



CC Intelligent Solutions, Inc.  
7701 Six Forks Road  
Suite 105  
Raleigh, North Carolina 27615

Phone (919) 844-2111  
Fax (919) 844-2051

---

# Technical White Paper

---

## Services-Oriented Architectures

---

## 1 Motivation

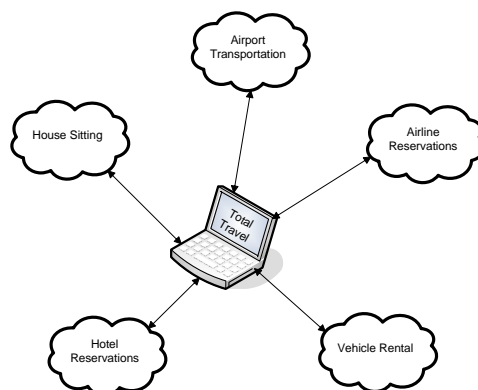
There is a long-standing problem in computer science surrounding the development of applications that interoperate with other applications, or n-tier applications where the tiers are not co-located. Solutions in this space often involve tight couplings between the communicating entities, are difficult to scale and not sufficiently generic to allow new entities to be included. Services-Oriented Architecture (SOA) addresses this problem by abstracting the interoperating entities into *services* that communicate by some form of messaging, more and more typically by *Web Services*.

## 2 Problem Description

The easiest way to illustrate the problem areas addressed by SOA and to provide a basis of understanding for addressing those problems is by example. Consider a comprehensive travel-booking system (Total Travel) that provides the following capabilities, all of which are optional, to would-be travelers.

- House-sitting
- Airport transportation
- Airline travel
- Hotel reservations
- Vehicle rental

It would be relatively straightforward to implement this application if all of the services were provided by the same company, and under a common configuration control. This design is shown below in Figure 1.

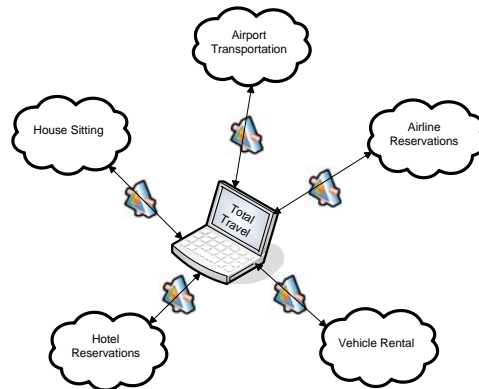


• Figure 1 - Total Travel System With Direct Links

In this case, the application modules can be written to allow easy communication and interoperation. The real problem surfaces in a system where *none*, or only some subset, of the capabilities is provided by one company, but instead relies upon third-party systems (and not always the same ones) to take care of particular features. In this case, it is exceedingly unlikely that all, or indeed any, of the service-providers will choose to open their systems to direct access by an application not developed in their shop. This problem is exacerbated by geographical uniqueness (e.g. there is probably not a single house-sitting service that covers all possible locations).

### 3 Addressing the Problem

Fortunately, the abstraction lessons learned in Object-Oriented Architecture extend into this space. If each of the features provided by the Total Travel system is treated as an encapsulated element (similar to an object) where implementation detail is hidden, but where there is a public interface for interaction, we can achieve similar benefits as we did in the OO system. In this case, the encapsulated elements are called *services*, and the public interface is a collection of *Web Services*. This system is depicted below in Figure 2.

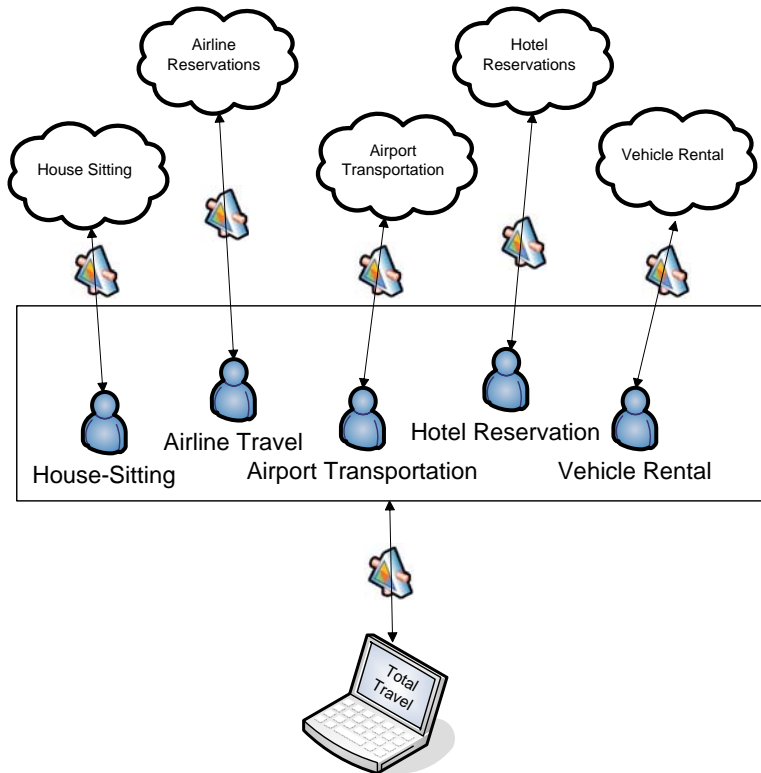


• Figure 2 - Total Travel System With Web Services

In this model, so long as the Web Service interface description is constant, the Total Travel system and the individual services can be modified and updated independently.

This model takes us a long way toward addressing our underlying problem; however, there are still several open issues. First, the Total Travel system should allow users to select an airline or a hotel chain, in the event that they have rewards programs or other preferences. The system above only allows for one service to fill each need, which precludes this user story.

To address this issue, each of the Web Services in the above system should be replaced with a *Service Broker*. This Service Broker would be called by the Total Travel system, and would, based upon user preferences, select the appropriate service to fit each need the user has specified. The Service Broker could also be extended to include *agents*, which are software entities responsible for taking actions on a caller's behalf. A likely use of such agents would be to look for low-cost providers by calling the possible services and comparing results. Then, the Service Broker can return the results of a call that matches the user's needs, while considering preferences and ordering by price.



• Figure 3 - Total Travel System With Service Broker and Agents

The system shown in Figure 3 addresses almost all the requirements of a comprehensive travel system. The missing piece is transactional integrity. If the Service Broker returns a result set, and by the time the user books that set, it is no longer available, or the price has changed, then the entire set of reservations needs to be rolled back to prevent partial bookings. The introduction of a Distributed Transaction Coordinator (DTC) can solve this last issue. In this scenario, the user selects and books their itinerary and the DTC ensures that if any transaction fails, the transactions that were committed are rolled back or compensated. This DTC would exist inside the Service Broker or between the Service Broker and the agents if all pieces are separate.

## 4 Application of SOAs and Conclusion

There is an abundance of applications where architectures like the one outlined above not only facilitate development, but result in systems that are more extensible, easier to maintain, deploy and upgrade and, as a result, have a lower lifecycle cost. These applications typically fall into one of two categories:

- 1) Applications designed to interoperate with external or third-party systems
- 2) N-tier applications where the tiers may be logically or spatially distinct

We have seen one case of an application designed to interoperate with external or third-party systems in the Total Travel example outlined in the preceding sections. Many other examples exist, but they are similar in nature to Total Travel. The other case is more interesting.

Consider the presentation and business layers. The *presentation layer* of an n-tier application is responsible for rendering the application to the user. Typically it is graphical in nature, accepts user input and performs actions based upon that input. Often, business rules get incorporated into the presentation layer during software construction. The intermingling of business logic into the presentation layer makes the presentation, or business logic, code more difficult to isolate and manage.

The standard solution for this tight coupling issue is to divide the presentation layer from a separate *business layer* which is responsible for all of the business rules. Once this is accomplished, the coupling between presentation and business logic is broken, making maintenance and modification much easier.

Traditionally, these two layers have interacted directly, or by remote procedure calls (remoting in Microsoft's .NET platform). More and more, in modern applications, these two layers are interacting via Web Services. This allows the presentation and business layers to be separated not only logically, but also physically, thus distributing the computational workload, isolating business changes, and providing for server redundancy on the business layer.

This form of interaction by services is what makes SOA such a strong architectural pattern in the software development world. It allows applications to be more easily extensible, gives them the ability to easily distribute workload by spatial separation, making them more scalable, and breaks the tight coupling between applications or application tiers making them easier to deploy independently.