Conversation Patterns:

Interactions between Loosely Coupled Services

Gregor Hohpe
gregor@hohpe.com
www.eaipatterns.com

June 18, 2008

Abstract

As applications transform into collections of distributed services, the interaction between services steps into the foreground. Interactions between services range from simple request-response patterns to complex multi-party interactions that can span weeks or months. Service developers and designers define the conversation policies, the rules the interaction participants must abide by. Designing conversations is not simple task, however. Distributed services, unreliable networks, and failing or non-conforming participants require robust conversations that avoid deadlock and race conditions. Because this is not a task most developers are familiar with, design guidance is needed. Numerous standards and specifications define the syntactic elements needed to describe a conversation policy, but do not provide design guidance or evaluation criteria. This paper describes a pattern language for conversations between loosely coupled services, enabling developers to develop and describe robust interactions amongst services.

Contents

1 Introduction ... 3
1.1 Request-Response Revisited ... 3
1.2 Challenges of Distributed Communication ... 5
1.3 Conversations ... 5
1.4 Describing Conversations ... 7
1.5 Conversation Patterns ... 9
1.6 Related Work ... 10
1.7 Standards and Specifications ... 11

2 A Pattern Language for Conversations ... 14

3 Discovery ... 15
Dynamic Discovery ... 15
Consult Directory ... 19
Referral ... 20

4 Establishing a Conversation ... 22
Three-way Handshake ... 22
Acquire Token First ... 23
Rotate Tokens ... 25
Address Verification ... 25

5 Basic Conversations ... 28
Fire-and-Forget ... 28
Request-Response ... 29
Request-Response with Retry ... 32
Polling ... 34
Subscribe-Notify ... 35
Quick Acknowledgment ... 36

6 Multi-Party Conversations ... 38
Proxy ... 38
Relay ... 39
Leader Election ... 39

7 Ensuring Consistency ... 40
Ignore Errors ... 41
Retry ... 41
Contingent Requests ... 42
Tentative Operation ... 43
Compensating Action ... 43
Reaching Agreement ... 43
Perform Most Likely to Fail Action First ... 44
Perform Hardest to Revert Action Last ... 44

8 Reaching Agreement ... 46
Receiver Cancels ... 46
Sender Cancels ... 46
Binding Request ... 46
Binding Offer ... 47

9 Resource Management and Termination ... 48
Lease ... 48
Renewal Reminder ... 49
Heartbeat ... 50

10 References ... 51

1 Introduction

Service-oriented architectures (SOA) put interaction into the spotlight. A single service is about as useless as a single fax machine. Like the fax machine, the power of services stems from a network effect, best described by Metcalfe's Law, which states that the utility of a network increases with the square of the number of participants. Services represent functional assets that are widely available to applications, business partners, powerful composite services, or long-running business processes. Services-oriented architectures allow many heterogeneous systems to connect to each other, thus growing the size of the network and increasing the resulting utility.

One basic tenet of service-oriented architectures is a simplified, message-oriented interaction model. Such a model reduces coupling between services because it allows services to make fewer assumptions about each other. How loosely or tightly coupled services are depends very much on the way they interact. Ironically, the only truly loosely coupled services are the ones which are not connected at all [1]. The value of an SOA comes from interacting services, though, making some coupling inevitable. The much debated question is how much coupling is necessary or desirable. For example, one of the fiercest debates in the Web services world, the REST vs. WS-* debate [ref!], is primarily concerned with the service interaction model and the resulting amount of coupling. Generally speaking, the simpler the interaction between parties, the looser the coupling, the simplest interaction consisting of a single message being sent from one participant to another.

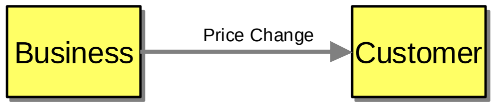


Figure 1: A very simple interaction

Not all system interactions can be this simple, though, especially when they span business boundaries or have to work over unreliable networks. For example, let's assume a customer requests a quote for an item. The seller checks the warehouse for availability, potentially contacts suppliers for estimated delivery times, before issuing a quote. The customer may subsequently place an order based on this quote, and receive an order confirmation, followed by an invoice and a shipping notice. The customer (hopefully) submits a payment and confirms receipt of the goods. Such a conversation can span weeks or months, and becomes even more complex once it accounts for transmission errors, mistakes, cancellations, retries, and compensating actions. To manage such a complex conversation, each business partner typically executes a process engine, which tracks the current state of the conversation.

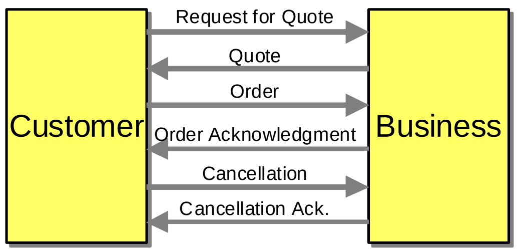


Figure 2: A business –to-business interaction

1.1 Request-Response Revisited

Even simple interactions often involve more than simply sending a message from A to B, once one takes into account the challenges of remote communication. Request-response is a very simple interaction between two systems, which mimics a procedure call from one system to the other: a service consumer sends a message to a service provider, expecting a response message in return (see Figure 3). The message-based interaction uses separate request and response messages. If the communication channels are implemented as queues, this approach enables temporal decoupling between service provider and consumer: the consumer does not have to sit around and wait for a response (or hold a connection open), but can go on to other things until the response message arrives.

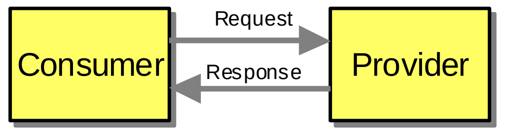


Figure 3: Request-Response Interaction

But, a nagging question surfaces: what should the caller do if it does not receive a response message? This failure scenario could not occur on the non-distributed world, because a method call always finishes with a return to a caller, either by passing a return value or by throwing an exception. But in the distributed world it is a quite possible occurrence, caused for example by a glitch in the communication. For example, the request message or the response message could be lost, or the server consumed the request message, but then crashed unexpectedly and never sent a response. The sender has essentially two choices to deal with this situation: give up or resend the original request message. If the caller decides to resend the request message, a number of new considerations come 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. In many cases, though, the provider should not perform the requested action a second time, i.e. it has to *idempotent*. An *Idempotent Receiver* [2] guarantees that repeated messages have no observable effect. Unless the operation itself is idempotent, such as a read operation, the receiver must be able to identify resent messages and ignore them. Allowing consumers to resend messages requires both service consumer and provider to maintain conversation state: the consumer needs to keep the request message to resend it, while the idempotent provider needs to keep track of already received message IDs, along with the responses, so it can resend them without performing the operation again.

The service consumer also has to deal with duplicate messages. For example, the provider might have sent a response through a message queue just as the consumer decided to give up waiting. In this case the consumer resends the request message, just to receive the original response message a fraction of a second later. This makes the consumer happy, but a little while later it will receive another response message, based on the resent request. In most cases the consumer should ignore this message because the response has already been processed. This means the consumer has to be idempotent as well.

What should the consumer do if it still receives no response even after resending the request? The consumer can retry the request again, but it should avoid resending the message ad infinitum. For example, the response may not come because the particular message is a "poison message", causing the service provider to crash every time. The consumer should limit the number of retries and give up eventually. This "agreement" requires the consumer to track an additional piece of interaction state, the number of retries so far.

1.2 Challenges of Distributed Communication

The request-response example highlights some of the inherent challenges of communication in distributed systems:

- *Lack of Atomic Transactions*. Communication between distributed systems does not have the benefit of atomic transactions, which guarantee an all-or-nothing outcome. While implementing two-phase commit semantics across distributed systems is possible, in reality it is rarely implemented, for reasons both of complexity and throughput [23]. This means communicating systems have to deal with failure scenarios that are otherwise handled "under the covers" by the transaction monitor.
- *Tentative Operations*. In a world without atomic transactions, communicating entities use tentative operations to reach a shared outcome. For example, a customer may request an airline reservation to be put on "hold", so he can check the availability of hotels before making the final booking. The reservation is booked in a separate step.
- *Timing Uncertainty*. Message-based communication achieves temporal decoupling between communicating parties. This approach brings distinct advantages, for example, the communication is not impacted even if one communication partner is temporarily unavailable. It also means systems can't have specific expectations how long certain operations will take.
- *State Uncertainty*. When multiple systems communicate, one system is inherently uncertain about the state of the other system. Did the other system receive the message? Was it processed successfully? Did the other system crash or is it simply slow? As a result, reaching consensus over a potentially unreliable communications channel can be difficult at best, or plain impossible. The Two Generals Paradox [5] and the more general Byzantine Generals Problem [4] describe this dilemma in much detail.
- *Numerous Failure Scenarios.* The number of things that can go wrong in a distributed system vastly surpass those of a monolithic system. Lost or delayed messages, unexpected or misrouted messages, data corruption, out-of-order messages describe just some of the possible scenarios. Network protocols can shield the application layer against some, but usually not all of these evils.

To overcome these challenges, the interaction between systems tends to be more complex than simply sending a message from A to B. In the request-response example, both consumer and provider had to eliminate duplicate messages, monitor time-out conditions, and count the number of retries. The interaction consists of a series of related messages, which are all part of a single *conversation*.

1.3 Conversations

A *conversation* is an exchange of individual, but related messages over time. A conversation consists of the following elements:

- The conversation occurs between two or more *participants*. The number of participants may be constant or can vary as the conversation progresses. For example, a business may solicit bids through an open bidding process and may not know in advance how many responses it'll receive. One participant typically acts as *initiator*, i.e. it starts the conversation by sending the first message.

- Participants communicate by exchanging *messages*. Each message has one *sender* and one or more *receivers*.
- Messages can be associated or *correlated* to the conversation. This is usually accomplished by embedding a conversation identifier of *correlation identifier* in each message.

The request-response conversation described above defines two participant roles: a *service provider* and a *service consumer*. Both the provider and the consumer send as well as receive messages, giving the conversation a certain symmetry. This symmetry underlines the differences between a remote procedure call, which is inherently asymmetric, and message-based communication as part of a symmetric conversation.

Participants can be involved in multiple conversations at the same time. For example, it's quite common for a service consumer to act as service provider to other consumers. Because communication is asynchronous, one participant may also take part in multiple instances of the same type of conversation, playing the same role. For example, one consumer may make engage in multiple request-response conversations at the same time, making requests to multiple service providers. When the consumer receives a response message, it uses the correlation identifier embedded in the message to associate the message with the right conversation instance. Participants typically track the state of each conversation they are involved in, so that they know what to do next when a message arrives.

Conversations between services share many similarities with conversations in real life. Humans frequently interact through asynchronous messages, e.g. leaving a voice mail, mailing a letter, or sending e-mail. In doing so, humans have to deal with many of the same challenges as loosely coupled computer systems, such as duplicate messages (“I really need that TPS report”), messages crossing in transit (“ignore this notice if you already sent your payment”), or coordination of multiple independent resources ("You got the money? You got the goods?"). Humans also use correlation identifiers (e.g., an order number) or context (e.g., the subject line of an e-mail message) to relate individual messages to the conversation.

A real-life conversation may start when a manager asks an assistant to help arrange a suitable meeting time (see Figure 4). The assistant contacts each meeting participant to check for available times. Based on this information, the assistant determines a time that suits everybody and announces the chosen time to all participants. Shortly before the meeting the coordinator might send a reminder notice to all participants.

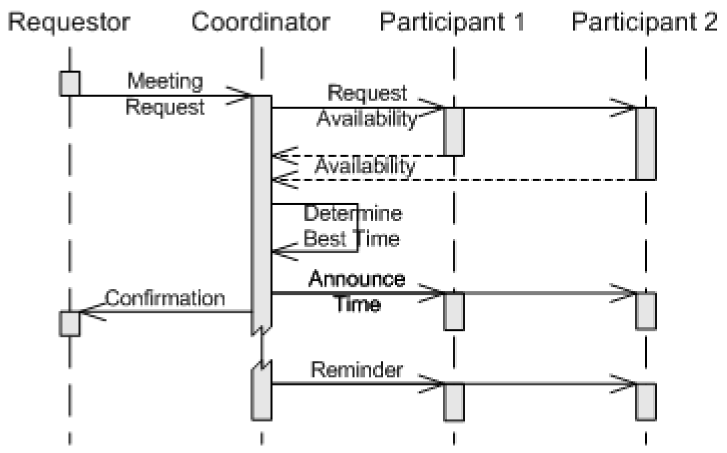


Figure 4: A conversation between four parties

This conversation occurs amongst four parties, two of which play the same role of Meeting Participant. To make the conversation successful, each party has to abide by the rules of the conversation, for example meeting participants need to provide available times when asked.

1.4 Describing Conversations

When a party participates in a conversation it needs to understand what is required of it, as well as what it can expect from the other conversation partners. For example, a service consumer needs to know whether it can safely resend a request, or whether it has to be prepared to deal with duplicate response message. In the above examples, we described conversations with prose or by looking at a specific example. Conversation participants need a precise description of *all* legal conversations, though. This is the purpose of a *conversation policy*, a definition of the conversation participant roles and the legal conversations between them. Such a policy consists of the following elements:

- **Participant Roles** - Each conversation assigns specific roles to the participants, for example, a buyer and a supplier. Or a meeting requestor, a meeting coordinator, and a meeting participant. One physical entity (a business, a human being, or a computer system) might play more than one role, for example the meeting requestor might also be a participant. Likewise, multiple people can play the same role, e.g. multiple people are likely to play the role of meeting participant.
- **Message Types** - Participants communicate through exchange of messages. Participants have to understand these messages and take appropriate action. For example, when a meeting participant receives a message requesting available times, he is expected to respond with a list of times. On the other hand, participants are not expected to respond to messages of type *Meeting Reminder*.
- **Protocol** - The protocol defines legal sequences of messages between the participants, i.e., which messages can be sent by whom, and in which order. It is a declarative specification, with which actual conversations must comply. For example, the meeting coordinator cannot send out a confirmation of the meeting time until

everybody has provided their availability, whereas the meeting participants should expect to be pinged about their availability at any time.

If a conversation fulfills the rules established by a conversation policy, we call that conversation *legal* with regards to that policy. Depending on the policy, many actual conversation instances can be legal. For example, a policy may not prescribe an order between the shipment of goods and the sending of an invoice. Accordingly, in some instances the supplier may send the *Invoice* message first, while in other instances it may send the *Shipment Notice* message first. Both conversations are legal (see Figure 5). Even if the supplier's internal implementation is hard-wired to always send the invoice first, the buyer still has to be prepared to receive them in either order, because the stated conversation policy allows messages to be sent in either order. This could be the case because the supplier reserves the right to change the implementation in the future to send the Shipment Notice first, without having to revise the conversation policy.

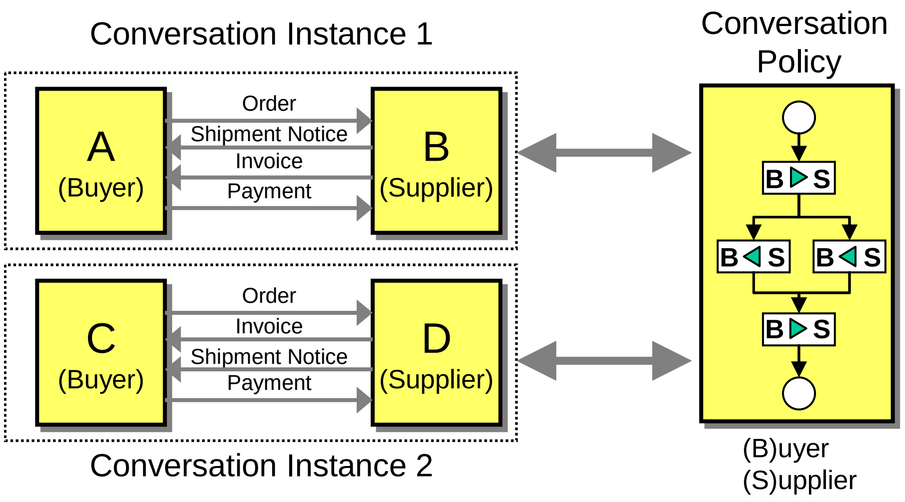


Figure 5: Conversation Policy and Legal Instances

Describing the protocol aspect of the conversation policy can be challenging. The following approaches are used in practice:

- **Enumeration**. If the conversation policy is fairly strict, the number of legal conversations may be low. In these cases, the policy can simply enumerate all legal conversations. For example, a basic request-response conversation could be described this way. While conceptually simple, this approach quickly reaches its limitations. For example, if the consumer in the request-response conversation is allowed to resend messages, enumerating all possibilities becomes tedious at best, and impossible for the general case.
- **Choreography.** The protocol describes which messages can be sent by who in which order. While some messages have to be sent one after another in a *sequence*, others can be sent in any order, i.e. in *parallel*. Thus, one can encode the rules of the conversation in a process diagram, which defines which messages can be sent by which participant. This process diagram view the conversation from the perspective of a neutral observer, who considers all conversation participants as equal parties. For

example, the choreography in Figure 6 defines that participant A can send messages to B and C in an order. B and C respond to these messages with a message each.

- **Temporal Logic.** Rather than using a process-like model to describe legal conversations, one can describe the relationships between the messages through temporal logic. For example, an *Order* message has to precede any *Invoice* message, while any conversation has to conclude with a *Payment* message.
- **Orchestration.** In many conversations one participant plays the role of a central coordinator. For example, in a request-response conversation the service provider typically determines the rules of the conversation. In the meeting coordination conversation depicted in Figure 4, the coordinator also plays a central role. If this *orchestrator* executes a process, which includes sending and receiving messages to other participants, the conversation policy for all participants is implicitly defined. Naturally, this approach only works for those conversations that have a central coordinator, but not for pure peer-to-peer conversations.

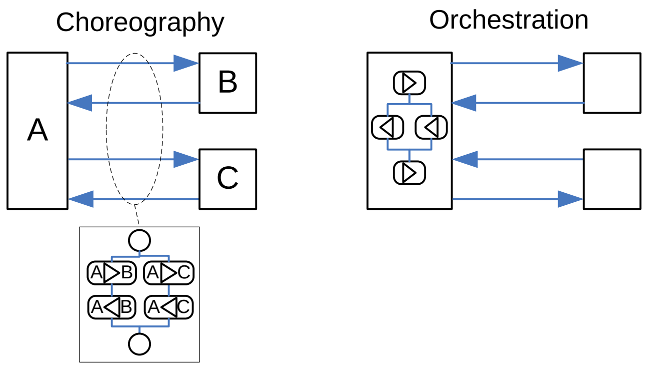


Figure 6: Comparing Choreography and Orchestration

1.5 Conversation Patterns

Even with the right language to *describe* a conversation policy, *designing* a conversation policy is not trivial. A good conversation should be efficient, i.e. it should not be overly "chatty". It should also be robust against delayed or lost messages and avoid deadlock situations.

Design patterns have proven themselves as a very effective learning vehicle for good software design. For example, in the object-oriented development community patterns have become the de facto way of documenting design insight and guidance. Design patterns capture "mind-sized" chunks of knowledge, which are easy to grasp, yet useful. They elaborate the "why" as much as the "how" of a proposed solution. Such patterns go far beyond simple cook book-style recipes, but instead help developers build solutions that balance often opposing forces and constraints. Patterns explain when a solution is suitable, and what its limitations are. Patterns are not universal copy-and-paste solutions, though: they all have strengths and weaknesses. Using the right pattern for the right purpose is key.

Because conversation protocol design is complex and subject to many constraints, design pattern promise to be an effective vehicle to provide the much needed guidance. That's why this paper defines a series of *conversation (design) patterns*, which can assist developers in designing correct and efficient conversations between loosely coupled systems. Many of the patterns will seem familiar from well-known protocols, or even daily life. This is one of the strengths of design patterns: similar solutions can often exist at different levels of abstraction. For example, a low-level network protocol might use the same *Polling* strategy as a high level business protocol or a customer who keeps calling a store to see whether an order has been processed.

The patterns are organized into a language, which is structured into the following focus areas, which are ordered roughly chronologically. For example, services have to discover their conversation partner before they can hold a conversation or reach agreement.

1. *Discovery*. Before services can interact, they have to identify their conversation partner(s). In a loosely coupled system, the participant roles of the conversation are dynamically filled in with physical service instances. This mapping requires the appropriate services to be "discovered".
2. *Establishing a Conversation*. Once the conversation partner is known, a conversation has to be established. These interactions include authentication, handshaking, and establishing the parameters of the conversation
3. *Basic Conversations*. Some conversations involve only two participants and a small number of message types. The basic patterns establish a core vocabulary for the description of more complex conversations, which can be composed from basic conversations.
4. *Multi-Party Conversations*. Complex conversations can involve three or more participants. In some cases, the number of participants may not be know at the beginning of the conversation.
5. *Reaching Agreement*. Long running interactions, such as business-to-business conversations lack atomic transaction semantics. However, the ability for multiple independent parties to reach an agreement is a critical part of many business interactions.
6. *Terminating Conversations*. Acquiring resources from a service results in a form of coupling. If the requestor disappears unexpectedly, the resource provider might continue to hold unneeded resources. Resource management patterns deal with resource allocation and deallocation.
7. *Error Handling*. Things don't always go according to plan, especially in a highly distributed system where communicating services may reside in different organizations or different continents.

Chapter 2 gives a detailed overview of the pattern language.

1.6 Related Work

Business-to-business integration and service-oriented architectures rely on distributed system communication and conversations between loosely coupled systems. The importance of conversations, coupled with the challenges, has motivated a variety of pattern collections. Most of these pattern languages focus on a specific aspect of a conversation.

This section lists pattern languages related to service conversations, without trying to be comprehensive.

- *Service Interaction Patterns* [3] describes 13 patterns for the interaction between services. The patterns range from basic communication constructs, such as *Send*, and *Receive*, to more complex conversations such as *Contingent Requests* or *Request with Referral*.
- *Remoting Patterns* [10] describes patterns for the implementation of distributed systems, particularly object-oriented systems.
- *Pattern Language for Service Discovery* [11] describes a pattern language composed of 12 patterns detailing the discovery of services. The patterns focus on the mechanisms of service discovery as opposed to the conversations involved.
- *Contracting Workflows and Protocol Patterns* [8] examines the interactions between a service provider and a customer. In this context, "services" does not refer to Web services but to any service provider in a business context, such as a subcontractor. The patterns cover business concepts from specification to negotiation, execution, and acceptance.
- *Workshop Report: Conversation Patterns* [9] documents the findings of a workshop on conversation patterns at the Dagstuhl Seminar on The Role of Business Processes in Service Oriented Architectures. The discussion covered the potential scope of the pattern language, notations to be used, as well as concrete pattern candidates.

1.7 Standards and Specifications

The wide adoption of service-oriented architectures can partly be attributed to the definition of vendor-neutral standards and specifications. Because conversation policies are a valuable component of a public service contract and ensure smooth and robust interaction between services, Web service specifications also address conversation policies. Unlike the conversation patterns, the specifications define language syntax and semantics, which can be used to describe conversation policies, but do not give guidance on the design of robust conversations. Also, because these languages are designed to be executed by machines, they often operate at a very low level, and may make limited sense to the human reader.

This section provides a brief reference of related standards and specifications:

Message-Exchange Patterns

Given the challenges in creating a correct conversation policy, it is sensible to simply enumerate a few common conversations, which services choose from. 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. WSDL 2.0 defines additional MEPs, and lets services define their own. However, the WSDL specification doesn’t include a language to describe the conversation policy associated with a MEP; it uses plain English, which means that humans have to interpret and implement these policies.

WS-CDL

The Choreography Description Language (WS-CDL) specification describes the language as "aimed at being able to precisely describe collaborations between any type of participant regardless of the supporting platform or programming model" [7] WS-CDL takes the choreography approach described in Section 1.4, defining which participant can send what type of message to which other participant in what order. The message ordering constraints are expressed through the elements *sequence*, *parallel*, and *choice*, similar to a process models. For example, a sequence element can prescribe that a request message has to be sent by the consumer before a response message can be sent by the service provider. Likewise, the parallel element expresses that two or more messages can be sent in any order.

WS-CDL describes a conversation from a neutral observer's viewpoint, "watching" the messages being exchanged between the participants. This "choreographer" is purely a logic construct — passing all messages through a central observer would clearly be undesirable in a highly distributed environment. Accordingly, the choreography is only a specification, not an executable language. Tools can, however, project use this specification to generate endpoint processes (e.g. Java code or a BPEL process template) that fulfill the constraints set forth by the choreography.

WS-BPEL

WS-BPEL [12] has established itself as the de facto standard for the description of process templates, which can be executed inside a Web service. The process definition includes activities for sending and receiving messages, which describe the interaction of the service with other services. Effectively, the service orchestrates he conversation with other participants.

Unlike WS-CDL, WS-BPEL is an executable specification, which is parsed and executed by a process engine, thus requiring all messages in the conversation to pass through the central "orchestrator". The specification can support conversations between more than two parties as long as one of them assumes the role of central orchestrator.

RosettaNet

RosettaNet is a consortium that defines processes and protocols for business to business (B2B) data exchange, mostly for supply chain optimization in the semiconductor and electronics business. B2B conversations can span over extended periods of time and may require many messages to be exchanged between business partners, especially when taking into consideration alternative scenarios such as order cancellations or updates. The precise definition of these conversation policies are the key to successful integration between business partners.

RosettaNet defines its conversations on two levels. The RosettaNet Implementation Framework (RNIF) [18] defines the basic interactions, such as *Asynchronous Single-Action Activity*, *Asynchronous Two-Action Activity*, or *Synchronous One-Action/Two-Action Activity* (section 2.6 of RNIF 2.0). RNIF describes the conversations through a combination of flow diagrams and plain text. Partner Interface Processes (PIPs) are layered on top of the RNIF and define message schemas as well as the valid sequences of messages. The scope of each PIP is relatively small, for example PIP3A1 defines *Request Quote* whereas PIP3A4 defines *Request Purchase Order*, which follows the request for a quote. However, when taking into account acknowledgement messages as well as exception scenarios, even the simple interchanges follow relatively complex conversation policies.

BPMN

The Business Process Modeling Notation (BPMN) is graphical notation for drawing business processes. These processes can include message exchanges between participants, each of which executes a portion of the process. The placement of message send and receive activities within the process branches defines the conversation policy, i.e. which messages can be exchanged in which order. This approach resembles the way BPEL defines conversations. The conceptual similarity allows processes designed in BPMN to be translated into BPEL process templates for execution in an orchestration engine.

UML 2.0 Sequence Diagrams

The Unified Modeling Language (UML) is a notation to describe various aspects of software systems, with a particular focus on object-oriented systems. It includes the notion of *sequence diagrams*, which describe messages exchanged between objects (Figure 4 uses the sequence diagram notation). Earlier versions of the UML could only describe one specific sequence of messages, resembling the Enumeration approach described in Section 1.4. Naturally, it suffered from the same limitations, most importantly the inability to describe conversation policies that allow a wide variety of message sequences. This limitation was rectified with version 2.0 of the UML specification. Enhanced sequence diagrams can now describe messages that can be sent in either order as well as repeating messages.

Agent Communication Languages

Agent programs are designed to autonomously collaborate with each other in order to satisfy both their internal goals and the shared external demands generated by virtue of their participation in agent societies [15]. Conversations and conversation policies are essential to the successful coordination between multiple agents. Approaches such as KAoS [17] define conversation policies based on finite state machines. KAoS also includes a set of frequently occurring conversations between agents, such as *inform*, *offer*, *request*. [16] gives a good overview over work related to conversation policies for agent-based systems.

2 A Pattern Language for Conversations

This paper presents a pattern language that assists developers in designing effective and robust conversations between services. The patterns range from low-level communication primitives to business-level negotiation.

The conversation patterns in this paper are organized into a language that guides developers in the design of robust conversations between services. The patterns form a language that enables service developers to design solutions at a higher level of abstraction and to communicate their design decisions and intent more effectively.

The patterns are organized into a language, which is structured into the following focus areas, which are ordered roughly chronologically. For example, services have to discover their conversation partner before they can hold a conversation or reach agreement. Each focus are tries to find a compromise between often conflicting trade-offs or *forces* and suggests patterns that balance these requirements.

Expand 7 include pattern names!

1. *Discovery*. Before services can interact with one another they have to identify their conversation partner(s). In a loosely coupled system, the participant roles of the conversation are dynamically filled in with physical service instances. This mapping requires the appropriate services to be "discovered".
2. *Establishing a Conversation*. Once the conversation partner is known, a conversation has to be established. These interactions include handshakes and reaching agreement on parameters of the conversation
3. *Basic Conversations*. Conversation patterns are meant to be composable, such that more complex conversations can be described as a combination of smaller sub-conversations. The basic patterns establish a core vocabulary for the description of more complex conversations.
4. *Multi-Party Conversations*. While basic conversations involve only two parties, more complex conversations can involve three or more parties.
5. *Ensuring Consistency*. In service-oriented environments, operations commonly span across multiple, independent services. In most situations, consistent execution of the operation is required, i.e. all individual operations should succeed, or, if something goes wrong, no operation should be completed. In a tightly-coupled, and tightly controlled environment, consistency is usually provided through a transaction mechanism. Generally, such transactions are not available or practical, in highly distributed, loosely coupled systems.
6. *Reaching Agreement*. Reaching agreement in a business-to-business setting can be challenging. Representing the negotiations, cancellations, amendments, in a system can be equally challenging.
7. *Resource Management and Termination*. Acquiring resources from a service results in a form of coupling. If the requestor disappears unexpectedly, the resource provider might continue to reserve unneeded resources. Resource management patterns address such issues through resource allocation and leases.

3 Discovery

Conversations involve interaction between multiple conversation partners. To keep the interaction as loosely coupled as possible, the partners should not be fixed, but should instead "discover" each other. This is a two part process. First, the participants in the conversation have to be identified. Subsequently, the participants have to agree as to which participant is playing which role. For some conversations, the discovery inherently defines the roles, such as in a simple *Request Response* conversation: the consumer (or requestor) always discovers the service provider, alleviating the need to agree on roles. Other conversations, however, may require some negotiation. For example, multiple participants may elect a leader out of a pool of candidates using *Leader Election*.

Many of the tradeoffs in service discovery revolve around the following design decisions:

- *Centralized or distributed.* Some discovery processes rely on a central registry while others are fully decentralized. Each approach has advantages and disadvantages. A central registry has knowledge about all services and can make smart decisions about how to fulfill a discovery request, for example based on load, proximity, or similar policies. However, a central registry can become a bottleneck or point of failure. Also, the registry can get out of sync with the reality when a registered service is no longer available. Lastly, a service has to discover the registry in the first place.
- *Precision*. Discovery requires the initiator to express what it is looking for. The initiator may specify its need using keywords, a magic identifier, a set of criteria, or based on a refined ontology.
- *Division of responsibilities.* Depending on the precision of the discovery, different amounts of participation may be required on part of service providers or the initiator. For example, the initiator might have to solicit responses from all services and then has to figure out which one it wants. Other approaches may put more burden on the to-be-discovered services, requiring them to announce their presence, their functional and non-functional attributes, etc.
- *Dynamicity*. Some service environments are more static than others. For example, enterprise services may be reliable and do not come and go at will. However, on a short-range wireless network services may appear and disappear all the time.
- *Security and Control*. Allowing conversation partners to discover each other can require some amount of control as well security. Can you trust your conversation partner? Or could it be an imposter? Who should be allowed to "see" a service and connect to it?
- *Addressability*. Most discovery patterns assume addressability of services, i.e. the ability to contact a service endpoint based on a resource locator, a channel name, or a similar unique identifier. These identifiers have to be embeddable in messages so that one service can pass a service's address to another service.

Dynamic Discovery

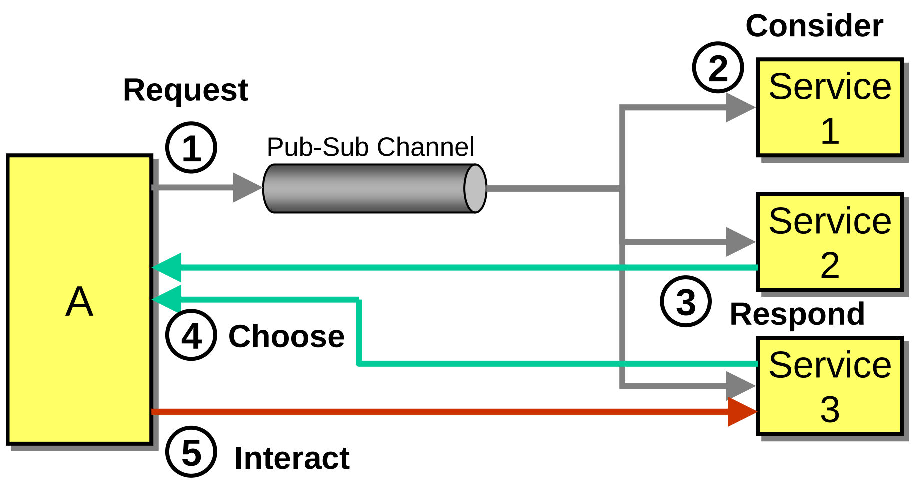
Ask Local Network [11]

To initiate a conversation, a service needs to find a conversation partner. When a service joins a network it might not have any knowledge about which other services are available on the network.

How can a service find a conversation partner in a network when the service has no knowledge whatsoever about the network and its services?

- Point-to-point communication requires the initiator to know the conversation partner it wants to connect to. Publish-subscribe communications allow a sender to broadcast messages to multiple subscribers without the sender having to know the subscribers identity.
- The late binding between a service consumer and the service endpoint lowers the location coupling between them. However, it also puts the discovery on the critical path to establishing a conversation.
- Even in the presence of a central lookup service, a new participant has to first establish a connection to the lookup service. This can provide a bootstrap problem.
- Deploying new services should be easy and involve as little manual administration as possible.

The client broadcasts a query to the network, indicating what kind of services it is looking for. All potential service providers listen to the query, with the ones who are able and willing to fulfill the request responding. If the initiator receives more than one response it decides which service to interact with.



This discovery process consists of the following five steps:

1. **Request**. The initiator sends a service request via a publish-subscribe channel. The requestor has no knowledge of any specific service recipient.
2. **Consider**. Potential service providers receive the request and can decide whether to respond or not.
3. **Respond**. If the service decides to offer a service to the initiator, it responds to the supplied Return Address.

4. **Choose**. The requestor receives responses and decides which service to interact with.
5. **Interact**. The requestor begins a conversation with the chosen service. The initiator may also announce its decision to the other service candidates to allow them to relinquish potentially allocated resources.

The publish-subscribe "channel" often takes the form of an IP multicast on a local network segment. This approach has the advantage that only nearby services see the request, increasing the chances that an initiator finds a nearby service that can fulfill the request.

Both the initiator and the service providers can apply rules as to whether they want to become conversation partners. These rules can be based on criteria specified by the initiator (e.g. the type of service that is needed) or the state of the service provider (e.g., number of clients that are currently being served).

The initiator acts as an *Aggregator* [2] when it receives responses from candidate services. This means the initiator has to decide how long to wait for incoming responses and how pick a service provider from the responses. Because the initiator does not know how many services may respond, the completeness condition typically employs a "first best" or "timeout" strategy.

If the initiator receives more than one response from a potential service provider, it might keep the whole list and use it to engage in a *Contingent Requests* conversation.

This dynamic discovery approach works particularly well in environments where services appear and disappear frequently. Such a dynamic environment can make it difficult to keep a registry in sync with the reality. *Dynamic Discovery* can also be used to discover a registry, in which case *Consult Directory* follows the *Dynamic Discovery*.

Once the initiator has identified a conversation partner, it may use other patterns to establish a conversation. For example, it might need to establish trust with the service provider before it can begin to exchange information. Because *Dynamic Discovery* establishes a direct connection between initiator without any intermediary, the communicating parties may have to spend more effort establishing a conversation.

Dynamic Discovery works best on a controlled network where service providers can be trusted. Because any provider can receive the broadcast request an untrusted provider could easily join the network and pretend to offer services.

Broadcasting across the network can be inefficient if initiators frequently look for conversation partners. Therefore, *Dynamic Discovery* works best with long running conversations or stateful conversations that do not require an initiator to rediscover a provider every time.

The initial interaction of *Dynamic Discovery* is similar to *Scatter-Gather* [2]. The main difference lies in the fact that *Scatter-Gather* is based on a stateless *Pipes-and-Filters* [2] model, whereas *Dynamic Discovery* is a stateful conversation. For example, in *Scatter-Gather* the initiator and the *Aggregator* [2] may be completely disconnected entities, whereas in *Dynamic Discovery* responses have to come back to the initiator to be meaningful.

Example: DHCP

Most IP-based networks assign IP addresses dynamically. When a machine joins a network, it is initially without an IP address and needs to acquire one. To avoid address collisions, a Dynamic Host Configuration Protocol (DHCP) server issues and tracks IP addresses "leased" to machines on the network. In order to obtain an IP address, a new machine has to connect to the DHCP server first.

DHCP uses UDP broadcast packets to locate a DHCP server. One or more servers respond with an offer to an IP address lease. The client accepts one offer and broadcasts its decision so that other DHCP servers can take back their offers (see Figure 7)

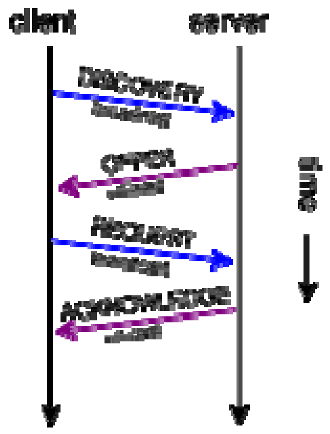


Figure 7: DHCP conversation

This interaction happens relatively infrequently as machines are connected to the network. TO manage IP addresses efficiently, DHCP servers use a *Lease* conversation to renew or free assigned IP addresses.

Example: TIBCO Repository

TIBCO middleware products such as BusinessWorks or Integration Manager store their design artifacts in a repository on the network. When a developer starts up the development environment, it uses *Dynamic Discovery* to discover reachable repositories. The *Choose* step of the conversation is performed by the human user by selecting a repository from the list.

Example: Finding a Car Buyer

In real life, people looking to sell their cars often "broadcast" the availability of the car by putting a sticker in the window as they drive around or park the car on a busy street. Interested parties contact the seller. The seller will choose one buyer and conduct the transaction. In some cases, the buyer may inform other interested parties that the deal was closed. This stays in contrast to finding a contractor by consulting a directory, such as the yellow pages.

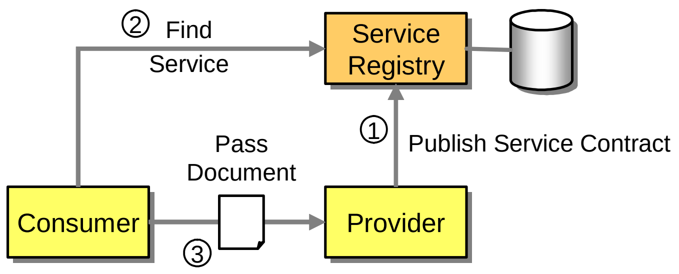
Consult Directory

In order to begin a conversation, a service needs to find a conversation partner. When a service joins a network it might have limited knowledge about which other service are available on the network. Also, multiple services may be able to fill the initiator's needs.

How can a service find a conversation partner across a large network without flooding the network with requests?

- The late binding between a service consumer and the service endpoint lowers the location coupling between them. However, it also puts the discovery on the critical path to establishing a conversation.
- Many networks do not route broadcast packets beyond the local network. This makes it difficult or impossible for an initiator using *Dynamic Discovery* to discover services that are not nearby.
- Broadcast or multicast queries do not scale to large networks of services as the network may become flooded with discovery requests.
- Often centralized administration is involved in setting up a new service.

The initiator consults a directory that manages a database containing information about available services. The directory refers the initiator to one or more appropriate services.



The directory has to provide a language for the initiator to express what type of service it desires. The following are common strategies:

- *Unique Identifiers*. Each type of service may be manually assigned a unique identifier. When a client discovers a service instance, it provides this identifier. This approach requires manual registration of service types.
- *Interface Definition*. An initiator may look for a service that conforms to a specific type interface.
- *Attributes*. Service may represent their capabilities through sets of attributes. The universe of all attributes may be fixed or open.
- *Keyword match*. Directories may not impose a structure at all, but allow the initiator to express its needs in plain text. The directory uses search heuristics to supply the initiator with services candidates.

A service directory is only useful if it keeps up-to-date with which services are available. Most directories use *Services Register in Directory* or *Directory Finds Services* [11] to maintain an accurate list of services. Alternatively, services can be entered by hand. The more frequently services appear or disappear, the more difficult it is for the directory to stay in sync with the real state of the network.

In many cases, the directory is represented by a single, specialized directory service. Initiators may locate the directory through *Dynamic Discovery* or global configuration. In some cases, multiple services may elect a directory from amongst them. This approach avoids a single point of failure in the network.

Because queries to the directory may be expensive, conversation initiators frequently cache results of previous inquiries. This approach reduces traffic on the directory but increases the likelihood that the initiator operates with out-of-date information. This approach is most viable in networks where service discovery is frequent, but services join or depart infrequently.

Example: UDDI Directory

In the Web services world, UDDI is the prototypical implementation of *Consult Directory*. tModels. <<more>>

Example: ProgrammableWeb

Developers looking for a Web service to perform a special function may consult programmableweb.com to find potential candidates.

Example: Yellow Pages

In real life, *Consult Directory* is very much like looking up a provider in the yellow pages. This directory organizes providers by type of service. However, due to the physical format the ontology used is only two-dimensional. First, by location (represented by the edition of the yellow pages book), then by type of profession or service offered.

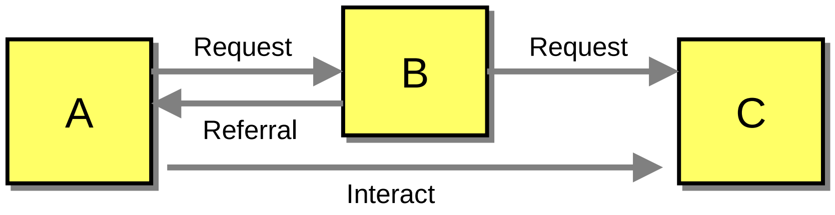
Referral

A service wants to initiate a conversation. The choice of conversation partner depends on an existing conversation or relationship between services.

Sometimes the choice of conversation partner depends on the context of a conversation, e.g. the fact that another service is already interacting with the same partner. How can an initiator discover the right service?

- In order to perform well, directories are generally context free, i.e. they do not keep track of existing conversations. They may take load or proximity information into account, but this information tends to be statistical and not based on a specific conversation that is already taking place.
- Some services may not be ready to be "discovered" by any other service. However, "friends of friends" are allowed to interact with them.

One service can refer a service provider to another service, such that the two services can start to interact.



The initiator interacts with the referrer, who knows about the referee. The referrer passes the referee's address to the initiator, who subsequently initiates a conversation directly with the referee. This conversation can be initiated either by A (looking for a referral) or by B (B wants A and C to have a direct conversation without having to relay messages).

This approach can work across trust boundaries. If C trusts B and B trusts A then B is allowed to pass a reference for C to A. This resembles the "Mafia approach".

A directory is a specialized case of a *Referral*, where the sole purpose of the referrer is to provide references.

This approach assumes addressability, such as URI's. These URI's can be embedded in a message from the referrer to the initiator. If trust and security are an issue, the referrer may also pass additional information, such as a token along.

Example: Supplier Refers Shipper to Customer

When a customer orders from a supplier, such as Amazon, the supplier often refers the customer to the shipper's service, for example FedEx's or UPS's tracking service. This referral usually includes a *Conversation Identifier* or handle, so that the customer can retrieve the associated information from the shipper.

Example: Social Networks

The power of social like Facebook or LinkedIn is based on referrals. People start to interact through a reference from their friends.

4 Establishing a Conversation

Once a service has identified a conversation partner, it can begin the actual conversation. But before the parties can start exchanging data, they might have some housekeeping to do, for example:

- *Confirming Roles*. In some peer-based conversations, participants need to agree on the role they are going to play. For example, participants may elect a leader from amongst themselves.
- *Authentication*. A service provider may require that potential clients authenticate themselves. Authentication can range from the simple inclusion of a user ID in the initial message to more complex conversations including a certificate authority. (Note: This is not a paper on security patterns. For a more detailed treatment see for example [20])
- *Sequencing*. Some protocols that need to transmit reliably over unreliable channels need to synchronize first to make sure no messages are lost later.

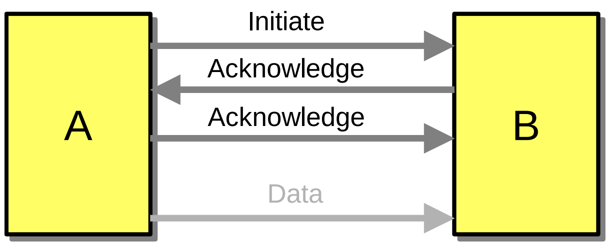
Three-way Handshake

A service consumer has identified a service provider and is about to initiate a conversation.

How can two services sync up so they can start a conversation from the same point?

- Reliable protocols often require sequence numbers to detect lost messages. Both parties need to start with the same sequence number so they can detect if the first message is already lost.
- Conversation partners may want to agree on parameters of the conversation, based on both parties' capabilities. For example, if both parties support compression or binary encoding, they can agree to switch to this more efficient message format.

Have the services perform a Three-way Handshake, where the requestor initiates a conversation, followed by an acknowledgment by the service provider, which is turn is followed by an acknowledgment (Ack-Ack) from the initiator.



The conversation starts with a request from the initiator, which often contains the desired parameters of the conversation, such as whether to use encryption, compression, a sequence number and the like. The service provider responds to the message with an acknowledgment. Lastly, the initiator begins the conversation with a final acknowledgment.

Establishing the basic parameters of a conversation, such as compression, data formats, etc, is a special case of a *Reaching Agreement*. One party makes a suggestion, which can then be acknowledged by the conversation partner. Alternatively, the partner can make a counter-offer, which is then acknowledged by the initiator.

Example: TCP Connection

The TCP protocol allows reliable, in-order delivery of network packets. It uses sequence numbers to detect out-of-order or lost packets. At the beginning of a TCP connection, the client and the server have to agree on initial sequence numbers. The client first sends a SYN (Synchronize Sequence Numbers) request to the server, providing its sequence number. The server acknowledges the client's sequence number by returning the number plus one, and also sends a SYN to the client, specifying its sequence number. Finally, the client acknowledges the server's sequence number by sending an ACK message with the number plus one.

Example: Fast Infoset

Fast InfoSet is a proposed standard for XML data compression and faster parsing. When a Web service client communicates with a service, it needs to make sure that both parties support the Fast Infoset spec. This is called the "negotiation" phase. The initial request made by a client is always encoded in XML, but the client can indicate that it prefers Fast Infoset encoded data by setting a specific HTTP header field. If the service supports Fast Infoset, it replies with a message that is Fast Infoset encoded. The remainder of the conversation between the client and the service will then be encoded in Fast Infoset. Sometimes this type of negotiation is referred to as *pessimistic*, in contrast to the *optimistic* case in which a client directly initiates a message exchange using the more efficient encoding.

Acquire Token First

Developer Key, Brokered Authentication [21]

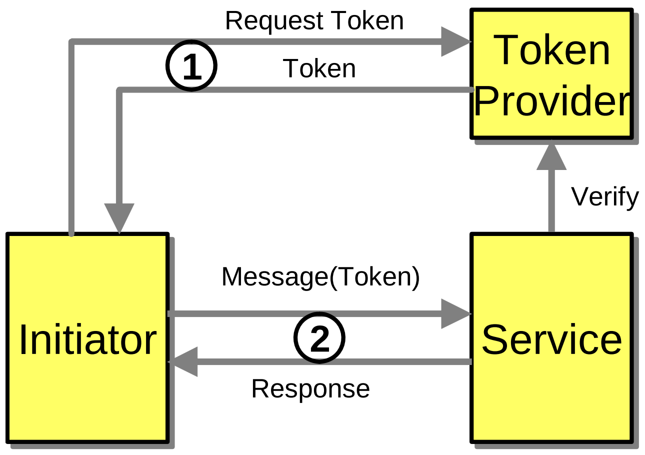
A service consumer has identified a service provider and is about to make a request. However, this service requires that consumers authenticate themselves so it can track or limit usage.

How can a service provider ascertain the identity of a requestor?

- Many service providers like to track who is using their service or limit access to their service.
- Public services may throttle the number of requests they allow from an individual user within a given time frame.

- In bidirectional channels (e.g. TCP), the originator can relatively easily fake its address, because the provider will never use that address to connect to the consumer. Therefore, a client's claimed address may not be reliable form of identification.

Have the initiator acquire a reference token first. This token has to be included in subsequent requests.



Before the initiator can interact with the service, it first has to request a token from the token service. This initial part of the conversation may happen over a separate channel. For example, public Web services may require an e-mail request to generate a developer token. The token request may have to include additional information, for example the developer's e-mail address. This portion of the conversation may include a more complex conversation, e.g. an *Address Verification*.

The token is not necessarily meant to be secure or difficult to forge. Many services use it simply to track and limit usage. Some initiators *Rotate Tokens* to increase the number of requests they can make.

Tokens can become a challenge when building a composite service, which accesses a back-end service that requires a token. A successful public service may quickly exhaust the usage quota allocated to a single token. It can *Rotate Tokens* to stretch the limit, but that is often not a scaleable approach. Alternatively, the public service can require its clients to pass in a valid token, which it propagates to the back-end service. However, this is inconvenient for the clients, as manual setup is required before they can use the service. This approach also breaks encapsulation as the client should have no or little knowledge of the back-end services the composite service uses. Lastly, some providers stop issuing tokens, so that it is impossible for the client to supply its own token (for example, Google stopped issuing tokens for its SOAP Search Service).

Some tokens can be verified without the service provider having to contact the token provider for each token submitted, for example, if the token provider and the service share a public/private key combination.

Known Partners [20] describes a robust version of this approach that can be used for services that require mutual trust. More detailed implementation guidance can be found in [21], which describes *Brokered Authentication* using X.509, Kerberos, and a Security Token Service.

Example: Amazon Web Services

Before you can make a request to Amazon's e-Commerce Web services you need to create an account and obtain a developer token, which has to be passed with all service requests.

Rotate Tokens

A service consumer acquired a token but is experiencing more traffic than the service allows a single client to create.

How can a service that requires *Acquire Token First* issue more requests than allowed by the service provider?

- Many public services limit the number of requests made by a single consumer. They may use a variety of algorithms, such as "leaky bucket", "throttling" (limiting the rate of requests), or simply limit the number of requests in a fixed period, such as a single day.
- Client applications typically identify themselves through the use of a token, which they acquire through *Acquire Token First*.

Have the consumer acquire more than one token, and use them in rotation, pretending that it acts on behalf of multiple consumers.

Rotate Tokens is an interesting variation on load balancing, such as round-robin. Instead of balancing load to different physical servers, the client rotates across multiple user accounts.

Some service providers prevent *Rotate Tokens* by making it a violation of the terms of service.

Address Verification

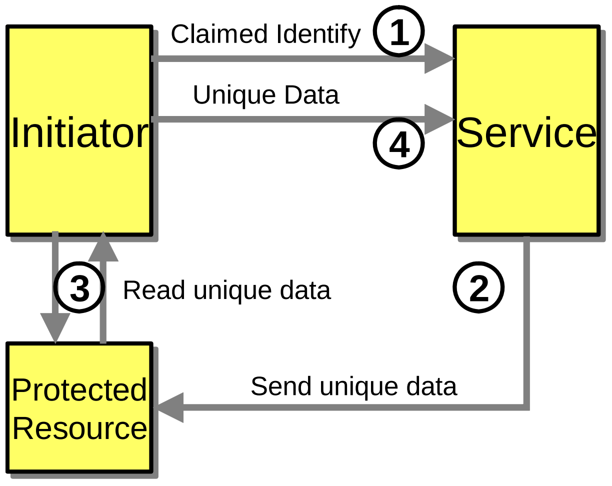
A service provider requires authentication of the initiator's identity.

In the absence of a certificate authority, how can a service provider verify the claimed identity of a consumer?

- Many authentication schemes assume the presence of a trusted authority that can issue and verify security tokens. This is not always practical (it takes time) or economical (it costs money).
- In many cases, the user's identify is linked to the ability to read data from a protected resource. Others may be able to send information to the protected resource, though. For example, identify may be established through an e-mail address.

- The service provider may reasonably assume that only the rightful owner has access to the protected resource. It is willing to trust the security scheme established by the protected resource.

The service provider sends a message with unique data to the consumer supplied address. The consumer has to pass the unique data back to service provider to complete the authentication.



The conversation consists of the following steps:

1. The initiator requests to interact with the service. It provides a claimed identify.
2. The service provider generates a unique data set and links it internally to the claimed identity. The provider then sends the unique data item to the identity claimed by the initiator.
3. The initiator accesses the protected resource linked to the identity to retrieve the unique data item.
4. The initiator provides the unique item to the service provider. The service provider matches the data item against the claimed identity and grants (or denies) access to the service.

Depending on the security requirements, the unique data set can be more or less complex so unauthorized initiators cannot "guess".

This approach is very common with Web sites. In fact, it is so common, that it is used for phishing. A user is spammed with an e-mail messages that pretends to be an *Address Verification*. The link lures the recipient to a web site where he or she may be prompted to enter private data.

Example: E-mail verification

When a user provides an e-mail address to a service, the service validates the address by sending a unique token (usually a long random number) to the provided address. To complete the sign-up process, the user has to access his or her mail, extract the token, and provide it back to the service (or click on a link embedded in the e-mail).

Example: Bank account linking

Many on-line banks allow their customer to link external accounts so that they can transfer money back and forth. When establishing such a link, the bank wants to be sure that the external account is owned by the same person, or at least that the person has full access to the external account. To do so, the bank will issue a few random, small deposits to the external account. The user has to access the external account using his or her credentials with that bank, and then provide the amount numbers to the original bank to complete the linking process.

5 Basic Conversations

Once two parties have established a connection they can begin the application-level conversation. Basic conversations consist of two conversation partners. Still, we observe a variety of different conversation patterns. These conversations differ primarily in the following design aspects:

- *One-way vs. two-way*. Some conversations need to transmit information only in a single direction, while others need to transmit information between in both directions.
- *Incoming connections*. Even if bi-directional information exchange is desired, one conversation partner may not be able to receive unsolicited incoming messages. He may however, be able to receive responses to requests he makes.
- *Single or multi-message*. Some conversations consist only of a single conversation (a boundary case), some consist of two message, while others consist of any number.
- *Natural termination*. Some conversations have a natural ending point. For example *Request-Response* always ends after the response message. Other conversations, such as *Subscribe-Notify*, keep on exchanging messages until the conversation is terminated.
- *Error Conditions*. Simple conversations differ in the way they deal with error conditions. For example, they may retry actions or simply ignore errors.

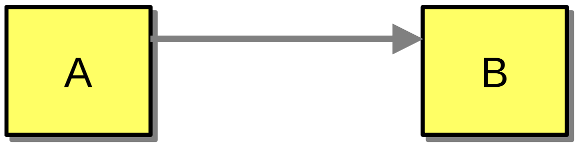
Fire-and-Forget

A service has identified a conversation partner and wants to pass information to the partner.

How can one service notify another service efficiently?

- Notifications do not require a response from the recipient.
- Some connections are inherently bidirectional, but the interaction is more efficient of the initiator does not have to wait for the result data to be sent.

The initiator sends a Fire-and-Forget message and does not expect any response from the recipient.



Fire-and-Forget is the simplest of all conversations. The conversation is stateless, because the conversation does not contain a state. Therefore, *Fire-and-Forget* provides the loosest of all possible coupling. In a *Pipes-and-Filters* architectural style [2], all components communicate using *Fire-and-Forget*. This kind of stateless interaction results in highly scaleable systems.

But the simplicity and loose coupling comes at a price. Error handling is not possible in *Fire-and-Forget*. This means the conversation has to use *Guaranteed Delivery* [2] or consider the conversation "best effort", where lost messages are acceptable. Even, if no messages are lost, errors can occur at the application level. For example, the recipient may not be able to process the message correctly due to invalid or corrupt data. Since the recipient has no way of sending message back to the initiator (it may not even know who the originator of the message is), it may route the bad message to an *Invalid Message Channel* [2].

Fire-and-Forget is most effective with asynchronous communication, which allows the sender not to wait until the message is delivered to the recipient, but instead to continue pursuing other tasks.

Example: Messaging Systems

Most messaging systems, such as JMS-compliant systems or IBM Websphere MQ provide *Fire-and-Forget* communication by default. These systems implement *Guaranteed Delivery*, so that no messages are lost even if the receiver or the messaging systems goes down.

Example: UDP Packets

On local networks, applications can send *Fire-and-Forget* messages through UDP. UDP is connection-less, and does not require a sync up between sender and receiver. However, UDP is not reliable, so this mode of communication carries "at most once" semantics for message delivery.

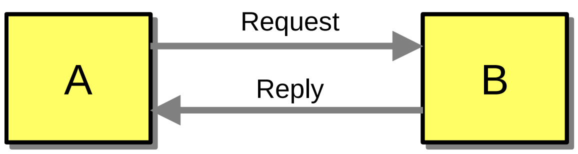
Request-Response

A service has identified a conversation partner.

How can one service request information from another service?

- Message channels transport messages only in a single direction. For a bi-directional exchange of information, two channels may be necessary.
- Some connection-oriented channels, such as TCP/IP allow sending and receiving data over one connection. Still, the exchange involves two separate messages and message formats.

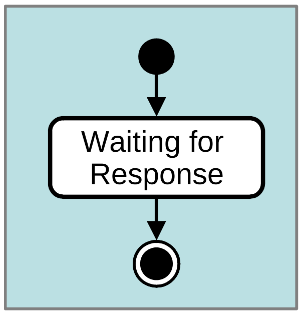
Use two messages, a request message and a response message.



This conversation is often compared to (or confused with) *Remote Procedure Invocation* (aka Remote Procedure Call, RPC) [2]. However, the two patterns' intent is quite different. RPC's purpose is to invoke a method on another component and to return the results to the invoker. *Request-Response* does not tie specific semantics to the exchange of messages. The message exchange may be used to invoke a method, query information, or the reply may simply acknowledge the receipt of a message. RPC is also often tied to a synchronous client-side programming model, which makes the conversation look like a method call on the client side. This provides convenience for developers, who are used to working with synchronous method calls, but can result in brittleness and poor performance as the simple programming model masks the inherent challenges of the networked interaction, such as latency, partial failure etc.

Request-Reply describes a conversation and does not presuppose a programming model inside the requestor or the service provider. Generally speaking, an asynchronous model is more difficult to work with because it requires the client program to manage program state explicitly as it can no longer rely on the call stack to do so. Still, an asynchronous model is better suited to asynchronous, long running (when compared to a local method call) interactions.

When using *Request-Response* over asynchronous message channels, the similarity to RPC can be deceiving. Because the conversation consists of two independent messages, the requestor is responsible for matching the response message to the original request. The asynchronous nature of the conversation allows the client to engage in more than one *Request-Response* conversation at one time. In this case, response messages may arrive in a different order than the requests because some requests may be processed more quickly (or by a different service instance) than others. Therefore, it is important to consider this seemingly simple interaction a full-fledged conversation and to equip messages with a unique *Correlation Identifier* [2], which ties the messages to the conversation.



The *Request-Response* conversation has a single state, a "Waiting for response" state. The service provider does not have to keep a conversation state because all it does is process a request and send a response message. This makes the *Request-Response* conversation stateless from the perspective of the service provider, meaning subsequent conversations can be services by different instances of the service provider.

The initiator can avoid tracking the conversation state if the response message contains all data necessary for the initiator to process it. If the initiator does not have to associate any internal state with the incoming message, the response message can be seen as an independent *Fire-and-Forget* message instead of a response message associated with a *Request-Response* conversation. Some service providers allow clients to "pass through" additional data of their choice so they do not have to keep local state. While it can make the conversation "stateless" (all relevant state resides inside the message), it can also cause security concerns and increase network traffic.

If the response message is self-contained, it becomes irrelevant whether the original initiator receives the response or whether it's sent to a third service. In such a case, the conversation turns into a *Pipes-and-Filters* [2].

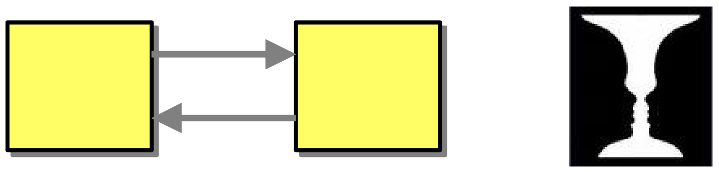


Figure 8: Optical Illusion: *Request-Response* or two *Fire-and-Forget* conversations?

The similarity between the implementation of *Request-Response* and the use of two *Fire-and-Forget* interactions highlights the fact that the patterns convey more than simply the technical solution -- they convey the intent behind the solution. If the fact that the response message returns to the original initiator is purely coincidental, one would express the interaction as two *Fire-and-Forget* interactions, highlighting the fact that the sameness of initiator and recipient of the second message is coincidental and is not core to the intent of the solution.

Typically, a single service provider is engaged in *Request-Response* conversations with different initiators. If two separate channels are used for the conversation, the initiators pass a *Return Address* to the service provider, so that the provider can send the response message to the specified address.

From the initiator's point of view, four error conditions can occur:

1. A reply message may never arrive, or not arrive within an expected time interval. This can happen because channels are not reliable, very slow, or because the service provider failed. The initiator cannot tell, which of the possible scenarios occur, causing uncertainty about the state of the provider. To address the error condition, the consumer can either resend the message, turning the conversation into *Request-Response-Retry*, or ignore it using the *Do Nothing* approach.
2. The response message can indicate an error condition. This may happen for example if the request contains invalid data and the service provider indicates that it did not process the request.
3. The response message may be ill formed or otherwise invalid, preventing the initiator from processing it successfully.
4. Because request and response message travel over separate channels, the consumer may receive a response to a different request than expected. In some cases it may even receive a (possible well-formed) response to a request that it did not even make. This can happen for example, if the channel is implemented as a persistent message queues and a faulty client crashed before it consumed the response for a request it sent. When the client restarts and sends a new request, it may be reading the previous response from the channel.

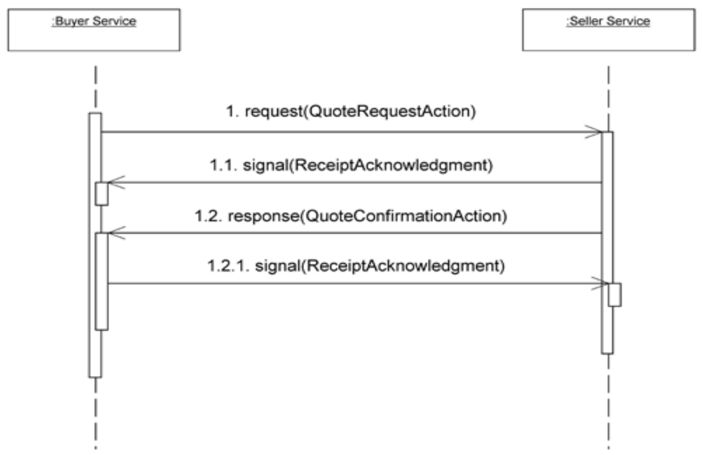
Example: Web Service Invocation over HTTP

Single channel, two messages.

Example: RosettaNet Two-Action Activity (Asynchronous)

Consists of 4 messages, including Ack's. Examples: PIP3A1: Request Quote,

PIP3A2: Request Price and Availability.


:Buyer Service
:Seller Service
1. request(QuoteRequestAction)
1.1. signal(ReceiptAcknowledgment)
1.2. response(QuoteConfirmationAction)
1.2.1. signal(ReceiptAcknowledgment)

Each message requires a *Quick Acknowledgment*. If we consider the acknowledgment part of the transport layer, then this looks exactly like *Request-Response*.

Request-Response with Retry

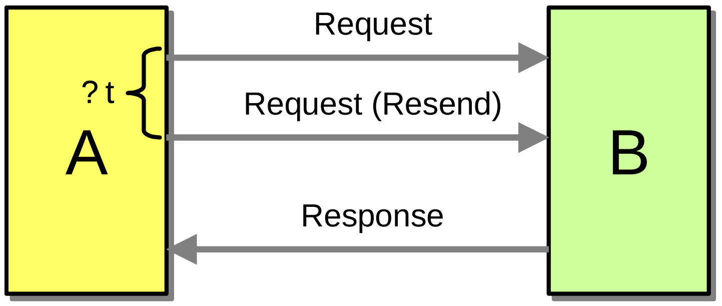
Reliable Delivery

Two services engage in a *Request-Response* conversation.

How can a service consumer deal with a missing response message in a Request-Response conversation?

- Because *Request-Response* uses separate messages for the request and the response, the initiator may expect a response message, but never receive it.
- Remote communication can fail for a number of reasons, many of which may be intermittent. For example, a network router may be unavailable, or service may not be operating. Such circumstances can cause a valid operation to fail some time and succeed another time.
- If both initiator and provider are *Transactional Clients* [2] in combination with *Guaranteed Delivery*[2], no messages are lost – the message is either in the request channel, being processed by the service, or in the reply channel. However, using this type of interchange requires three transactions: one to enqueue the request, one to process it, and one to consumer the response [19].
- Even if no message is lost, the initiator may become "impatient". For example, the request may be processed by a very slow service provider. The initiator may get a faster response by sending a new request.

Have the consumer retry the request the request if it does not receive a response within a certain time interval. Maybe both the initiator and the service idempotent so they can deal with duplicate messages.



The conversation originally is analogous to *Request-Response*. However, the initiator uses a time-out condition to decide how long to wait for a response. If the response does not arrive, it resends the request.

If the time-out resulted from a lost (or delayed) response message, the service provider now receives the same request message a second time. If we want to keep the provider from perform the requested action a second time, it has to be an *Idempotent Receiver* [2], i.e. it must be able to distinguish a resent message from two distinct requests that just happen to contain the same data. To aid the service in detecting duplicates, the initiator can equip the resent message with the same *Correlation Identifier* [2]. If the provider has in fact processed the message before, it can skip the requested operation and simply return the previous response message.

Unless the service operation is inherently idempotent (such as a read operation), this conversation is no longer stateless from the service provider's perspective. The provider needs to keep a list of received conversation IDs, and possibly a list of already sent responses.

The initiator has to deal with duplicate messages as well. The service provider might have sent a response through a message queue just as the consumer decided to give up waiting. In this case the consumer resends the request message, just to receive the original response message a fraction of a second later. This makes the consumer happy, but a little while later it will receive another response message, based on the resent request. Most likely the consumer should ignore this message as the response has already been processed, requiring the consumer to be idempotent as well.

Another consideration is a failed client. If a client fails and restarts, can it detect the state of the conversation? If all parties are *Transactional Clients* [2], and the message channel can provide information as to which message IDs have been pushed to the channel, a client can recover the conversation state without resending messages ([19]). If the system is not transaction, the client would likely resend the message.

Error recovery from conversations is a complex topic. *Request-Response with Retry* is the application of the *Retry* error recovery pattern in a *Request-Response* conversation. For more detail on error handling strategies see Ensuring Consistency

Example: RosettaNet RNIF 2.6.6

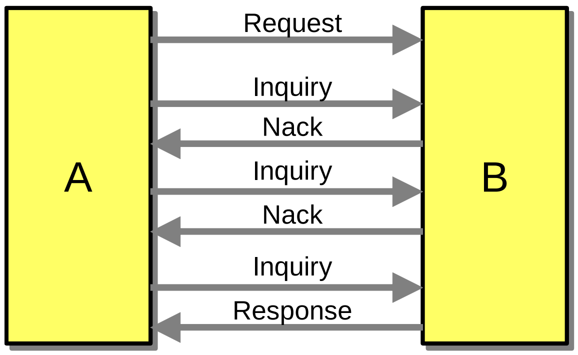
Polling

A service has identified a conversation partner and would like to request information from it.

How can a service request information from another service if it cannot accept unsolicited incoming messages?

- Some transport layers allow a service to receive data, but only in the context of a connection it established, i.e. it cannot receive unsolicited incoming connections. This is the case for many connection-oriented protocols, such as TCP/IP. Making a connection is inherently different from processing incoming connections. Some services may not be programmed to receive connections.
- Even if a service is programmed to receive incoming connections, the network topology may prevent such a connection. This is common in cases where the initiating service resides behind a firewall that prevents incoming connections.
- Even if the initiator is able to receive incoming messages or connections, its application program may not be interested in doing so. This can be the case if the initiating service can continue processing up to a point where it requires the data from the service provider. In this case, it is a more natural programming model for the initiator to make a request, continue processing up to a synchronization point, the obtain the response from the external service.

Have the originating service poll, i.e., inquire whether the results are ready, multiple times if necessary.



The response to the last inquiry can include the response message or indicate that the results are available, requiring the initiator of the conversation to make a separate request to retrieve the results.

Polling is significantly more "chatty" than *Request-Response*. It should only be used when either of the communicating parties cannot support the *Request-Response* conversation pattern.

Polling requires the service provider to keep state at the convenience of the requestor. Keeping state should be considered a form of coupling and can limit the scalability of the service provider if responses consume large amounts of memory and initiators are slow to pick up their results. Ultimately, the service provider may use additional patterns to terminate the conversation, such as a *Lease*, so it can relinquish the allocated storage if the initiator never consumes the results.

Queuing can often eliminate the need for *Polling*. The initiator can consume results whenever it pleases, alleviating the provider to keep results until the initiator picks them up. Storing results in a message queue is generally more efficient than storing it in a server. The initiating service can also use a *Poll Object* [10] to implement polling internally, turning it into a *Polling Consumer* [2]. This allows the consumer application to control when the message is read while still using the preferred *Request-Response* conversation with the service provider.

Subscribe-Notify

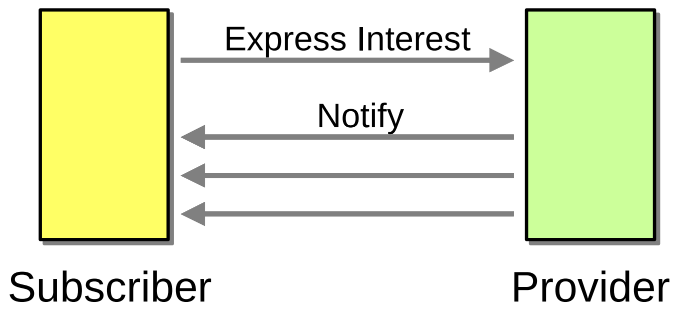
Multi-responses [3]

A service wants to retrieve information from another service. However, the response cannot be packed into a single message.

How can one service receive information from another service, if that information cannot be packaged into a single response message?

- A service provider could aggregate individual messages into a larger message. However, the message data may be time sensitive, reducing the message's value with any further delay.
- The response data may not be a finite set of data but a continuous stream of data.

The interested service initiates a *Subscribe-Notify* conversation by sending a message, which expresses its interest. In response, the provider sends a series of messages to the initiator.



Subscribe-Notify assumes that the originator can receive inbound messages. This can be a challenge in situations where the initiator sits behind a firewall. In these cases, the conversation has to fall back to the initiator issuing repeated *Request-Response* conversations.

Subscribe-Notify makes more efficient use of network resources than repeated *Request-Response*. Not counting administrative messages, such as renewals, *Subscribe-Notify* transmits only half as many messages as a repeated *Request-Response*.

Unlike the previous patterns, this conversation does not have a natural stopping point. It therefore depends on a stopping condition. Typical stopping conditions are:

- The subscriber sends a stop request
- The conversation uses a *Lease*, causing the conversation to stop after a deadline is reached without the subscriber renewing interest.
- If the channel is connection-oriented (such as an HTTP connection), the provider stops sending messages when the subscriber does not accept a connection.
- The provider may notify the subscriber of end of transmission.
- The provider may simply stop sending messages, leaving the subscriber to realize this via a time-out condition.

Example: Streaming Results from a Long-running Query

When a service consumer invokes a query that takes a long time, some of the results may be available before the query completes (e.g. if no sort is required). In this case, the service provider can start sending partial results before the whole operation completes. The client can now start processing the result data sooner, possible providing a better user experience.

Example: Stock Ticker Subscription

A service may want to receive frequent updates to the price of a security. Instead of repeatedly polling, the service should subscribe and receive the price updates as *Fire-and-Forget* messages.

Exampe: WS-Eventing and WS-Notification

Different approaches: WS-Eventing uses single URL, not meant for pub-sub topics. WS-Notification more complex, includes topic hierarchy.

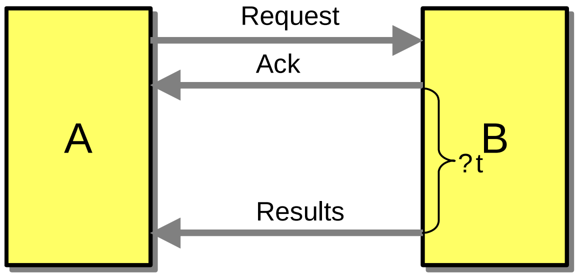
Quick Acknowledgment

Some operations provided by a service provider may take a long time, leaving the client in a state of uncertainty.

How can a client be sure a service accepted a request, even if the request takes a long time?

- A good rule is "fail fast", i.e. detect failures as soon as possible. A service should return an error message as soon as possible so that the consumer is aware of the failure.
- Likewise, the consumer might want to know that things are going well.

Send an acknowledgment message quickly, followed by the actual results later.



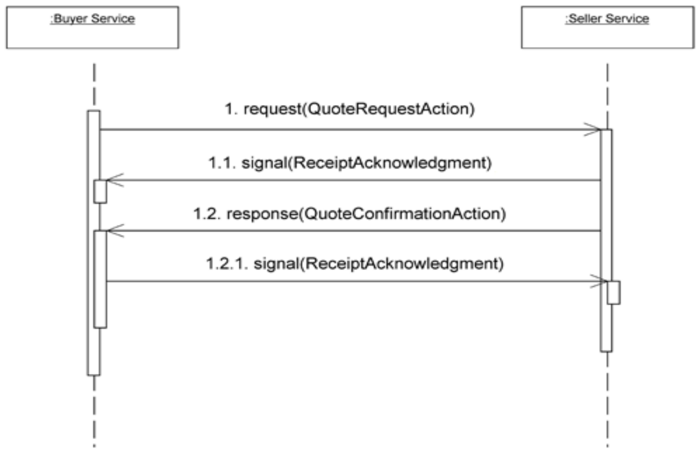
The acknowledgment message typically contains additional information, such as a *Correlation Identifier* [2], which can be used to refer to the same conversation in subsequent messages. Some communication standards use quick acknowledgment messages for all interactions, including the sending of the result (see RosettaNet example below).

Example: Ordering from Amazon

Most on-line stores such as Amazon use *Quick Acknowledgment* by sending an immediate confirmation both as part of the HTTP request, as well as per e-mail ("Thank you for your order"). The acknowledgment also contains an order number that serves as *Correlation Identifier*. Time-consuming activities such as stock availability checks or credit card verification are performed after the acknowledgment. Additional e-mail messages notify the initiator of progress or potential error conditions.

Example: RosettaNet Request Quote

RosettaNet message exchanges always require an acknowledgement message from the recipient, even if the transport is a bidirectional, connection oriented channel such as TCP/IP. The acknowledgment message does more than indicate the availability of the service, but also confirms that the recipient has successfully parsed and validated the message schema.


:Buyer Service
:Seller Service
1. request(QuoteRequestAction)
1.1. signal(ReceiptAcknowledgment)
1.2. response(QuoteConfirmationAction)
1.2.1. signal(ReceiptAcknowledgment)

6 Multi-Party Conversations

While many conversations involve only two parties, more complex conversations can involve multiple parties. When designing multi-party conversations, the following considerations come into play:

- *Coordinator*. Some conversations have an explicit coordinator, for example a transaction monitor.
- *Peers*. Some multi-party conversations are held amongst equal peers, i.e. each participant plays (or at least starts in) the same role. Other conversations have a clear division of roles.
- *Limited / unlimited*. Some conversations have multiple participants, but the number is limited. Other conversations can be amongst an essentially unlimited number of participants.
- *Fixed number of participants*. Some conversations allow participants to join or leave the conversation while others require that the number of participants remains constant for the duration of the conversation.

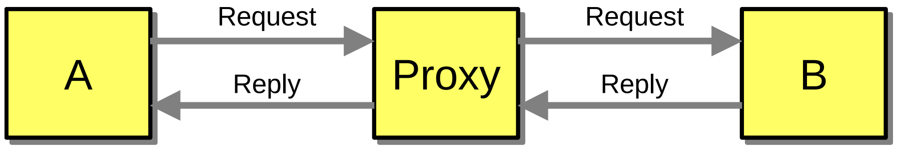
Proxy

A service wants to engage in a conversation but its intended conversation partner may not be reachable or known.

How can a service communicate with a service that is not reachable?

- Not all services may be reachable directly across a widely distributed network.
- A service may desire to stay anonymous by not connecting so a service directly.

Relay all messages through a Proxy, a service that forwards messages between two services.



A *Proxy* can increase the latency because two hops are necessary.

The *Proxy* needs to deal with time-out and error conditions.

A *Proxy* can support a variety of conversations, with Request-Reply naturally being the simplest.

Example: Cross-domain Proxy for AJAX Applications

JavaScript running in a Web browser can only connect to service in the domain where the script originated ("same origin policy"). Many mashup and other AJAX applications need to access external data sources, For example, a housing map needs to access a housing listing service. To gain access to this external service, the JavaScript connects to a *Proxy*, which resides in the same domain as the JavaScript source. The proxy relays requests to the external service. (see Ajax Design Patterns)

Relay

Two participants want to engage in a two-way conversation. However, the services may be part of a network that only allows outbound communication due to security policies.

How can two services engage in a two-way communication when each service is limited to outbound requests?

•

The participants communicate through a third participant, a *Relay*. Both participants connect to the Relay through outbound communication. The Relay forwards messages from one participant to the other.

<<Sketch>>

Useful when services are behind firewall.

Need to trust the relay. May not be certain who you are talking to.

Example: Biztalk Relay Service

http://labs.biztalk.net/Connectivity.aspx

Leader Election

Multiple conversation participants act as peers, but require the existence of a leader.

How can a number of conversation participants agree on electing a single leader?

- Leader election is a common distributed algorithm problem [22] with a number of proven algorithms.

<<need simple way to express algorithm>>.

Leader election is needed in a variety of situations. For example, a distributed caching approach requires one party to be the "master", propagating data to the other caches. If the master fails, the remaining participants have to elect a new master.

7 Ensuring Consistency

In a service-oriented environment, operations commonly span across multiple, independent services. Consistent execution of the individual interactions is usually required, i.e. all operations should succeed, or, if something goes wrong, no operation should be completed. In a tightly coupled and controlled environment, transaction mechanisms usually ensure this type of consistency, guaranteeing the classic ACID properties: operations are atomic, consistent, isolated, and durable. In widely distributed, loosely coupled systems, however, such ACID transactions are either not available or not practical. Some people even proclaim that the "New ACID Properties" for such systems are Associative, Commutative, Idempotent, and Distributed (*need reference*).

In the absence of traditional ACID transactions, services have to engage in conversations to ensure consistency. The approaches try to balance the following considerations:

- *Reducing uncertainty*: When a system communicates with another system over a potentially unreliable channel, it is inherently uncertain about the other system's state. For example, if a service consumer does not receive a response to a request, it cannot be certain whether the received or processed the request.
- *Mitigating risk*: In the absence of traditional ACID transactions, interacting services have to accept the fact that consistency cannot be achieved in all cases. Therefore, the conversation should try to minimize the probability and potential downside of inconsistent outcomes.
- *Optimistic vs. Pessimistic*: Interacting services can choose to execute optimistically, optimizing for the case where everything goes well, gaining simplicity and high throughput, but accepting the downside of having to deal with difficult to recover or complex failure scenarios. Alternatively, services can be pessimistic, trying to minimize the frequency and severity of failure scenarios at the expense of slower throughput or higher complexity.
- *Idempotency*: One of the simplest strategies to cope with failed operations is to retry the operation. In the face of uncertainty across distributed systems this can mean, though, that a service is asked to repeat an already successfully executed operation. Idempotent services recognize such a situation and avoid duplicate execution.
- *Certainty vs. Complexity*. More involved conversations, including multiple acknowledgements can increase the likelihood of a consistent outcome, but also increase the complexity of the interaction, which may reduce throughput.
- *Detecting errors*. Due to the inherent uncertainty across services, detecting an error condition itself may prove difficult, even more so if external systems are involved. For example, how does one detect that a letter sent with regular mail did not arrive?

To simplify the discussion of forces and resulting contexts in these patterns, we typically assume a simple Request-Response interaction between a service consumer and a service provider. However, the same patterns and strategies apply to more complex conversations.

Include error detection patterns? e.g., time-out, error response, acknowledgment

Errors at different levels: transport, message syntax, business validation

Contingent Request is also an error handling strategy

Split this chapter into 2 parts: error handling and ensuring consistency?

Ignore Errors

A service consumer is interacting with a service. The requested operation may fail.

How can a service consumer cope with a failing operation?

- Error handling can be complicated
- Detecting errors with certainty may be difficult. For example "time out" is only an approximation for detecting errors and may cause false negatives.
- Operations may be irreversible, so if one operation is completed and another one fails, there is not much the service can do.
- Most conversations cannot ensure 100% consistency anyway, so accepting a failure is reality
- Not dealing with failure can increase throughput for successful scenarios.

The service consumer ignores the failure and continues as if nothing happened.

<<Sketch>>

While this "solution" may not seem like an acceptable solution to many engineers, in a real business context, this approach can often be the most economical. Building a system to handle errors automatically may be more expensive than the losses incurred from ignoring errors. Naturally, when computing the cost of not handling errors, one has to include the total cost, for example including loss in customer satisfaction due to poor or inconsistent service.

Even though *Ignore Errors* does not affect the conversation between service consumer and service provider, this does not mean that the consumer has nothing to do. The failed conversation may be part of a larger operation, which may have to take a different path due to the missing results from the service. Likewise, the service consumer may have to de-allocate resources related to the operation.

Example: Starbucks

A coffee shop may have already started preparing a coffee for a customer when it turns out that the customer has insufficient funds. Simply discarding the coffee and processing other customers' orders is generally more efficient than trying to obtain the funds.

Shrug Shoulders, Write-off

Retry

A service consumer is interacting with a service. The requested operation may fail.

How can a service consumer cope with a failing operation?

- Remote communication can fail for a number of reasons, many of which may be intermittent. For example, a network router may be unavailable, or service may not be operating. Such circumstances can cause a valid operation to fail some time and succeed another time.
- If a service consumer makes an invalid or incorrect request, the service provider may respond with a specific error message, allowing the consumer to understand the cause of the failure. However, not every service may provide this capabilty.
- Some service requests can cause a service provider to crash. Such “poison messages” can cause a service provider to fail every time it receives such a request. The service consumer may not be aware that the message causes the service to crash and may mistake the failure for a network failure.

The service consumer retries the failed operation until the number of retries reaches a certain limit.

<<Sketch>>

As indicated in 1.1, allowing retry requires both the service consumer to be *idempotent*, i.e. they have to be resilient against duplicate messages. Message duplication can occur because resending inserts a repeated message into the conversation. If the original message was in fact already received by the service provider, the retry message could cause the service operation to be executed twice.

Because of the possibility of poison messages, the service consumer should limit the number of retries. Otherwise the consumer may continue to crash the service provider continuously. This implies that the application layer, which makes the service request, has to accept the fact that remote operation may fail despite retries.

Expanding retry intervals

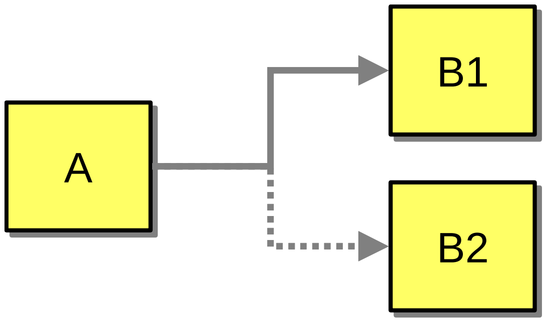
Contingent Requests

A service consumer interacts with a service provider.

How can a service get a response, even if the service provider is unavailable?

- Resending a request to the same service provider may not be successful if the service provider is unavailable.
- In many service-oriented architectures more than one service may be able to fulfill the initiator's needs.

The service consumer issues a request to one service provider. If it does not receive a response within a certain timeframe, it sends the request to another service provider.



The initiator has to deal with multiple responses because some of the earlier requests can generate responses after the initiator already decided to contact an alternative provider.

The initiator has to have a list of multiple potential providers. It might use *Dynamic Discovery* and keep the list of all possible candidates.

- See *Service Interaction Patterns*

Tentative Operation

A service consumer is interacting with multiple, independent services. This interaction includes side effect-ful activities, such as allocating a resource, making a payment, etc.

Compensating Action

A service consumer is interacting with multiple, independent services. This interaction includes side effect-ful activities, such as allocating a resource, making a payment, etc.

How can the service consumer ensure a consistent outcome across multiple, independent resources?

Reaching Agreement

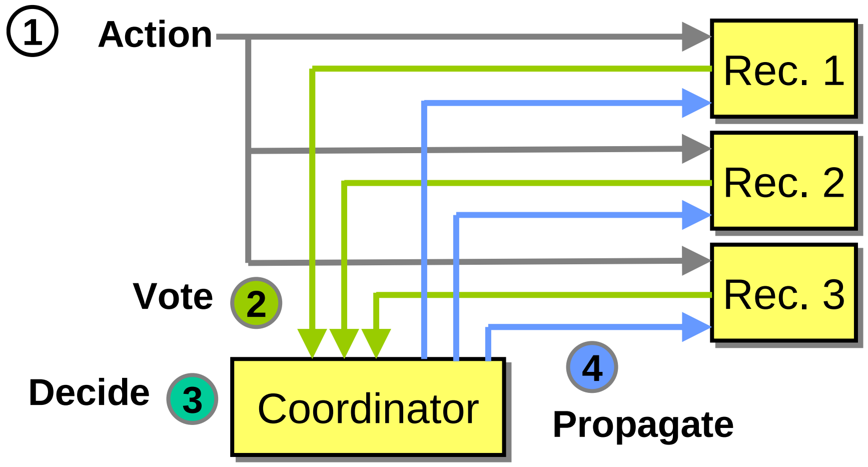
Multiple participants each perform a stateful action or allocate some resource.

How can you reach agreement across a number of independent parties, so that all participants see the same outcome?

- Interactions between conversation participants typically do not offer ACID style transactions because the interactions can span across a long time span.
- However, many interactions require coordination, such that all participants should perform a specific action, or none of them. The classical example is making a travel

reservation. You don't want a hotel reservation if you cannot get a flight ticket or vice versa.

Have a separate coordinator. The coordinator collects responses from each participant, determines the outcome, and informs each participant of the outcome.



This is a more generic case of XA transactions. XA uses a specific coordination rule: only if all Recipients say OK, send a Commit. Otherwise roll back.

Many other strategies are possible: majority vote, minimum count, etc. This is optimized for the pessimistic case: lots of traffic, but robust. WS-Coordination covers this general case.

Example: Scheduling a Meeting

The "resource" being allocated by the participants may be time, for example when scheduling a meeting. A coordinator may prompt each participant as to their availability. Once the coordinator has that information, he or she picks a suitable time based on one of the strategies mentioned above, for example, a majority decision.

Perform Most Likely to Fail Action First

Perform Hardest to Revert Action Last

Example: eBay

eBay does not use DB transactions. When they write parent-child records, they write children first. If the parent write failed, they have some garbage data, but no inconsistent data.

Example: Book Hotel before Flight

A lot of flights have cancellation penalties, hotels do not. Book the hotel before the flight, so if you can't get a flight, you have no loss.

8 Reaching Agreement

Reaching agreement in a business-to-business setting can be challenging. Representing the negotiations, cancellations, amendments, in a system can be equally challenging. The conversation patterns in this section tie stronger semantics to the individual messages than most of the other chapters. While most of the basic conversations deal with generic "request" and "response" messages, reaching agreement between parties requires a common understanding amongst the parties as to the meaning of individual messages. For example, an "offer" message is different from a "cancellation" message. The conversation policies for some of these conversations resemble the policies of some basic message flows. For example, *Implicit Accept* resembles *Fire-and-forget*. However, the context and intent of the patterns are different.

Our patterns are not concerned with the messages' specific data formats. In an implementation of the patterns, the generic placeholder messages would be replaced with concrete message types, depending on the application domain.

When designing conversations to reach agreement, the following design considerations have to be considered:

- *Symmetry*. Some conversations allow one party to make proposals, allowing the other party to accept or decline. Other conversations allow both parties to make proposals and counter-proposals.
- *Acceptance*. Acceptance can be explicit (by sending a message) or implicit (by not protesting).

THIS SECTION IS UNFINISHED. PLEASE IGNORE THE REST OF THE SECTION.

Multi Party Transaction Paper:

http://www.choreology.com/downloads/2002-11-14.Multi-party.BusinessTransactions.pdf

Receiver Cancels

Receiver can send cancellation within certain interval

- Receiving party can cancel within certain timeframe

Sender Cancels

Sender can send cancellation / change within certain interval (or until next message arrives)

Binding Request

Binding Offer

9 Resource Management and Termination

Some conversations do not have a natural termination point. For example, *Subscribe-Notify* continues indefinitely, unless the subscriber or the provider explicitly terminate it. Many other interactions require service providers to allocate resources, such as storage, connections, or money for a service consumer. In a loosely coupled setting, the provider should not hold these resources indefinitely, but make sure the consumer is still present and does still require the resources. Otherwise, the service may hold resources for no longer existing clients and run out of resources before too long.

The following considerations play an important role in conversations that deal with resource management.

- *Explicit vs. implicit termination*. A conversation can be explicitly terminated by sending a specific message or implicitly, e.g. through a period of inactivity.
- *Who can terminate?* Some conversations can be terminated by any party while others can only be cancelled by a specific party.
- *Division of responsibility*. The responsibility to keep a conversation alive can rest with the initiator or the service provider, or both.
- *Notification vs. silent termination*. One party may decide to terminate the conversation without notifying the other party. The other party may find out implicitly or by encountering an error when it attempts to continue the conversation.

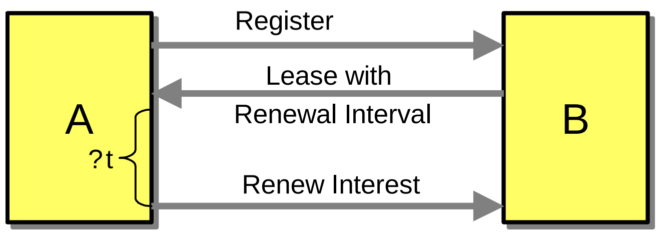
Lease

A service provider can engage in long-running conversations with multiple consumers. Each of these conversations requires the provider to allocate resources.

How can a service provider avoid holding resources for clients who are no longer interested?

- Service consumers may voluntarily or involuntarily terminate a conversation without notifying the service provider. For example, a connection may drop.
- A service provider may not be able to contact consumers to see whether they are still reachable. This could be the case because the consumers are not able to receive inbound connections.
- If a provider keeps on holding resources for clients who may no longer be interested or even running, it will waste these resources and ultimately run out.
- A service provider may want to gauge how long it is willing to hold resources for a consumer, depending on its current resource usage.

Have the service provider issue a lease to the resources. If not renewed, the lease will expire, allowing the provider to free allocated resources.



When the resource consumer first issues a request that requires the allocation of resources, the resource provider specifies a time frame until which it is willing to keep the resource. The resource provider can change this time frame with every lease it grants, based on attributes of the requestor or based on its own state, e.g., the number of resources available.

The consumer of the service is also referred to as the "holder" of the lease while the service is called the "grantor". The lease holder can actively cancel the lease before it is up to help the grantor release resources.

The lease holder can apply for a new lease before the existing one runs out. The lease grantor can deny such a request, even though in reality this is an unlikely scenario.

Following this approach the lease holder is always certain about the state of the lease. For example, if a "Renew Interest" request fails (e.g. because the lease grantor is unreachable), the lease holder knows that the grantor will relinquish the lease.

This conversation puts the burden for lease renewal on the lease holder.

Example: Jini

Jini is designed for dynamic service environments where services can come and go quite frequently. Therefore, Jini employs the lease patterns for any kind of resource allocation,

Example: DCHP

A DCHP server assigns IP addresses to machines on a temporary basis. Machines have to renew their assigned IP address. Otherwise, the server may reclaim it and assign it to a different machine.

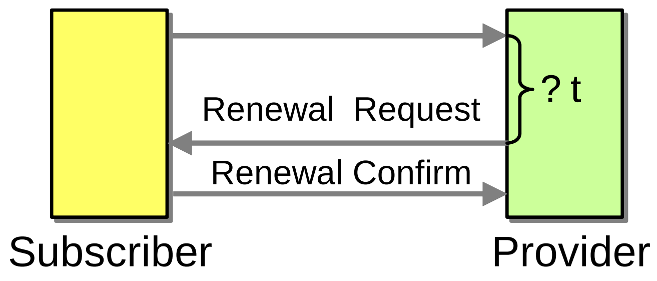
Renewal Reminder

A service provider can engage in long-running conversations with multiple consumers. Each of these conversations requires the provider to allocate resources.

How can a service provider ensure that a service consumer is still present and still requires resources to be allocated?

- A *Lease* allows the grantor to determine the length of the lease at the time it is issued. If the provider gets into a resource crunch it has no way to revoke leases that have already been granted.
- Lease requires the lease holder to actively monitor the duration of the lease. This might be cumbersome for some clients.
- Clocks may get out of sync, causing a lease holder to believe the lease is still valid, while the lease grantor already expired the lease.

Have the provider periodically check with the consumer whether it still requires resources. If the consumer answers "no" or is unreachable, the provider can relinquish the allocated resource.



Renewal Reminder gives the provider more control but also burdens it with more house keeping. The consumer can be simpler as it does not have to track the expiration of the lease. It does, however have to be able to respond to incoming requests.

If resources are plentiful, this conversation can result in very low traffic as the provider will have little need to ask for renewals. As the provider gets into a space crunch it can react by requesting renewals, hoping that some clients are no longer present.

If a client is unreachable, the provide can decide to retry the request or relinquish the resource on the spot. This can leave provider and consumer in an inconsistent state. The consumer, not aware that the provider tried to contact it, might be under the impression that the resource is still available whereas the provider has already cancelled it.

Example: Magazine Subscription

A magazine likes you to renew their resource. Therefore they send you plenty of reminders.

Heartbeat

Needs to send message every so often… See Ajax Patterns

10 References

[1] *Achieving Loose Coupling*, David Orchard, http://dev2dev.bea.com/pub/a/2004/02/orchard.html

[2] *Enterprise Integration Patterns*, Hohpe, Woolf, 2003, Addison-Wesley, http://www.eaipatterns.com

[3] *Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection*. A. Barros, M. Dumas and A. ter Hofstede: Technical Report FIT-TR-2005-02, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, March 2005.

[4] *The Byzantine Generals Problem*, Lamport, Shostak, Pease, ACM Transactions on Programming Languages and Systems, Volume 4 , Issue 3

[5] *Notes on Database Operating Systems*, Jim Gray, in *Lecture Notes In Computer Science; Vol. 60*, Springer Verlag, 1978

[6] http://dev2dev.bea.com/pub/a/2002/06/SOAPConversation.html Attempt at describing conversation headers

[7] *Web Services Choreography Description Language*, http://www.w3.org/TR/ws-cdl-10/

[8] *Contracting Workflows and Protocol Patterns*, Andries van Dijk, in W.M.P. van der Aalst et al. (Eds.): BPM 2003, LNCS 2678, Springer, 2003.

[9] *Workshop Report: Conversation Patterns*, Hohpe, Dagstuhl Seminar Proceedings 06291: The Role of Business Processes in Service Oriented Architectures, 2006, http://drops.dagstuhl.de/opus/frontdoor.php?source_opus=828

[10] *Remoting Patterns*, Völter, Kircher, Zdun, Wiley, 2004

[11] *Pattern Language for Service Discovery*, in *Pattern Languages of Program Design 5*, Manolescu, Voelter, Noble, 2006, Addison-Wesley

[12] *WS-BPEL, Business Process Execution Language*, http://www.oasis-open.org/committees/document.php?document_id=18714

[13] *Multi-Party Electronic Business Transactions*, Bob Haugen, Fletcher, http://www.choreology.com/downloads/2002-11-14.Multi-party.BusinessTransactions.pdf

[14] *Design Patterns*, Gamma, Helm, Johnson, Vlissides, Addison-Wesley

[15] *Agent Communication Languages*: The Current Landscape, Labrou, Finin, Peng, IEEE Intelligent Systems March / April 1999. http://www.flacp.fujitsulabs.com/~yannis/publications/ieeeIntelligentSystems1999.pdf

[16] Issues in Agent Communication, Dignum, Greaves (eds), LNCS 1916, Springer, 2000

[17] *KAoS: An Open Agent Architecture Supporting Reuse, Interoperability, and Extensibility*, Bradshaw, http://ksi.cpsc.ucalgary.ca/KAW/KAW96/bradshaw/KAW.html

[18] RosettaNet Implementation Framework: Core Specification, V02.00.01, http://rosettanet.org

[19] *Principles of Transaction Processing*, Bernstein, Newcomer, 1997, Morgan Kaufman

[20] *Security Patterns*, Schumacher et al, 2006, Wiley

[21] *Web Service Security: Scenarios, Patterns, and Implementation Guidance*, 2006, Microsoft Corporation

[22] *Distributed Algorithms*, Lynch, 1996, Morgan Kaufmann

[23] *Life Beyond Distributed Transactions: an Apostate's Opinion*, Helland