TABLE 6.2 General Interoperability Scenario

Portion of Scenario	Possible Values
Source	A system initiates a request to interoperate with another system.
Stimulus	A request to exchange information among system(s).
Artifact	The systems that wish to interoperate.
Environment	System(s) wishing to interoperate are discovered at runtime or known prior to runtime.
Response	One or more of the following: ▪ The request is (appropriately) rejected and appropriate entities (people or systems) are notified. ▪ The request is (appropriately) accepted and information is exchanged successfully. ▪ The request is logged by one or more of the involved systems.
Response Measure	One or more of the following: ▪ Percentage of information exchanges correctly processed ▪ Percentage of information exchanges correctly rejected

SOAP vs. REST

If you want to allow web-based applications to interoperate, you have two major off-the-shelf technology options today: (1) WS* and SOAP (which once stood for "Simple Object Access Protocol," but that acronym is no longer blessed) and (2) REST (which stands for "Representation State Transfer," and therefore is sometimes spelled ReST). How can we compare these technologies? What is each good for? What are the road hazards you need to be aware of? This is a bit of an apples-and-oranges comparison, but I will try to sketch the landscape.

SOAP is a protocol specification for XML-based information that distributed applications can use to exchange information and hence interoperate. It is most often accompanied by a set of SOA middleware interoperability standards and compliant implementations, referred to (collectively) as WS*. SOAP and WS* together define many standards, including the following:

- *An infrastructure for service composition.* SOAP can employ the Business Process Execution Language (BPEL) as a way to let developers express business processes that are implemented as WS* services.
- *Transactions.* There are several web-service standards for ensuring that transactions are properly managed: WS-AT, WS-BA, WS-CAF, and WS-Transaction.
- *Service discovery.* The Universal Description, Discovery and Integration (UDDI) language enables businesses to publish service listings and discover each other.

- *Reliability.* SOAP, by itself, does not ensure reliable message delivery. Applications that require such guarantees must use services compliant with SOAP's reliability standard: WS-Reliability.

SOAP is quite general and has its roots in a remote procedure call (RPC) model of interacting applications, although other models are certainly possible. SOAP has a simple type system, comparable to that found in the major programming languages. SOAP relies on HTTP and RPC for message transmission, but it could, in theory, be implemented on top of any communication protocol. SOAP does not mandate a service's method names, addressing model, or procedural conventions. Thus, choosing SOAP buys little actual interoperability between applications—it is just an information exchange standard. The interacting applications need to agree on *how to interpret* the payload, which is where you get semantic interoperability.

REST, on the other hand, is a client-server-based architectural style that is structured around a small set of create, read, update, delete (CRUD) operations (called POST, GET, PUT, DELETE respectively in the REST world) and a single addressing scheme (based on a URI, or uniform resource identifier). REST imposes few constraints on an architecture: SOAP offers completeness; REST offers simplicity.

REST is about state and state transfer and views the web (and the services that service-oriented systems can string together) as a huge network of information that is accessible by a single URI-based addressing scheme. There is no notion of type and hence no type checking in REST—it is up to the applications to get the semantics of interaction right.

Because REST interfaces are so simple and general, any HTTP client can talk to any HTTP server, using the REST operations (POST, GET, PUT, DELETE) with no further configuration. That buys you syntactic interoperability, but of course there must be organization-level agreement about what these programs actually do and what information they exchange. That is, semantic interoperability is not guaranteed between services just because both have REST interfaces.

REST, on top of HTTP, is meant to be self-descriptive and in the best case is a stateless protocol. Consider the following example, in REST, of a phone book service that allows someone to look up a person, given some unique identifier for that person:

```
http://www.XYZdirectory.com/phonebook/UserInfo/99999
```

The same simple lookup, implemented in SOAP, would be specified as something like the following:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap=http://www.w3.org/2001/
    12/soap-envelope
 soap:encodingStyle="http://www.w3.org/2001/12/
    soap-encoding">
  <soap:Body pb="http://www.XYZdirectory.com/
    phonebook">
```

```
      <pb:GetUserInfo>
        <pb:UserIdentifier>99999</pb:UserIdentifier>
      </pb:GetUserInfo>
    </soap:Body>
  </soap:Envelope>
```

One aspect of the choice between SOAP and REST is whether you want to accept the complexity and restrictions of SOAP+WSDL (the Web Services Description Language) to get more standardized interoperability or if you want to avoid the overhead by using REST, but perhaps benefit from less standardization. What are the other considerations?

A message exchange in REST has somewhat fewer characters than a message exchange in SOAP. So one of the tradeoffs in the choice between REST and SOAP is the size of the individual messages. For systems exchanging a large number of messages, another tradeoff is between performance (favoring REST) and structured messages (favoring SOAP).

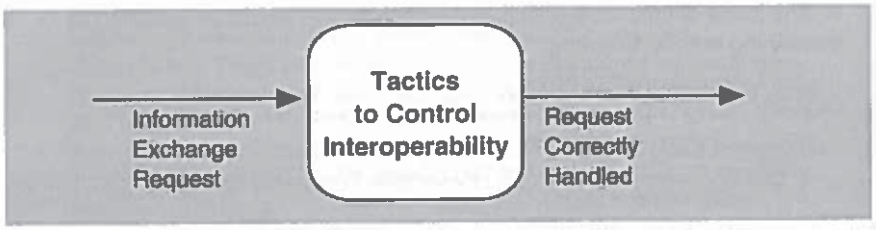
The decision to implement WS* or REST will depend on aspects such as the quality of service (QoS) required—WS* implementation has greater support for security, availability, and so on—and type of functionality. A RESTful implementation, because of its simplicity, is more appropriate for read-only functionality, typical of mashups, where there are minimal QoS requirements and concerns.

OK, so if you are building a service-based system, how do you choose? The truth is, you don't have to make a single choice, once and for all time; each technology is reasonably easy to use, at least for simple applications. And each has its strengths and weaknesses. Like everything else in architecture, it's all about the tradeoffs; your decision will likely hinge on the way those tradeoffs affect your system in your context.

—RK

6.2 Tactics for Interoperability

Figure 6.2 shows the goal of the set of interoperability tactics.



We identify two categories of interoperability tactics: locate and manage interfaces.

Locate

There is only one tactic in this category: *discover service*. It is used when the systems that interoperate must be discovered at runtime.

- *Discover service*. Locate a service through searching a known directory service. (By "service," we simply mean a set of capabilities that is accessible via some kind of interface.) There may be multiple levels of indirection in this location process—that is, a known location points to another location that in turn can be searched for the service. The service can be located by type of service, by name, by location, or by some other attribute.

Manage Interfaces

Managing interfaces consists of two tactics: *orchestrate* and *tailor interface*.

- *Orchestrate*. Orchestrate is a tactic that uses a control mechanism to coordinate and manage and sequence the invocation of particular services (which could be ignorant of each other). Orchestration is used when the interoperating systems must interact in a complex fashion to accomplish a complex task; orchestration "scripts" the interaction. Workflow engines are an example of the use of the orchestrate tactic. The mediator design pattern can serve this function for simple orchestration. Complex orchestration can be specified in a language such as BPEL.
- *Tailor interface*. Tailor interface is a tactic that adds or removes capabilities to an interface. Capabilities such as translation, adding buffering, or smoothing data can be added. Capabilities may be removed as well. An example of removing capabilities is to hide particular functions from untrusted users. The decorator pattern is an example of the tailor interface tactic.

The enterprise service bus that underlies many service-oriented architectures combines both of the manage interface tactics.

Figure 6.3 shows a summary of the tactics to achieve interoperability.