The Fractal Nature of Web Services

Christoph Bussler
Cisco Systems

Applying SOA concepts to the runtime structure can be problematic.

Conventional wisdom holds that the service-oriented architecture approach is the silver bullet for all IT problems nowadays. According to this view, SOA will lead to near-perfect applications in which every function is implemented as a service, and a service can call any other service to implement its functionality. This includes not only services that provide business functionality, but also nonfunctional services for logging, monitoring, data transformation, and so on.

However, if every service is free to call every other service remotely, many remote invocations that often are nontransactional can freely mix synchronous and asynchronous connections at runtime. This "fractal" situation is extremely fragile and demands close examination, leading me to issue a warning—"naïve SOA considered harmful"—akin to that in Edsger Djikstra's seminal paper, "Go To Statement Considered Harmful" (*Comm. ACM*, Mar. 1968, pp. 147-148).

SOA'S PROMISE

Services in their simplest form involve two parties: a provider that exposes and provides services, and a requester that uses services to achieve its goals (for simplicity, I equate services with Web Services Description Language operations). A service provider can serve many service requesters; a service requester can utilize many service providers. Execution concurrency is therefore inherent in the concept and implementation of services.

As service providers and requesters typically reside in their own computing environment, their data and execution state are independent of each other, making services in general a heterogeneous, autonomous, and distributed system. SOA reduces the common technology between service providers and requesters to a shared communication infrastructure as well as a widely accepted service interface language and message binding at execution time.

This independence leads to talk about composable systems somewhat akin to Lego blocks (J. Bloomberg, "The Lego Model of SOA," ZapThink, 11 Dec. 2006; www.zapthink.com/report.html?id=ZAPFLASH-20061212). Service composition enables service requesters to invoke existing services provided by different service providers in a specific order to obtain a result that no single service would be able to offer on its own.

Providers can achieve service composition by programming language code or by explicitly defining the service invocation order, preferably in a declarative approach. A predominant standard in this space is the Business Process Execution Language, which developers use to implement service composition.

Declarative modeling enables businesses to adapt to changing needs by loosely coupling services. At any point, the service composition, as well as the service providers used in the composition, can be changed without altering the service requester's IT infrastructure; only the declarative composition definition requires adjustment. This in turn lets businesses quickly modify their internal operations to adjust to market activity without being held back by internal IT development projects or long software vendor release cycles.

SOA thus ideally makes it possible for businesses to achieve sufficient performance, reliability, and dependability on a system level. Uniformity in service design and implementation leads to a pattern in which "everything should be a service"—not only the implementation of business functionality, but also system functionality like logging, monitoring, or data transformation.

ONE POSSIBLE REALITY

A world full of services that can call each other freely, while remaining flexible and performing at a high level, is certainly desirable for many enterprises. However, reality is not like this—yet.

While a business may choose to implement many services from scratch, others are likely already in place, at least in principle, through packaged applications like enterprise resource planning systems. In this case, the task is not to implement services but to "wrap" or abstract from the packaged application implementation and make its functionality available as services.

Some enterprises take a standardized approach to service invocation that dictates all communication between services occurs over asynchronous messaging middleware like a queuing system. This means that if one service invokes another, the invocation will take the form of a request message and a return message in various request and response queues inside the system.

Due to overall IT system complexity, a logging and monitoring infrastructure is essential from both a system- and business-monitoring viewpoint. This infrastructure supports the need to examine the processing state at any point as well as postmortem analysis in case of a failure.

As the SOA approach is independent of a particular business domain, a generic logging service and a generic monitoring service can be implemented for all services. All services invoke these nonfunctional services to provide timely and sufficient runtime information.

Because not all services are written from scratch, but also encapsulate existing systems, data mismatches between services must be addressed through a process that transforms data types and models into one another without modifying or losing the data semantics. Such data transformation can be implemented as another nonfunctional service (R. Schmelzer and J. Bloomberg, "The Role of Transformation Services in SOA," ZapThink, 3 Oct. 2006, www.zapthink.com/report.html?id=ZAPFLASH-2006103).

A pattern thus emerges in which every business and service functionality is implemented as a service in itself, sharing the same communication infrastructure and following the same service implementation principles.

MICROANALYSIS

Consider a classic single-service invocation in which S1 provides an input message to service S2 and expects a result back from service S2 after the latter finishes executing. If nothing else were to occur, this request-reply scenario results in two queue messages, one representing the request of S1 to S2 and a second representing the result from S2 to S1.

Each message must be put into and retrieved from a queue (or two, a request queue and a response queue), resulting in four queue operations. In the worst-case scenario of the queuing system being in a separate computing environment, these four queue operations would result in four remote invocations.

If the queuing system is not only persistent but transactional, each queue operation is a transaction. In general, as the services internally keep state, enqueuing or dequeuing a message becomes a distributed transaction across the services and the queuing system, with the messages stored on disk.

Service requester and service provider data models often differ.

In summary, then, a service invocation thus far results in four distributed and remote transactions.

Service requester and service provider data models often differ. For example, a service requester might deal with purchase orders defined by Electronic Data Interchange standards while a service provider might implement purchase orders according to RosettaNet specifications. In this case, either the service requester or provider must transform the data type from EDI to RosettaNet (and back for the acknowledgment messages).

Data transformation is implemented as a service that must be called twice: once for the request message and once for the response message. As each request-reply service invocation results in four invocations, the two data transformation invocations add eight remote invocations; in this case, the additional remote invocations are nontransactional because data transformation is typically idempotent.

Summarizing again, we now have four distributed, remote transactions plus eight remote invocations for data transformation for the scenario in which S1 calls S2.

Each service follows a bare-minimum logging and monitoring strategy. A service requester logs a service invocation before and after the invocation, while a service provider logs its invocation right after the invocation starts and right before the invocation finishes. Logging information includes parameter values, the invocation context, and other elements.

Each logging is a one-way service invocation, resulting in a total of eight remote invocations: four by the service requester and four by the service provider, two for each queue operation. The same is true for the monitoring service. Thus, an additional eight remote invocations occur.

In conclusion, the scenario in which S1 calls S2 involves four distributed transactions and 24 remote invocations. The process generates a total of 14 messages: two for the invocation and result data, four for the two transformation invocations, four for the four logging invocations, and four for the monitoring invocations. This is considerable effort for a simple request-reply invocation.

MACROANALYSIS

When considering service composition, this amount of effort must be multiplied by the number of services invoked by the composition. The invocation thus exhibits a fractal structure—that is, at every level of detail, the structure repeats itself (http://en.wikipedia.org/wiki/Fractals).

Service invocation requires logging and transformation. Transformation in itself might do some logging for its own purposes. In general, every service might call other services, and those calls are completely hidden behind the service interface definitions.

With a high number of transactions, remote invocations, and persistent messages, infrastructure characteristics dominate performance. Every additional service invocation will impact

throughput as it will cause additional remote invocations and persistent data store accesses.

Error recovery becomes another big burden for the naïve service-implementation scenario. Every time an invocation breaks, independent of the reason, at least one asynchronous connection is left hanging; this means, for example, the invocation takes place—the message was submitted—but the result message is not picked up.

This also leads to inconsistent data states, as the overall service invocation fails. However, the data states advance; asynchronously triggered functionality cannot be rolled back through transaction demarcation.

With asynchronous communication, compensation becomes a necessity. If the asynchronous queuing system is nontransactional, then any compensation must first determine if a consistent state has been reached, leading to the implementation and invocation of idempotent services.

Services that exist as independent concepts at design time are implemented as independent execution entities at runtime. Assuming that the conceptual system structure is equally useful during execution is a naïve and potentially dangerous mistake.

Thinking about overall system structure in terms of independent services makes perfect sense given the paradigm of functionality containment and loose coupling. However, applying service concepts to the runtime structure causes many difficult problems and can lead to a very complex system.

Instead, applying high-performance transaction system design criteria that optimize for runtime properties like performance, throughput, and resiliency should be paramount. "Think SOA, implement HPTS" is an approach consistent with the Organization for the Advancement of Structured Information Standards SOA Reference Model (www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm), which clearly separates SOA concepts from implementation technology. ■

Christoph Bussler is a member of the technical staff at Cisco Systems. Contact him at cbussler@cisco.com.

Editor: Simon S.Y. Shim, Department of Computer Engineering, San Jose State University; sishim@email.sjsu.edu