been defined
- internal information used by the BPEL engine (such as the current instruction pointer into the process flow and transaction scoping information)

Typically, the persistence mechanism for the dehydration store is implemented using a relational database. Under heavy volume with large message payloads, this can become a bottleneck for processing (no matter how good the database is). A solution for this is to use the SOA grid as the backing for the BPEL dehydration store, rather than using the database directly (see Figure 5).

![Figure 5: BPEL dehydration using an SOA grid.](image)

**Data Serialization Versus Process Logic Relocation**

Certain advanced features found in an SOA grid introduce some interesting architectural design decisions. For example, one such feature allows remote nodes to cache not just the shared data objects but also the programming logic to operate on the cached data. This essentially enables more of a logic co-location versus a data fetch model. One of the main benefits of logic co-location is that it allows the process logic to operate on the node in which the data resides, thus removing any data serialization that would occur if the service were to fetch into local memory the data from a remote node. This is more advantageous for operating on larger data objects because it reduces network traffic and serialization, which can be expensive.

**Re-locatable BPEL Processes**

If we were to take this discussion one step further and apply this trade-off to the SOA infrastructure, we can introduce the notion of a re-locatable BPEL process: a BPEL process that dehydrates and then rehydrates itself somewhere else on the grid to execute closer to the service and instance data that it is operating on.

There are several reasons for wanting to do this:

1. The service state is rather large and would be costly to serialize across the grid.
2. The BPEL process is calling into an adapter to an ERP system that is only hosted on a particular machine, and co-locating the BPEL process with that adapter will reduce the amount of network overhead of messages between the BPEL server and the adapter/ERP system.

As shown in Figure 6, a lightweight notification/signaling mechanism is used to trigger the appropriate node on the SOA grid to activate and rehydrate the BPEL process.

![Figure 6: Re-locatable BPEL processes rehydrate themselves where needed.](image)

The net result of this is complete flexibility over where BPEL processes get executed without the usual hub-and-spoke network traffic between the BPEL engine and the services it may need to operate on. Furthermore, the network latency involved in maintaining primary and secondary grid memory management is dramatically reduced for large data objects.

### 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.

Stateful services are the primary benefactor of the SOA grid because the requirement to persist service state is a primary bottleneck - not just in performance, but also in recoverability. In short, the SOA grid helps you attain a much faster response time with more predictable QoS. It can accomplish this with minimal configuration overhead using self-healing functionality to automatically adjust to server outages. The addition of an SOA grid as part of your fundamental infrastructure will eliminate numerous traditional inhibitors and help you realize a successful implementation and, ultimately, drive the aggressive and accelerated adoption of service-oriented computing initiatives.

*The authors would like to thank Jonathan Purdy of the Oracle Coherence team for his contributions to this article.*