CHAPTER 2

Services

In this chapter we continue our SOA overview with a focus on services. A service includes the following aspects:

- a service implementation
- elementary access details
- a contract

A *service implementation* is the core business logic, which might be written in Java, COBOL, Enterprise Generation Language (EGL), or any other programming language.

Elementary access details include the *location*, which is an address where the service implementation resides, and the *binding*, which identifies the transport protocol (such as HTTP or Java Message Service) that formats a message at the start of transmission and unformats it at the end of transmission. Formatting occurs when the invocation message originates at the requester; in that case, unformatting occurs when the message arrives at the service location. Formatting also occurs if the service issues a response; in that case, unformatting occurs when the response arrives at the requester.

A *contract* describes the service's intended behavior and is independent of the implementation details. The contract includes two elements: a service interface and a Quality of Service.

The service interface provides a description of the data that can pass between a requester and a service, along with details on each operation the service provides. The interface includes information on the messages and answers the questions, "What is the format of a message (for example, two strings followed by an integer)?" and "What are the restrictions on content?" The interface also includes details on the message exchange patterns, which indicate how the requester and service interact. "Does the service always respond to the requester?" "Can the service do a task without reporting back?"

Some aspects of the service's behavior are implicit in the service interface. For example, a service might provide a stock quote but return an error message if the submitted stock symbol is invalid.

An interface is an aspect not only of a service but also of a high-level design for the service. The interface precedes the implementation in most cases, and the service implementation (or sometimes the service as a whole) is said to *implement* the interface.

The service's *Quality of Service (QoS)* is a description of interaction rules that go beyond those implied by either the elementary access details or the service interface. For example, the service might require that the invocation message include authorization details that prove the user's right to use the service. You'll learn more about QoS later in the chapter.

One note about the service contract: Rather than "contract," we could have used the phrase "proposed contract" because a requester and service may undertake a negotiation, possibly at run time, to determine certain details of the interaction. We expect this kind of negotiation to become more prevalent as time goes by. For further details, see the description of the WS-Agreement specification in Appendix A.

Last, the terms "service" and "service implementation" are often used interchangeably. The second term is most appropriate when the focus is on the details of the business logic.

Loose Coupling

As you learned in Chapter 1, an important characteristic of an SOA is *loose coupling*, which means that one unit of software is largely independent of another. This independence implies that changes to one unit of software cause less turmoil and cost to an organization than when the software is more interdependent. It also means you can substitute one unit of software for an already-deployed unit relatively easily. The value of loose coupling is greatest when technical changes are expected over time.

The following questions identify some of the ways in which a service might be loosely coupled with other software.

How easily can the logic inside a service be revised without changing how the service is invoked?

The logic and programming language of a service implementation should be independent of the service contract. You can change the service internals for greater efficiency without changing the requester in any way.

How protected is the requester from disruption in the face of increased service capabilities?

An SOA runtime product may support either of two kinds of service interface:

- *Remote procedure call (RPC):* In this case, the requester submits a set of arguments to a particular service operation as if invoking a local function. The service uses each argument as a discrete unit in accordance with the meaning and type of the related parameter.
- *Document:* In this case, the requester submits a string of arbitrary length, and the service reviews that string to determine what operations to perform.

The RPC and Document categories overlap, as when an RPC invocation submits a single string to a service that in turn dispatches the message to one of several subroutines. The point, however, is that if the contract between requester and service features a long string (rather than a set of typed arguments) and if a later version of the service adds new functionality, the service interface is unaffected.

Requester updates (as needed to use the new service functionality) are likely to be needed only over time rather than in urgent response to a change in the service.

How easily can a service be incorporated into a larger process without changing the service implementation?

A change to a runtime security mechanism, for example, shouldn't require the code for a service to be rewritten or recompiled. The looseness of coupling in this case may depend on the power of an SOA runtime product because that product allows more decisions to occur at configuration time.

To what extent is a requester dependent on service availability?

If an SOA runtime product maintains message queues on each side of a remote transmission, the product can guarantee message delivery between requester and service, in which case network failures will tend to have less of an effect. The benefit is greatest if the requester isn't waiting for a response.

Can the requester continue running in the absence of a response?

If the requester can invoke a service and continue running, the requester is less dependent on service availability.

Is the service dependent on state information?

A specific requester might invoke the same service repeatedly to fulfill a single task (to update a checking account, for example), and the service might need to retain *state information* (such as a checking-account number) between each invocation. In general, state information is the set of values that are needed for the service to maintain a conversation with a specific requester.

If a service needs to retain state information, the SOA runtime product won't be able to direct processing to an identical service at a second location, as might be necessary in response to network traffic. If the SOA runtime product can redirect the state information as well, the restriction does not apply.

Service Registry

A requester must be able to reference a service's access and proposed contract details, which are often available in an online registry. Such a registry often conforms to the rules of Universal Description, Discovery, and Integration (UDDI), as described in Chapter 5.

A company can create its own registry for internal use and can create additional public registries, often in concert with other companies in the same industry. In addition, the company SAP AG operates a public UDDI registry.

In theory, the requester can retrieve the registered information at run time; however, this kind of programmatic retrieval is rare in practice. In most cases, the registered information is retrieved earlier. The registered information may be available to the requester at development time, as reflected in the requester's code or in the requester's call to a library that contains the details. The library may be written either before or after the requester is developed.

Alternatively, the registered information may be available to a network administrator or other professional who configures and then deploys the requester. This availability gives an organization greater flexibility because the selection of a service can occur relatively late.

Chapter 9 describes configuration-time opportunities that are available when you're working with Service Component Architecture.

Service Level Agreements

A *Service Level Agreement (SLA)* is a document that gives human readers the information necessary to decide how and whether to invoke a particular service from other software. The presence and use of an SLA varies by SOA vendor and corporate user. If present, the SLA

- includes elementary-access and proposed-contract details in most cases
- may be written by software designers to help negotiate what functionality is to be included in a given service
- may communicate plans to potential service users and other interested parties

- may be the basis of a legal document that indicates what level of reliability the service offers (for example, how many hours per week the service is available)
- may be used as an input to an automated process that creates invocation details for use when developing a requester

Message Exchange Patterns

A service supports one or more *message exchange patterns (MEPs)*, or kinds of interaction between requester and service. At this writing, only two elementary MEPs are widely used.

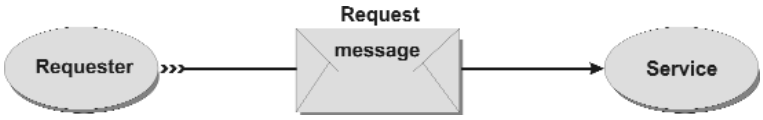

Figure 2.1: One-way pattern

Figure 2.1 depicts a *one-way* pattern (sometimes called *in-only* or *fire-and-forget*), in which the requester invokes the service with a request (an input message) but does not receive a response.

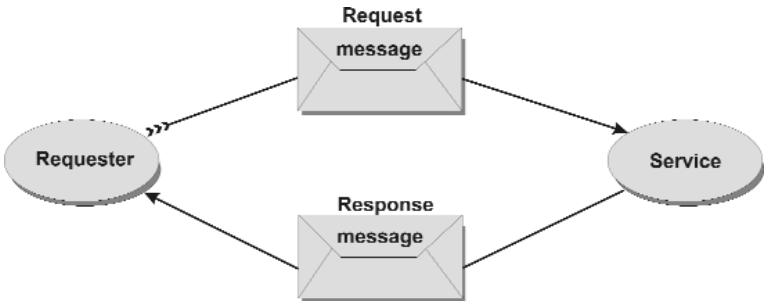

Figure 2.2: Request-response pattern

Figure 2.2 depicts a *request-response* pattern (sometimes called *in-out*), in which the requester invokes the service with a request and receives a response.

Two other elementary MEPs are uncommon but will be supported over time.

Figure 2.3: Notification pattern

In a *notification* pattern (sometimes called *out-only*), the service submits a message in the absence of an ongoing conversation, as when a service sends news to a subscriber. Figure 2.3 illustrates this pattern.

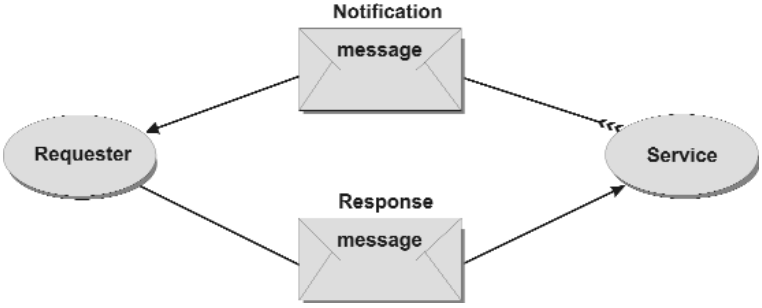

Figure 2.4: Solicit-response pattern

Figure 2.4 depicts a *solicit-response* pattern (sometimes called *out-in*), in which the service submits a notification to a requester and receives a response.

The following advanced features are available for the first two MEPs and will be available for the others in the future:

- A message in the returning direction can be optional.
- A service or requester can submit a *fault message*, which is data sent in response to a runtime error.

Synchronous and Asynchronous Communication

The request-response pattern and the solicit-response pattern represent *synchronous* communication, which means that the initiator of the message suspends processing until the initiator receives a response. In contrast, the one-way and notification patterns represent *asynchronous* communication, which means that the initiator of the message continues running the next coded statement, without suspending processing.

Callbacks

Assume that Service01 invokes Service02. To complete the interaction, Service02 may issue a *callback*, which is an invocation of an operation that can be provided by Service01. The invoked operation is called a *callback operation*, as shown in Figure 2.5.

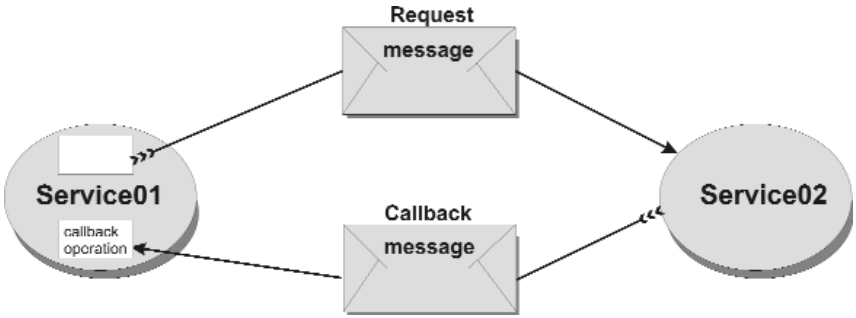

Figure 2.5: Request and callback

Service02 could be a credit-rating service that can respond to a request for customer details only after an hour, while Service01 is a service that requests credit details. Service01 shouldn't wait for the details, because the wait would use memory and other resources. Instead, the overall business process can ensure that Service02 invokes a callback operation in Service01.

A callback doesn't necessarily involve asynchronous processing, although the two ideas often come together. Consider the following variations:

- Service01 requests a credit report and receives a confirmation of the request, then waits for the report as before.
- Service01 allows Service02 to issue synchronous callbacks that request additional details.

From a business point of view, a callback operation is like any other. Service01 can have multiple operations, and the callback operations in Service01 are those invoked by services that previously received requests from Service01.

Quality of Service

A service interface defines the interaction between a service and a requester in a narrow sense, but the interaction can have many additional characteristics. In some contexts, Quality of Service refers only to reliability guarantees, such as the percentage of time a service is promised to be available. In our view, however, QoS refers to all runtime processing aspects that go beyond the service interface.

QoS also includes advanced aspects of runtime processing:

- reliability guarantees
- security mechanisms
- service coordination, including transaction control
- runtime update of address, binding, and message content

A particular service may be offered with different QoS characteristics, as when a company charges a different fee in exchange for a different level of reliability.

In describing the QoS issues, we hope to give you a sense of the flexibility and power of an advanced SOA runtime product.

Reliability Guarantees

Reliability guarantees may be described in a Service Level Agreement. The guarantees can be affected by the quality of the network hardware, and most are quantitative. For example:

- During what percentage of time will the service be available, or during what hours?
- What is the expected *throughput* — the number of messages to which the service will respond in a given duration?
- What is the expected *latency period* — the waiting time between a service request and response?
- What is the probability of a successful response within the latency period?
- Is message receipt assured, even if the network fails?

Security Mechanisms

Security mechanisms are often affected by features of the SOA runtime:

- *Authentication:* How does the SOA runtime ensure that a message is from a specified requester?

- *Authorization:* How does the SOA runtime ensure that a specified requester is allowed to access a specified service?
- *Confidentiality:* How does the SOA runtime ensure that the content of a message isn't viewed during transmission?
- *Integrity:* How does the SOA runtime verify that a given message was unchanged during transmission and was delivered with all data in the appropriate order?
- *Non-repudiation:* How does the SOA runtime verify the integrity of a given message and ensure that the message came from the specified requester? Non-repudiation proves (for example) that a party to a transaction made a promise, as in an online purchase.
- *Protection from denial-of-service attacks:* How does the SOA runtime prevent a flood of messages from reaching a given service?

Service Coordination

Service coordination concerns how services work together to fulfill a business process. Coordination takes one of two forms: orchestration or choreography.

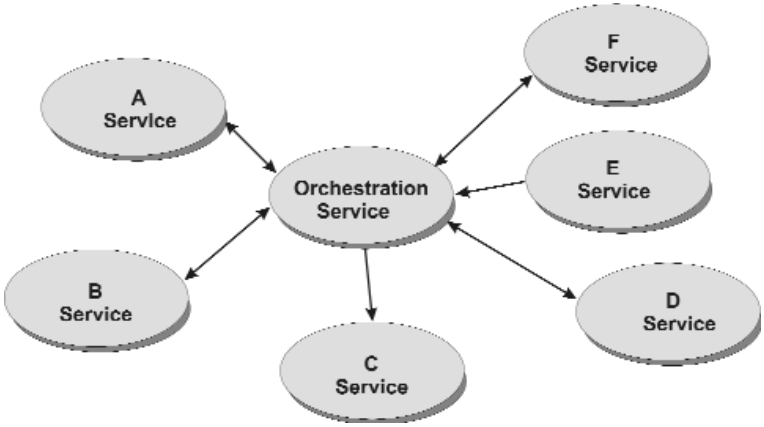

Figure 2.6: Orchestration

As Figure 2.6 illustrates, *orchestration* refers to a form of processing in which one service acts as a controlling hub in relation to other services, which act as spokes. The hub might receive a message from one spoke and make decisions based on that message, as by changing the format or content of the data and invoking some other spoke.

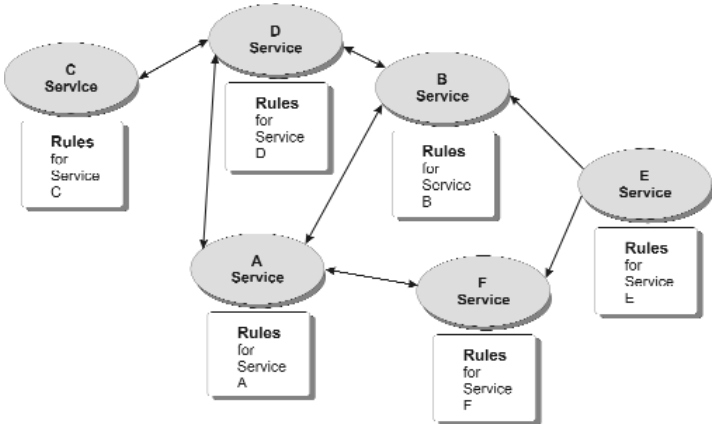

Figure 2.7: Choreography

As Figure 2.7 shows, *choreography* refers to a more decentralized form of processing, where multiple services interact without submitting messages to a hub. This sort of processing is based on rules known to each service and may be enabled by establishing a local configuration file for each service.

Often, orchestration is said to coordinate services in the same company, while choreography is said to guide the interactions of trading partners. Usage of the terms is inconsistent, and some analysts employ them interchangeably.

An important subset of service coordination is transaction control, which primarily concerns how services handle the update of databases. In general, a service may *commit* changes (ensuring that changes are permanent) or *rollback* changes (returning the database to a previous level of update). Among the QoS issues:

- Is the service allowed to revise database changes that were made (but not committed) by the requester and possibly by other services?
- Is the service prohibited from issuing a commit or rollback? A prohibition is likely if the requester is responsible for committing changes made by the service.
- Does the service depend on a *compensating service*? A compensating service is one that will be invoked only if necessary to make up for a runtime change (in this case, a database change) that is later found to be undesirable. As invoked by a coordinating service, for example, Service01 commits a

database change (to indicate that a purchase was completed); days later, the coordinating service receives a cancellation and invokes a compensating service to reverse the effect of the committed change.

Last, the term *coordination* sometimes refers specifically to a kind of orchestration that is detailed in the WS-Coordination specification, as described in Appendix A.

Runtime Update of Message Content or Destination

Some QoS issues are especially meaningful in the context of a network that handles a variety of services, computer types, and data-traffic patterns. Among the issues:

- Can the flow of traffic be changed at run time? The change usually has one of two purposes: to provide faster access to higher-priority services or to equally distribute the data being directed to services that provide the same functionality but reside at different locations.

  The process of altering data traffic to conform more closely to a performance ideal is called *load balancing*. This process might occur in response to a configuration setting or to an operator's intervention.

- Can a message be reformatted at run time — for example, to allow transmission to a computer that uses a different transport protocol? If so, a configuration setting may be involved.

- Can a service be configured to send messages to a destination other than the requester — for example, to print a runtime error or to invoke an additional service in some cases but not others? If so, either a configuration setting or details in the message itself can cause the change.

Endpoints, State, and Correlation

The language of SOA is a bit messy, with the same terms having different meanings in different contexts and in different minds. This section tells the relatively absolute truth.

An *endpoint* is a location: the addressing details that are necessary and sufficient to access or provide a service at run time. If service A is on both machine 1 and

machine 2, you can speak of two endpoints, and in this case, endpoint is synonymous with service location.

The endpoint is a step removed from the implementation. The operating system

- receives inbound data at the endpoint and presents that data to the implementation code. The implementation is said to be *listening* at the endpoint.
- accepts outbound data at the endpoint and transmits that data to a service at another endpoint.

From the point of view of a network administrator, a *service instance* is an implementation that is listening at a particular endpoint. If service A is on machines 1 and 2, two service instances are available. When a request arrives at run time, the service instance dispatches the request to a *thread of execution*, which is an operating-system facility that runs the service. If two requests arrive at the same time, each request is given its own thread.

Most services (specifically, most service implementations) are *stateless*, which means that the runtime code never relies on data from a previous invocation. A stock-quote service, for example, receives a trading symbol and returns a quote, and the data used in one invocation is independent of the data used in the next.

A *stateful* service, in contrast, sometimes needs access to values that were assigned in a previous invocation of the same or another service. The need arises because the service participates in a multi-step *conversation*, which is a sequence of invocations that constitute a relationship between the service and a requester.

In the usual situation, a requester's invocation initiates the conversation, and subsequent requests from the same requester are valid only if they arrive in a specific sequence. The service has a technology-specific way to ensure that each request is directed to the appropriate conversation rather than to a conversation occurring at the same time between the service and a different requester.

A conversation can be based on a persistent connection, but two other options are far more likely, especially in services that run for days or months. In one option, an SOA runtime product uses the input message or a system value to direct the message to the appropriate conversation. In a second option, the service

implementation itself uses the input message. In either case, *state* is a processing status — the sequential position of the last received message in the sequence of messages that are expected on the service's side of the conversation.

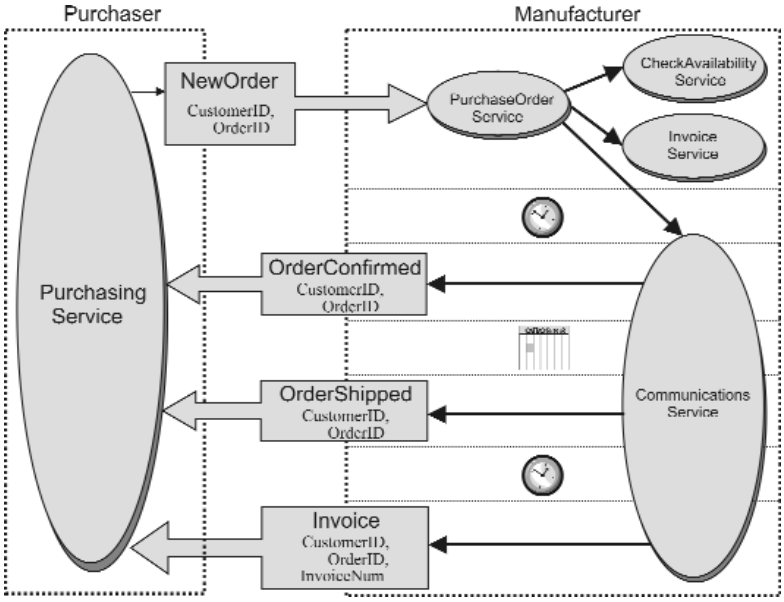

Figure 2.8: Conversation over time

Figure 2.8 illustrates a service conversation over time. As shown, a purchasing service establishes a conversation with a manufacturing service and expects a sequence of messages (confirmed, shipped, invoiced) for a specific order. When the endpoint of the purchasing service receives the confirmed message, for example, the service implementation ensures that

- the message applies to the appropriate conversation
- the state of the purchasing service changes so that at the next receipt of data from the requester, only a shipped message is considered valid

When a business analyst talks about "directing a message to the correct service instance," the meaning is really to direct a message to the appropriate conversation. We'll accept this point of view, using the term *service instance* (or *process instance*) to mean the runtime logic that handles a conversation with a specific requester.

To ensure that a message is directed appropriately, you often need to include specific IDs in the message — for example, a customer ID, an order ID, an invoice number, or some combination of identifiers. Those identifiers also *correlate* the processing done by the instance of one service (for example, the service that sends a purchase order) with the processing done by instances of other services (to confirm an order, send a shipment notification, and so on).

We say more about correlation in the chapters on Business Processing Execution Language (BPEL).