6. Interoperability

With Liming Zhu

The early bird (A) arrives and catches worm (B), pulling string (C) and shooting off pistol (D). Bullet (E) bursts balloon (F), dropping brick (G) on bulb (H) of atomizer (I) and shooting perfume (J) on sponge (K). As sponge gains in weight, it lowers itself and pulls string (L), raising end of board (M). Cannon ball (N) drops on nose of sleeping gentleman. String tied to cannon ball releases cork (O) of vacuum bottle (P) and ice water falls on sleeper’s face to assist the cannon ball in its good work.

—Rube Goldberg, instructions for “a simple alarm clock”

Interoperability is about the degree to which two or more systems can usefully exchange meaningful information via interfaces in a particular context. The definition includes not only having the ability to exchange data (syntactic interoperability) but also having the ability to correctly interpret the data being exchanged (semantic interoperability). A system cannot be interoperable in isolation. Any discussion of a system’s interoperability needs to identify with whom, with what, and under what circumstances—hence, the need to include the context.

Interoperability is affected by the systems expected to interoperate. If we already know the interfaces of external systems with which our system will interoperate, then we can design that knowledge into the system. Or we can design our system to interoperate in a more generic fashion, so that the identity and the services that another system provides can be bound later in the life cycle, at build time or runtime.

Like all quality attributes, interoperability is not a yes-or-no proposition but has shades of meaning. There are several characterizing frameworks for
interoperability, all of which seem to define five levels of interoperability “maturity” (see the “For Further Reading” section at the end of this chapter for a pointer). The lowest level signifies systems that do not share data at all, or do not do so with any success. The highest level signifies systems that work together seamlessly, never make any mistakes interpreting each other’s communications, and share the same underlying semantic model of the world in which they work.

“Exchanging Information via Interfaces”

Interoperability, as we said, is about two or more systems exchanging information via interfaces.

At this point, we need to clarify two critical concepts central to this discussion and emphasize that we are taking a broad view of each.

The first is what it means to “exchange information.” This can mean something as simple as program A calling program B with some parameters. However, two systems (or parts of a system) can exchange information even if they never communicate directly with each other. Did you ever have a conversation like the following in junior high school? “Charlene said that Kim told her that Trevor heard that Heather wants to come to your party.” Of course, junior high school protocol would preclude the possibility of responding directly to Heather. Instead, your response (if you like Heather) might be, “Cool,” which would make its way back through Charlene, Kim, and Trevor. You and Heather exchanged information, but never talked to each other. (We hope you got to talk to each other at the party.)

Entities can exchange information in even less direct ways. If I have an idea of a program’s behavior, and I design my program to work assuming that behavior, the two programs have also exchanged information—just not at runtime.

One of the more infamous software disasters in history occurred when an antimissile system failed to intercept an incoming ballistic rocket in Operation Desert Storm in 1991, resulting in 28 fatalities. One of the missile’s software components “expected” to be shut down and restarted periodically, so it could recalibrate its orientation framework from a known initial point. The software had been running for some 100 hours when the missile was launched, and calculation errors had accumulated to the point where the software component’s idea of its orientation had wandered hopelessly away from truth.
Systems (or components within systems) often have or embody expectations about the behaviors of its “information exchange” partners. The assumption of everything interacting with the errant component in the preceding example was that its accuracy did not degrade over time. The result was a system of parts that did not work together correctly to solve the problem they were supposed to.

The second concept we need to stress is what we mean by “interface.” Once again, we mean something beyond the simple case—a syntactic description of a component’s programs and the type and number of their parameters, most commonly realized as an API. That’s necessary for interoperability—heck, it’s necessary if you want your software to compile successfully—but it’s not sufficient. To illustrate this concept, we’ll use another “conversation” analogy. Has your partner or spouse ever come home, slammed the door, and when you ask what’s wrong, replied “Nothing!”? If so, then you should be able to appreciate the keen difference between syntax and semantics and the role of expectations in understanding how an entity behaves. Because we want interoperable systems and components, and not simply ones that compile together nicely, we require a higher bar for interfaces than just a statement of syntax. By “interface,” we mean the set of assumptions that you can safely make about an entity. For example, it’s a safe assumption that whatever’s wrong with your spouse/partner, it’s not “Nothing,” and you know that because that “interface” extends way beyond just the words they say. And it’s also a safe assumption that nothing about our missile component’s accuracy degradation over time was in its API, and yet that was a critical part of its interface.

—PCC

Here are some of the reasons you might want systems to interoperate:

• Your system provides a service to be used by a collection of unknown systems. These systems need to interoperate with your system even though you may know nothing about them. An example is a service such as Google Maps.

• You are constructing capabilities from existing systems. For example, one of the existing systems is responsible for sensing its environment, another one is responsible for processing the raw data, a third is responsible for interpreting the data, and a final one is responsible for producing and
distributing a representation of what was sensed. An example is a traffic sensing system where the input comes from individual vehicles, the raw data is processed into common units of measurement, is interpreted and fused, and traffic congestion information is broadcast.

These examples highlight two important aspects of interoperability:

1. **Discovery.** The consumer of a service must discover (possibly at runtime, possibly prior to runtime) the location, identity, and the interface of the service.

2. **Handling of the response.** There are three distinct possibilities:
   - The service reports back to the requester with the response.
   - The service sends its response on to another system.
   - The service broadcasts its response to any interested parties.

These elements, discovery and disposition of response, along with management of interfaces, govern our discussion of scenarios and tactics for interoperability.

---

**Systems of Systems**

If you have a group of systems that are interoperating to achieve a joint purpose, you have what is called a **system of systems** (SoS). An SoS is an arrangement of systems that results when independent and useful systems are integrated into a larger system that delivers unique capabilities. Table 6.1 shows a categorization of SoSs.

**Table 6.1. Taxonomy of Systems of Systems**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directed</td>
<td>SoS objectives, centralized management, funding, and authority for the overall SoS are in place. Systems are subordinated to the SoS.</td>
</tr>
<tr>
<td>Acknowledged</td>
<td>SoS objectives, centralized management, funding, and authority in place. However, systems retain their own management, funding, and authority in parallel with the SoS.</td>
</tr>
<tr>
<td>Collaborative</td>
<td>There are no overall objectives, centralized management, authority, responsibility, or funding at the SoS level. Systems voluntarily work together to address shared or common interests.</td>
</tr>
<tr>
<td>Virtual</td>
<td>Like collaborative, but systems don't know about each other.</td>
</tr>
</tbody>
</table>

* The taxonomy shown is an extension of work done by Mark Maier in 1998.

In directed and acknowledged SoSs, there is a deliberate attempt to create
an SoS. The key difference is that in the former, there is SoS-level management that exercises control over the constituent systems, while in the latter, the constituent systems retain a high degree of autonomy in their own evolution. Collaborative and virtual systems of systems are more ad hoc, absent an overarching authority or source of funding and, in the case of a virtual SoS, even absent the knowledge about the scope and membership of the SoS.

The collaborative case is quite common. Consider the Google Maps example from the introduction. Google is the manager and funding authority for the map service. Each use of the maps in an application (an SoS) has its own management and funding authority, and there is no overall management of all of the applications that use Google Maps. The various organizations involved in the applications collaborate (either explicitly or implicitly) to enable the applications to work correctly.

A virtual SoS involves large systems and is much more ad hoc. For example, there are over 3,000 electric companies in the U.S. electric grid, each state has a public utility commission that oversees the utility companies operating in its state, and the federal Department of Energy provides some level of policy guidance. Many of the systems within the electric grid must interoperate, but there is no management authority for the overall system.

### 6.1. Interoperability General Scenario

The following are the portions of an interoperability general scenario:

- **Source of stimulus.** A system that initiates a request.
- **Stimulus.** A request to exchange information among systems.
- **Artifacts.** The systems that wish to interoperate.
- **Environment.** The systems that wish to interoperate are discovered at runtime or are known prior to runtime.
- **Response.** The request to interoperate results in the exchange of information. The information is understood by the receiving party both syntactically and semantically. Alternatively, the request is rejected and appropriate entities are notified. In either case, the request may be logged.
- **Response measure.** The percentage of information exchanges correctly processed or the percentage of information exchanges correctly rejected.
Figure 6.1 gives an example: Our vehicle information system sends our current location to the traffic monitoring system. The traffic monitoring system combines our location with other information, overlays this information on a Google Map, and broadcasts it. Our location information is correctly included with a probability of 99.9%.

Figure 6.1. Sample concrete interoperability scenario

Table 6.2 presents the possible values for each portion of an interoperability scenario.

Table 6.2. General Interoperability Scenario
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:

**Click here to view code image**
http://www.XYZdirectory.com/phonebook/UserInfo/99999

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

**Click here to view code image**
```xml
<?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.

![Figure 6.2. Goal 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.

![Interoperability Tactics Diagram](image)

**Figure 6.3. Summary of interoperability tactics**

---

**Why Standards Are Not Enough to Guarantee Interoperability**

*By Grace Lewis*

Developer of System A needs to exchange product data with System B. Developer A finds that there is an existing WS* web service interface for sending product data that among other fields contains price expressed in XML Schema as a decimal with two fraction digits. Developer A writes code to interact with the web service and the system works perfectly. However, after two weeks of operation, there is a huge discrepancy between the totals reported by System A and the totals reported by System B. After
conversations between the two developers, they discover that System B expected to receive a price that included tax and System A was sending it without tax.

This is a simple example of why standards are not enough. The systems exchanged data perfectly because they both agreed that the price was a decimal with two fractions digits expressed in XML Schema and the message was sent via SOAP over HTTP (syntax)—standards used in the implementation of WS* web services—but they did not agree on whether the price included tax or not (semantics).

Of course, the only realistic approach to getting diverse applications to share information is by reaching agreements on the structure and function of the information to be shared. These agreements are often reflected in standards that provide a common interface that multiple vendors and application builders support. Standards have indeed been instrumental in achieving a significant level of interoperability that we rely on in almost every domain. However, while standards are useful and in many ways indispensable, expectations of what can be achieved through standards are unrealistic. Here are some of the challenges that organizations face related to standards and interoperability:

1. Ideally, every implementation of a standard should be identical and thus completely interoperable with any other implementation. However, this is far from reality. Standards, when incorporated into products, tools, and services, undergo customizations and extensions because every vendor wants to create a unique selling point as a competitive advantage.

2. Standards are often deliberately open-ended and provide extension points. The actual implementation of these extension points is left to the discretion of implementers, leading to proprietary implementations.

3. Standards, like any technology, have a life cycle of their own and evolve over time in compatible and noncompatible ways. Deciding when to adopt a new or revised standard is a critical decision for organizations. Committing to a new standard that is not ready or eventually not adopted by the community is a big risk for organizations. On the other hand, waiting too long may also become a problem, which can lead to unsupported products, incompatibilities, and workarounds, because everyone else is using the standard.

4. Within the software community, there are as many bad standards as
there are engineers with opinions. Bad standards include underspecified, overspecified, inconsistently specified, unstable, or irrelevant standards.

5. It is quite common for standards to be championed by competing organizations, resulting in conflicting standards due to overlap or mutual exclusion.

6. For new and rapidly emerging domains, the argument often made is that standardization will be destructive because it will hinder flexibility: premature standardization will force the use of an inadequate approach and lead to abandoning other presumably better approaches. So what do organizations do in the meantime?

What these challenges illustrate is that because of the way in which standards are usually created and evolved, we cannot let standards drive our architectures. We need to architect systems first and then decide which standards can support desired system requirements and qualities. This approach allows standards to change and evolve without affecting the overall architecture of the system.

I once heard someone in a keynote address say that “The nice thing about standards is that there are so many to choose from.”

### 6.3. A Design Checklist for Interoperability

Table 6.3 is a checklist to support the design and analysis process for interoperability.

<p>| Table 6.3. Checklist to Support the Design and Analysis Process for Interoperability |</p>
<table>
<thead>
<tr>
<th>Category</th>
<th>Checklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation of</td>
<td>Determine which of your system responsibilities will need to interoperate with other systems.</td>
</tr>
<tr>
<td>Responsibilities</td>
<td>Ensure that responsibilities have been allocated to detect a request to interoperate with known or unknown external systems.</td>
</tr>
<tr>
<td></td>
<td>Ensure that responsibilities have been allocated to carry out the following tasks:</td>
</tr>
<tr>
<td></td>
<td>▪ Accept the request</td>
</tr>
<tr>
<td></td>
<td>▪ Exchange information</td>
</tr>
<tr>
<td></td>
<td>▪ Reject the request</td>
</tr>
<tr>
<td></td>
<td>▪ Notify appropriate entities (people or systems)</td>
</tr>
<tr>
<td></td>
<td>▪ Log the request (for interoperability in an untrusted environment, logging for nonrepudiation is essential)</td>
</tr>
<tr>
<td>Coordination Model</td>
<td>Ensure that the coordination mechanisms can meet the critical quality attribute requirements. Considerations for performance include the following:</td>
</tr>
<tr>
<td></td>
<td>▪ Volume of traffic on the network both created by the systems under your control and generated by systems not under your control</td>
</tr>
<tr>
<td></td>
<td>▪ Timeliness of the messages being sent by your systems</td>
</tr>
<tr>
<td></td>
<td>▪ Currency of the messages being sent by your systems</td>
</tr>
<tr>
<td></td>
<td>▪ Jitter of the messages’ arrival times</td>
</tr>
<tr>
<td></td>
<td>▪ Ensure that all of the systems under your control make assumptions about protocols and underlying networks that are consistent with the systems not under your control.</td>
</tr>
</tbody>
</table>
Data Model

Determine the syntax and semantics of the major data abstractions that may be exchanged among interoperating systems.

Ensure that these major data abstractions are consistent with data from the interoperating systems. (If your system’s data model is confidential and must not be made public, you may have to apply transformations to and from the data abstractions of systems with which yours interoperates.)

Mapping among Architectural Elements

For interoperability, the critical mapping is that of components to processors. Beyond the necessity of making sure that components that communicate externally are hosted on processors that can reach the network, the primary considerations deal with meeting the security, availability, and performance requirements for the communication. These will be dealt with in their respective chapters.

Resource Management

Ensure that interoperation with another system (accepting a request and/or rejecting a request) can never exhaust critical system resources (e.g., can a flood of such requests cause service to be denied to legitimate users?).

Ensure that the resource load imposed by the communication requirements of interoperation is acceptable.

Ensure that if interoperation requires that resources be shared among the participating systems, an adequate arbitration policy is in place.

Binding Time

Determine the systems that may interoperate, and when they become known to each other. For each system over which you have control:

- Ensure that it has a policy for dealing with binding to both known and unknown external systems.
- Ensure that it has mechanisms in place to reject unaccept-able bindings and to log such requests.
- In the case of late binding, ensure that mechanisms will support the discovery of relevant new services or protocols, or the sending of information using chosen protocols.

Choice of Technology

For any of your chosen technologies, are they “visible” at the interface boundary of a system? If so, what interoperability effects do they have? Do they support, undercut, or have no effect on the interoperability scenarios that apply to your system? Ensure the effects they have are acceptable.

Consider technologies that are designed to support interoperability, such as web services. Can they be used to satisfy the interoperability requirements for the systems under your control?

6.4. Summary

Interoperability refers to the ability of systems to usefully exchange information.
These systems may have been constructed with the intention of exchanging information, they may be existing systems that are desired to exchange information, or they may provide general services without knowing the details of the systems that wish to utilize those services.

The general scenario for interoperability provides the details of these different cases. In any interoperability case, the goal is to intentionally exchange information or reject the request to exchange information.

Achieving interoperability involves the relevant systems locating each other and then managing the interfaces so that they can exchange information.

6.5. For Further Reading

An SEI report gives a good overview of interoperability, and it highlights some of the “maturity frameworks” for interoperability [Brownsword 04].

The various WS* services are being developed under the auspices of the World Wide Web Consortium (W3C) and can be found at www.w3.org/2002/ws.

Systems of systems are of particular interest to the U.S. Department of Defense. An engineering guide can be found at [ODUSD 08].

6.6. Discussion Questions

1. Find a web service mashup. Write several concrete interoperability scenarios for this system.

2. What is the relationship between interoperability and the other quality attributes highlighted in this book? For example, if two systems fail to exchange information properly, could a security flaw result? What other quality attributes seem strongly related (at least potentially) to interoperability?

3. Is a service-oriented system a system of systems? If so, describe a service-oriented system that is directed, one that is acknowledged, one that is collaborative, and one that is virtual.

4. Universal Description, Discovery, and Integration (UDDI) was touted as a discovery service, but commercial support for UDDI is being withdrawn. Why do you suppose this is? Does it have anything to do with the quality attributes delivered or not delivered by UDDI solutions?

5. Why has the importance of orchestration grown in recent years?

6. If you are a technology producer, what are the advantages and disadvantages of adhering to interoperability standards? Why would a
producer not adhere to a standard?

7. With what other systems will an automatic teller machine need to interoperate? How would you change your automatic teller system design to accommodate these other systems?