

## Feature Article



### Understanding Service Composition, Part IV: Dealing with Events

by David Chou

Published: July 11, 2010 (SOA Magazine Issue XLI: July 2010)

[Download PDF](#)

[Digg This](#) • [De.licio.us](#) • [Slashdot](#) • [Technorati](#) • [StumbleUpon](#) • [Google Bookmark](#)

#### Introduction

Many service-oriented architecture efforts today are focusing on implementing synchronous request-response interaction patterns (sometimes using asynchronous message delivery) to connect remote processes in distributed systems. While this approach works for highly centralized environments, and can create loose coupling for distributed software components at a technical level, it tends to create tight coupling and added dependencies for business processes at a functional level. Furthermore, in the migration towards real-time enterprises, which are also constantly connected and always available on the Web, organizations are encountering more diverse business scenarios and discovering needs for alternative design patterns in addition to synchronous request-driven SOA.

Event-driven architecture (EDA, or event-driven SOA) is often described as the next iteration of SOA. It is based on an asynchronous message-driven communication model to propagate information throughout an enterprise. It supports a more natural alignment with an organization's operational model by describing business activities as a series of events, and does not bind functionally disparate systems and teams into the same centralized management model. Consequently, event-driven architecture can be a more suitable form of SOA for organizations that have a more distributed environment and where a higher degree of local autonomy is needed to achieve organizational agility.

So, what do we mean by events? Well, events are nothing new. We have seen various forms of events used in different areas such as event-driven programming in modern graphical user interface (GUI) frameworks, SNMP and system notifications in infrastructure and operations, User Datagram Protocol (UDP) in networking, message-oriented middleware (MOM), pub/sub message queues, incremental database synchronization mechanisms, RFID readings, e-mail messages, Short Message Service (SMS) and instant messaging.

In the context of SOA however, an event is essentially a significant or meaningful change in state. And an event in this perspective takes a higher level semantic form where it is more coarse-grained and abstracted into business concepts and activities. For example, a new customer registration has occurred on the external Website, an order has completed the checkout process, a loan application was approved in underwriting, a market trade transaction was completed, a fulfillment request was submitted to a supplier, and so on. Fine-grained technical events, such as infrastructure faults, application exceptions, system capacity changes, and change deployments are still important but are considered more localized in terms of scope, and not as relevant from an enterprise-level business alignment perspective.

As event sources publish these notifications, event receivers can choose to listen to or filter out specific events, and make proactive decisions in near real-time about how to react to the notifications. For example, update customer records in internal systems as a result of a new customer registration on the Website, update the billing and order fulfillment systems as a result of an order checkout, update customer account as a result of a back office process completion, transforming and generating additional downstream events, and so on.

Event-driven architecture is an architectural style that builds on the fundamental aspects of event notifications to facilitate immediate information dissemination and reactive business process execution.

In an event-driven architecture, information can be propagated in near real-time throughout a highly distributed environment, and enable the organization to proactively respond to business activities. Event-driven architecture promotes a low latency and highly reactive enterprise, improving on traditional data integration techniques such as batch-oriented data replication and posterior business intelligence reporting. Modeling business processes into discrete state transitions (compared to sequential process workflows) offers higher flexibility in responding to changing conditions, and an appropriate approach to managing the asynchronous parts of an enterprise.

## Key Concepts

The way systems and business processes are connected in an event-driven architecture represents another paradigm shift from request-driven SOA. Here, we will discuss some of the fundamental aspects.

### *Autonomous Messages*

Events are transmitted and communicated in the form of autonomous messages — each message contains just enough information to represent a unit of work, and to provide decision support for notification receivers. Each event message also should not require any additional context, or any dependencies on the in-memory session state of the applications; it is intended to communicate the business state transitions of each application, domain, or workgroup in an enterprise.

For example, an externally-facing Web application publishes an event that a new customer registration was completed with these additional attributes: timestamp, ID=123, type=business, location=California, email=jane@contoso.com. As well as, information that is relevant to the rest of the enterprise, and supports any process-driven synchronous lookups for additional details to published data services from distributed systems.

### *Pub/Sub Model for Loose Coupling*

Instead of the traditional pull-based synchronous request/response or client/server RPC-style interaction models, event-driven architecture builds on the pub/sub model to push notifications out to interested listeners, in a fire-and-forget (does not block and wait for a synchronous response), uni-directional, asynchronous pattern.

This asynchronous message-based interaction model further encapsulates internal details of distributed systems in an environment. This form of communication does not need to be as concerned with some aspects of request-response models such as input parameters, service interface definitions such as WSDL in SOAP-based communication and hierarchical URI definitions in REST-based communication, and fine-grained security. The only requirement is a well-defined message semantic format, which can be implemented as plain old XML-based (POX) payload.

The event-driven pattern also logically decouples connected systems, compared to the technical loose coupling achieved in the request-driven pattern. That is, functional processes in the sender system, where the business event took place, do not depend on the availability and completion of remote processes in downstream distributed systems. Whereas in a request-driven architecture, the sender system needs to know exactly which distributed services to invoke, exactly what the provider systems need in order to execute those services, and depends on the availability and completion of those remote functions in order to successfully complete its own processing.

Furthermore, the reduced logical dependencies on distributed components also have similar implications on the physical side. In synchronous request-driven architectures, connected and dependent systems are often required to meet the service-level requirements (for example, availability and throughput) and scalability/elasticity of the system that has the highest transaction volume. But in asynchronous event-driven architectures, the transaction load of one system does not need to influence or depend on the service levels of downstream systems, and allows application teams to be more autonomous in designing their respective physical architectures and capacity planning.

Furthermore, due to the overall reduced dependencies, changes to each connected system can be deployed

more independently (and thus more frequently) with minimal impact to other systems.

In the case of an externally facing Web application that has just completed a new customer registration process, that information needs to be communicated to an internal customer management system. The Web application's only concern is to publish the event notification according to previously defined format and semantics - it does not need to remotely execute a service in the customer management system to complete its own processing or bind the customer management system into a distributed transaction to make sure that the associated customer data is properly updated. The Web application leaves the downstream customer management system to react to the event notification and perform whatever it needs to do with that information.

### *Event-Driven Activation*

The event-driven pattern shifts much of the responsibility of controlling flow away from the event source (or sender system), and distributes/delegates to event receivers. Event source systems do not need to explicitly define the scope of distributed transactions and manage them, or connected systems are less dependent on centralized infrastructure components such as transaction process systems (TPS), business process management (BPM), and enterprise service bus (ESB) products. The connected systems participating in an event-driven architecture have more autonomy in deciding whether to further propagate the events, or not. Instead of burdening the transaction-initiating sender systems with that knowledge, the knowledge used to support these decisions is distributed into discrete steps or stages throughout the architecture and encapsulated where the ownerships reside.

For example, in the new customer registration event scenario, the customer management system that receives the event notification owns the decision on executing a series of its own workflows, populating its own database of this information, and publishing a potentially different event than may be of interest to the marketing department. In a more independent and collective model, all these downstream activities rely on decisions owned and made at each node.

### Aspects of Implementation

At a high level, event-driven architecture models, business operations and processes use a discrete state machine pattern, whereas request-driven architecture tend to use more structurally static sequential flows. This results in some fundamental differences between the two architectural styles, and how to design their implementation approaches. However, the two architectural styles are not intended to be mutually exclusive; they are very complementary to each other as they address different aspects of operational models in an organization. Furthermore, it is the combination of request-driven and event-driven architectures that makes the advanced enterprise capabilities achievable. Here, we will discuss some of the major consideration areas, from the perspective of building on top of existing service-oriented architecture principles.

### *Event Normalization and Taxonomy*

First and foremost, event-driven architecture needs a consistent view of events, with clearly defined and standardized taxonomy so that the events are meaningful and can be easily understood—just as when creating enterprise-level standards of data and business process models, an enterprise-level view of events needs to be defined and maintained.

### *Process Design and Modeling*

Modeling business processes into discrete event flows is similar to defining human collaboration workflows using state machines. A key is to distinguish between appropriate uses of synchronized request-driven processes or asynchronous event-driven processes.

For example, one organization used the Synchronous Request-Driven pattern to implement a Web-based user registration process, in terms of populating data between multiple systems as a result of the registration request, - the implementation leveraged an enterprise service bus solution and included three systems in a synchronized distributed transaction. The resulting registration process waited until it received "commit successful" responses from the two downstream systems, and often faulted due to remote exceptions. It was a non-scalable and error-prone solution that often negatively affected end users for a seemingly simple account-registration process. On the other hand, as discussed previously, this type of scenario would obtain

much better scalability and reliability if an asynchronous, event-driven pattern was implemented.

However, if in this case the user registration process requires an identity proofing functionality implemented in a different system, in order to verify the user and approve the registration request, then a synchronous request-response model should be used.

As a rule of thumb, the Synchronous Request-Driven pattern is appropriate when the event source (or client) depends on the receiver (or server) to perform and complete a function as a component of its own process execution and is typically focused on reusing business logic. On the other hand, the Asynchronous Event-Driven pattern is appropriate when connected systems are largely autonomous, not having any physical or logical dependencies; typically focused on data integration.

### *Communication*

Event-driven architecture can leverage the same communication infrastructure used for service-oriented architecture implementations. And just the same as in basic SOA implementations, direct and point-to-point Web services communication (both SOAP and REST-based) can also be used to facilitate event-driven communication. And standards such as WS-Eventing and WS-Notification, even Real Simple Syndication (RSS) or Atom feeds can be used to support delivery of event notifications.

However, intermediaries play a key role in implementing an event-driven architecture, as they can provide the proper encapsulation, abstraction, and decoupling of event sources and receivers, and the much-needed reliable delivery, dynamic and durable subscriptions, centralized management, pub/sub messaging infrastructure, and so on. Without intermediaries to manage the flow of events, the communication graphs in a point-to-point eventing model could become magnitudes more difficult to maintain than a point-to-point service-oriented architecture.

### *Event Harvesting*

The use of intermediaries, such as the modern class of enterprise service bus solutions, message-oriented middleware infrastructure, and intelligent network appliances, further enhance the capability to capture in-flight events and provide business-level visibility and insight. This is the basis of the current class of business activity monitoring (BAM) and business event management (BEM) solutions.

### *Security*

Security in event-driven architecture can build on the same technologies and infrastructures used in service-oriented architecture implementations. However, because of the highly decoupled nature of event-driven patterns, event-based security can be simpler as connected systems can leverage a point-to-point, trust-based security model and context, compared to the end-to-end security contexts used in multistep request-driven distributed transactions.

### *Scope of Data*

So, how much data should be communicated via events? How is this different from traditional data replication? Data integration is intrinsic to event-driven architecture, but the intention is to inform other system occurrences of significant changes, instead of sending important data as events over the network. This means that only the meaningful data that uniquely identifies an occurrence of a change should be part of the event notification payload, or in other words, just an adequate amount of data to help event receivers decide how to react to the event. In the example discussed earlier with the customer registration event, the event message does not need to contain all user data elements captured/used in the customer registration process, but the key referential or identifying elements should be included. This helps to maintain clear and concise meaning in events, appropriate data ownerships, and overall architectural scalability. If downstream systems require more data elements or functional interaction to complete their processes, alternative means, such as synchronous Web services communication, can be used to facilitate that part of the communication.

### *Conclusion*

Event-driven architecture is based on an asynchronous message-driven communication model to propagate information. It supports a more natural alignment with an organization's operational model by describing business activities as a series of events. Event-driven architecture promotes a low latency and highly

reactive enterprise, improving on traditional data integration techniques such as batch-oriented data replication and posterior business intelligence reporting.

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Glossary](#) [SOA School](#) [SOA Books](#) [About/RSS](#)  
[Legal](#)

Copyright © 2006-2010  
SOA Systems Inc.