Editor: Steve Vinoski • vinoski@ieee.org

Let's Have a Conversation

Gregor Hohpe • Google

Communicating inside a single program is trivial: one method calls another, the result comes back, and the calling method continues. If anything goes wrong, an exception is thrown. If the program aborts altogether, both caller and callee share the same fate, making the interaction an all-or-nothing affair. This kind of binary outcome is a welcome behavior in the predictive world of computer software, especially one that's based on 1s and 0s.

In the land of loosely coupled distributed systems, things are a little more complicated. Using procedure call semantics for distributed systems is generally considered a bad idea, so systems should communicate in a more loosely coupled way — preferably by exchanging messages. Message-based communication results in a simple interaction model and, if used with message queues, temporal decoupling between sender and receiver: splitting the request–response interaction into two separate messages means that callers don't have to sit around waiting for response messages.

When designing such an asynchronous solution, a nagging question typically surfaces before too long: what should the caller do if the response message never comes? To be precise, this question contains two separate considerations. First, what does *never* mean? Because computer systems don't inherently understand the notion of *infinite*, "never" is generally represented as some time period that the user or application considers long enough. Second, what if never arrives and the sender still hasn't received a message? This could happen for various reasons — a glitch in the communication could cause the request or reply message to be lost, for instance, or the server could consume the request message and then crash unexpectedly without sending a reply. In any case, the sender has essentially two choices: give up or resend the original request message.

Try It Again

I actually find it kind of ironic how popular the try-it-again approach is in the generally very precise and predictive field of computer science. If something doesn't work, even the most rational computer scientist is inclined to just try it again. Even more shockingly, things often do work the second time around!

Let's assume the requestor decides to resend the request message. This brings several new considerations into play. For example, if the time out resulted from a lost response message, the service provider now receives the same request message a second time. Yet, in many cases, the provider shouldn't perform the requested action a second time — that is, it must be an *idempotent receiver*.[1] To be idempotent, a receiver must be able to distinguish a resent message from a distinct request that happens to look the same. The best way to identify messages is to equip each with a unique correlation identifier,[1] a magic number included in the message. A resent request would thus have the same correlation identifier, enabling the provider to identify it as a duplicate. The provider would then skip the requested operation and simply return the previous response message. Allowing requestors to resend messages requires that both service consumer and provider keep some state. The consumer needs to keep the request message around to be able to resend it, and an idempotent provider must keep a list of received message IDs together with the original responses.

However, the service consumer has to deal with duplicate messages as well. For example, the provider might have sent a response through a message queue just as the consumer gave up waiting. In that case, the consumer resends the request message and receives the original response a fraction of a second later. However, a second response message, based on the resent request, will arrive a

 1089-7801/07/$25.00 © 2007 IEEE

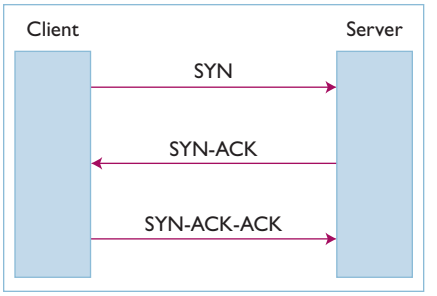

Figure 1. TCP's three-way handshake. This simple conversation defines which messages have to be exchanged in which order.

little later. Given that the consumer has already processed the first response, this message should probably be ignored, thus requiring the consumer to be idempotent as well.

So far so good, but what happens if the consumer still receives no response after resending the request message? It might be tempting to resend the message once more, but what if the response never arrived because this particular request crashed the server? Repeatedly resending such a "poison message" will just continue to crash the service provider. Therefore, the consumer should limit the number of retries and eventually give up.

Conversations

Suddenly, the interaction between service consumer and service provider looks much more complicated. Both parties have to track the interaction state, have time-out mechanisms, count the number of retries, and eliminate duplicate messages. Instead of simply invoking a method, the two systems are engaged in a conversation, an exchange of related messages over time. The concept of a conversation between services is analogous to conversations in real life. Humans frequently interact through asynchronous message exchange, as when leaving a voice mail, mailing a letter, or sending an email. These conversations can span hours, days, or months, and humans deal with many of the same issues, including duplicate messages ("I really need that TPS report"), messages crossing in transit ("Ignore this notice if you've already sent your payment"), and the need to coordinate multiple independent resources ("You got the money? You got the goods?").

Because multiple conversations tend to occur at the same time (in real life as well as in service-oriented architectures [SOAs]), messages are associated with a conversation through a correlation identifier (such as an order number) or context (the sender's name or subject line in an email, for example). When receiving a message, a participant can use these identifiers to recover the appropriate conversation's state and execute the next step, which often includes sending a follow-on message.

Describing Conversations

An expressive service should provide sufficient information to ensure smooth interaction between the provider and consumers. This includes a description of which conversations it can support. Is the consumer allowed to resend a request, for example? Does the consumer have to be prepared to receive more than one response message? More generally, in which order can service operations be invoked? A comprehensive service contract should be able to answer such questions in addition to describing the message data format (schema).

Yet, defining a conversation policy — that is, a description of all legal conversations — is nontrivial. Even my fairly simple example required many words to describe what's allowed and disallowed in the interaction. Some of those conversation rules extended beyond the sequence of messages being sent or received and touched on what the conversation partner is expected to do with a specific message (ignore duplicate responses, for instance).

Luckily, few things in the service-oriented world are entirely new, so we can expect to find some prior work that has addressed similar problems. In fact, most communication protocols include conversation rules for the communicating systems to observe. As Figure 1 illustrates, for example, TCP uses a three-way handshake to establish connections; the three-way handshake conversation defines which messages are sent, and in what order. Unfortunately, anyone who has spent time inside the network stack's transport and network layers can attest to the fact that protocol design isn't a simple matter. Given that the communicating parties typically run on different machines, issues such as message-travel delays and dead-lock situations require careful consideration. This is one reason that most protocol definitions are fairly static, well understood, and strictly implemented. In the world of application services, however, we can expect different providers to define unique application-specific conversation policies. To make matters worse, such policies will likely change over time as the services evolve.

We quickly realize that conversation policies are a valuable aspect of a service contract but can be difficult to describe. Luckily, this problem hasn't escaped the various standards and specification committees, so we have several approaches from which to learn.

Message-Exchange Patterns

Given the challenges in creating a correct conversation policy, one approach would be to simply enumerate a few common conversations and have services choose which to implement. WSDL follows this approach with the concept of message-exchange patterns (MEPs). WSDL 1.1 defines four transmission primitives, comprising sequences of input and output operations: one-way, request–response, solicit–response, and notification.[2] WSDL 2.0 defines additional MEPs, and lets services define their own.[3] Yet, the WSDL specification doesn't include a language to describe the conversation policy associated with each MEP; it uses plain English, which means that humans have to interpret and implement these policies.

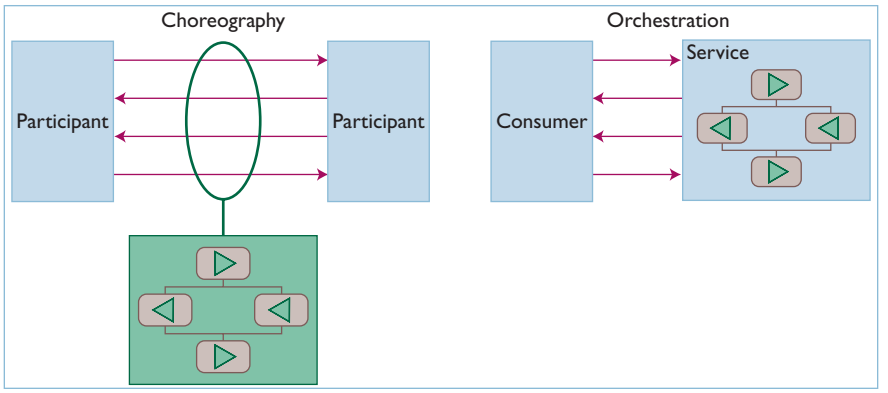

Figure 2. Choreography and orchestration. Choreography specifies the conversation from a neutral observer's viewpoint, whereas orchestration defines a process to be executed by the service provider.

Choreography

A conversation policy primarily concerns what happens between communicating parties — that is, who is allowed (or expected) to send messages to whom and in what order. At any point, we can consider the conversation to be in a specific state. For example, a simple request–response conversation has a single active state: "waiting for response." The conversation enters this state when the client sends a request message and leaves it when the service responds. If the client is allowed to retry, the conversation could also enter a state such as "two requests sent but still awaiting response."

As Figure 2 illustrates, choreography tracks the conversation state according to the messages being exchanged. The conversation state description takes a neutral observer's viewpoint. This "choreographer" is purely a logic construct — passing all messages through a central observer would clearly be undesirable in a highly distributed environment. Thus the conversation-state transition chart defined by the choreography is only a specification, rather than an executable language. But each individual's perspective can be derived from the neutral observer's view of the conversation, ensuring that all participants abide by the global rules. A client participating in a request-response conversation, for example, can easily derive that it must send a request message to start the conversation and that a response message ends it.

Various specifications have embraced the choreography model over the past few years to describe conversations, the strongest contender being the Web Services Conversation Description Language (WS-CDL).[4] Choreography's strength is that it extends well beyond simple two-party conversations. However, because the choreography isn't executable, it is conceptually rather abstract and can be more difficult to test and debug.

Orchestration

By pretending to observe a conversation between equal parties from "above," choreography assumes a certain symmetry between those communicating. However, most service contracts view the conversation from the provider's perspective, establishing rules that any potential consumer must observe. Concentrating on the service provider makes it easier to describe and enforce these rules. For example, a process containing send and receive primitives executing inside the service can orchestrate the interaction between the service, its consumer, and potential collaborators (see Figure 2). Consumers can also inspect this process definition to determine what messages they can send to the service, and which ones they can expect in response.

The Web Services Business Process Execution Language (WS-BPEL)[5] is currently the most popular language for defining processes in SOA environments. The BPEL specification includes abstract process templates, which describe the interaction between the process and conversation partners. The actual process executed by the service fills in this skeleton template with activities to execute incoming requests, compose response messages, and so on.

Rules

Both WS-CDL and WS-BPEL are based on a state-based conversation model, albeit from different viewpoints. In many cases, however, spelling out a complete state-transition chart for a conversation can be tedious. It might seem more natural to define a set of rules that the conversation must obey — for example, "after sending an invoice, a payment must follow" or "whenever the service receives a message, it replies immediately with an acknowledgment." Naturally, such rule sets' usefulness hinges on the vocabulary that the rules language supports and the ability to verify that a running system complies with the rules. For example, it would be difficult to verify a rule such as, "when the consumer receives a duplicate response, it ignores it."

The Soap Service Description Language (SSDL; www.ssdl.org/docs/v1.3/html/Rules SSDL Protocol Framework v1.3.html) supports a declarative rule-based approach. This rules language describes the relationship between messages — for example, "if an order was received, and no invoice has been sent in response, an invoice message is expected." These rules become part of the public service contract supported by SSDL.

Conversation Patterns

The standard specifications define languages to express conversation policies. These languages primarily target machines, and it's beyond the standards committees' scopes to provide

guidance on designing robust conversation policies that avoid deadlock situations and the like. Because designing conversation policies isn't something most developers are experienced in, design guidance is often as important as the conversation description language's syntax. A catalog of common conversations can fill this gap, as long as it goes beyond simply listing conversations (a la MEPs) to elaborate on the design trade-offs and assumptions incorporated in the specific conversation policy, when to use it, and when not to.

Design patterns have established themselves as an excellent tool for capturing knowledge and designing guidance that helps developers learn from others' experiences. Current work in conversation patterns[6,7] aims to create a human-readable language for conversation policies and to support service developers in making intelligent decisions when designing conversations.

Like most new tools, a conversation policy can turn into the proverbial hammer looking for nails. Service developers should keep in mind that conversations typically introduce an additional form of coupling — the more complex the conversation rules, the more tightly coupled service consumer and service provider tend to be. Furthermore, developers should avoid recreating lower layers of the communication stack inside the application layer. For example, protocols such as TCP and many middleware products already address duplicate packets, message resending, and other issues. Recreating this behavior at the service layer complicates the conversation without added benefit.

A precise definition of the conversations a service supports is a useful ingredient of an expressive service contract. However, designing conversation policies is neither a simple task, nor one that developers are usually familiar with. In addition to specifications and standards, we should start growing a body of knowledge on designing and implementing conversations. Real life is rich with examples of complex conversations and can serve as inspiration for cataloging useful conversation protocols.

References

1. G. Hohpe and B. Woolf, *Enterprise Integration Patterns*, Addison-Wesley, 2003; www.eaipatterns.com.
2. E. Christensen et al., "Web Services Description Language (WSDL) 1.1," W3C note, 15 Mar. 2001; www.w3.org/TR/2001/NOTE-wsdl-20010315#_porttypes.
3. "Web Services Description Language (WSDL), version 2.0 part 2: Adjuncts," W3C candidate recommendation, 27 Mar. 2006; www.w3.org/TR/2006/CR-wsdl20-adjuncts-20060327/#meps.
4. N. Kavantzas et al., eds., "Web Services Choreography Description Language, version 1.0," W3C candidate recommendation, 9 Nov. 2005; www.w3.org/TR/ws-cdl-10/.
5. P. Yendluri et al., eds., "Web Services Business Process Execution Language," Oasis committee draft, 17 May 2006; www.oasis-open.org/committees/document.php?document_id=18714.
6. A. Barros, M. Dumas, and A.H.M. ter Hofstede, *Service Interaction Patterns: Towards a Reference Framework for Service-Based Business Process Interconnection*, tech. report FIT-TR-2005-02, Queensland Univ. of Technology, Mar. 2005.
7. G. Hohpe, "Workshop Report: Conversation Patterns," *Dagstuhl Seminar Proc. 06291: The Role of Business Processes in Service Oriented Architectures*, F. Leyman et al., eds., Internationales Begegnungs und Forschungszentrum fuer Informatik (IBFI), 2006; http://drops.dagstuhl.de/opus/frontdoor.php?source_opus=828.

Gregor Hohpe is a software architect with Google. His interests focus on asynchronous messaging and service-oriented architectures. He is coauthor of *Enterprise Integration Patterns* (Addison-Wesley, 2003) and speaks regularly at technical conferences around the world. Find out more about his work at www.eaipatterns.com.