Extracted from:

Interface-Oriented Design

Chapter 6

Remote Interfaces

Systems today are moving away from self-contained programs; they tend to interact with a number of other programs on remote hosts. Dealing with remote interfaces involves several more issues than does dealing with local interfaces, so we'll explore these facets now.

Many remote interfaces use a document style, rather than a procedural style. Document-style interfaces have a different paradigm from procedural-style interfaces and are less familiar to programmers, so we will investigate documents in some detail. We'll also examine how many of the concepts we discussed before are applicable to remote interfaces, such as statefulness versus statelessness.

6.1 Introduction

If you are physically in the pizza parlor, you can see the order taker. You are aware whether he is writing down your order or discussing last night's ball game with the cook. If you are local, you don't have to worry about failure to connect.

The pizza interface we introduced in the first chapter is really a remote interface: you make a connection over the phone network. Dealing with a remote interface is different from a local interface. A whole host of problems can occur that you might need to handle.

What if the phone is busy? Do you try the dialing again, or do you try another pizza parlor? Is the busy phone because of a failure in the phone company or a failure in the pizza parlor's phone?

What if it rings but no one answers? Do you try again, thinking you may have dialed the wrong number? Do you assume that they aren't open?

Suppose you get cut off in the middle of the call. Do you call back?

External Interfaces

The problems of pizza ordering exist in any external interface. An external interface is one called by other processes (either local or remote). External interfaces differ from local interfaces by network considerations, by nonhomogeneity of hosts, and by multiple process interactions.[1]

If an entire software system is contained within a single process, the system fails if the process fails. With a system consisting of multiple processes, a calling process (e.g., a client) has to handle the unavailability of other processes (e.g., a server). The client usually continues to run in spite of the failure of servers, but it needs either to communicate the server failure to the user or to act on that failure in a transparent manner, as per the Third Law of Interfaces.

Remote interfaces are external interfaces that are accessed over a network. In addition to server failure, with a network you may have a network delay or a network failure. Note that if you are unable to connect to a server, it is difficult to determine whether the network is down or whether the server is down. Likewise, a delay may be due to an overloaded network or an overloaded server that is handling too many clients. In either case, the client needs to handle the failure.

With nonhomogeneity, the client and the server may be different processor types (e.g., IBM mainframe versus PC). Even on a local machine, where processor types are not a consideration, the caller and server may be written in different programming languages.

Network Disruption

What would you do if you were ordering a pizza by phone and the call dropped before you heard how long it was going to take? You'd call back. You'd try to continue to describe the pizza you were ordering. But

[1]A local interface is usually called by only one process, although it may be called by multiple threads within that process. A remote interface can typically be concurrently called by multiple remote processes.

the pizza shop says, "I've had hundreds of orders in the last minute for just that type of pizza." You don't really want to order a second pizza. And the store owner doesn't want to make an unclaimed pizza. How would we change the pizza-ordering interface to avoid this?

The store owner could take some identification at the beginning of a call—a name or a phone number. If the circuit is broken, you call back and give the same name or phone number. If the order taker determines the name corresponds to one of the uncompleted orders, he pulls it off the shelf and resumes at the appropriate place in the order sequence.[2]

Getting initial identification is a form of planning for the possibility of communication disruption. The interface protocol should assume that the network may go down. In a manner similar to the pizza shop, interfaces can use client-generated IDs to ensure that service invocations are not duplicated. For example, when credit card transactions are submitted to a bank, the merchant identifies each transaction with a unique ID. If the connection is broken in the middle of transmitting a transaction, the merchant resubmits transactions that have not been explicitly confirmed. The bank knows by the ID for the particular merchant whether a transaction has already been posted. If the transaction has been posted, the bank merely confirms the transaction without reposting it.

6.2 Procedural and Document Interfaces

In our example in Chapter 1, you called the pizza shop over the phone. Your pizza shop may also accept fax orders. What is different about making a phone call versus sending a fax order? In either case, the order needs to be put into terms both you and the pizza shop understand. With the voice system, you execute a series of operations to create an order. With the fax, you have an order form that defines the required and optional data.

Problems are discovered immediately in the voice system. For example, you can ask for anchovies and get an immediate response. The voice on the other end can say "nope," meaning either they never have anchovies

[2]Some readers might note that a name such as Jim might be given for different orders. If the given name matches a previous name, the order taker may inform you that you have to give a different name. A phone number is not only good for identification but also for verification. The store owner can check the caller ID against the phone number to see whether it's the same one you said.

(a permanent error) or they don't have any today (a temporary error). In either case, you can make up your mind whether you want to have an anchovyless pizza or find some other pizza place.

With the a fax-based system, you fill out an order and await a response. The response may be a fax with the time till delivery or a fax saying, "Don't have any." If the latter, you need to alter your order and resubmit it. You may wonder whether your order was received. Since you may have to wait a while to get a fax back, it is harder to determine when to resend the order. The pizza parlor's fax may be out of paper. The scanner for the return fax may not be working. The order may have been put onto a pile. Only when the order is retrieved from the pile is a fax returned. We shall see how these issues of ordering by fax have parallels in remote interfaces.

External interfaces can use either procedural style or document style. A procedural interface looks like the interfaces we've been describing in this book. On the other hand, document-style interfaces use sets of data messages, similar to the fax-based pizza order.

For the most flexibility, the client (interface user) and the server (interface implementation provider) should be loosely coupled in terms of platform and language. A client written in any language should be able to access the server. You can accomplish this decoupling with either style.

Procedural Style

You can use Common Object Request Broker Architecture (CORBA) to define procedural-style interfaces that are both language and platform independent.[3] With CORBA, you specify the interface with the Interface Definition Language (IDL).[4] IDL looks a lot like a C++ header or a Java interface. A transformation program turns an IDL declaration into code stubs appropriate for a particular language and platform. An example of an interface defined in IDL is as follows:

```
enum Size {SMALL, MEDIUM, LARGE};
enum Toppings {PEPPERONI, MUSHROOMS, PEPPERS, SAUSAGE};
```

[3]You can define remote interfaces in a language-dependent manner, such as Java's Remote Method Invocation. You could also define them in a platform-dependent manner, such as Window's Distributed Component Object Model (DCOM).

[4]See http://www.omg.org for more information about CORBA and IDL.

```
interface PizzaOrdering
    {
    exception UnableToDeliver(string explanation);
    exception UnableToMake(string explanation);
    typedef Toppings ToppingArray[5];
    set_size(in Size the_size) raises (UnableToMake);
    set_toppings(ToppingArray toppings) raises (UnableToMake);
    set_address(in string street_address);
    TimePeriod get_time_till_delivered() raises (UnableToDeliver);
    }
```

Procedural-style remote interfaces look familiar to programmers. Calls to methods in remote interfaces (a Remote Procedure Call [RPC]) appear in your code as if they were calls to local interfaces. The only major difference is that the code must handle communication failure situations. RPCs are typically used for an immediate request/response in interactive situations. A client that called the PizzaOrdering interface can find out immediately whether the shop cannot make the pizza.

Procedural-style interfaces tend to be fine-grained. For example, they frequently contain operations for accessing individual values such as set_size() in the PizzaOrdering interface.

Document Style

With document style, the client and server interchange a series of data messages (documents). For a pizza order, the sequence might start with a document:

```
Document: PizzaOrder
    Size
    Toppings
    Address
```

The response could be either like this:

```
Document: SuccessResponse
    TimePeriod time_to_deliver
```

or like this:

```
Document: ErrorResponse
    String error_explanation
```

You may be less familiar with document-style interfaces. The documents represent a series of messages transmitted between the client and the service provider. The protocol is defined by the sequence of messages. We'll explore a typical sequence later in this chapter. Messages are not necessarily processed immediately. Response documents, such as SuccessResponse, may come almost immediately. However, they may also be delayed. A client using the document interface to order pizzas may not instantly find out whether the requested pizza can be made.

A document-style interface tends to be very coarse-grained. For example, a PizzaOrder document that contains the size and toppings is sent in a single operation, like this:[5]

```
interface Ordering
    submit_order(PizzaOrder)
```

PROCEDURAL STYLE

Advantage—remote and local interfaces can appear the same

Disadvantage—can require more communication (especially if fine-grained)

DOCUMENT STYLE

Advantage—can require less communication

Disadvantages—style is less familiar to programmers

6.3 Facets of External Interfaces

We discussed several facets of interfaces in Chapter 3. Now we'll examine some additional facets of external interfaces.

Synchronous versus Asynchronous

In Chapter 3, we described asynchronous event handling using the Observer pattern. Likewise, communication between a client and a server can be either synchronous or asynchronous. With synchronous interfaces, the client does not end communication until a result is returned.

With asynchronous interfaces, a result is returned at some other time after the client has ended the original communication. For example, documents are often placed on message queues. The client creates a

[5]The most general document interface consists of three operations:
Request/response—`Document send_document_and_get_response(Document)`
Request—`void send_document(Document)`
Response—`Document receive_document()`
That's so coarse-grained, you can transmit anything. (OK, maybe not anything, but almost anything).

document (message) and puts it onto a message queue. The client usually continues processing, handling other events. At some time, the server retrieves the message from the queue and processes the document. It then returns a response document, either directly to the client or back onto the queue for retrieval by the client.

Two typical combinations of modes for applications that use external interfaces are asynchronous/document (e.g., message queues) and synchronous/procedural (e.g., RPCs). You could consider the World Wide Web to be an synchronous/document interface: you send a document (e.g., a filled-in form) and receive a document (a new web page) in return. The least frequently used combination is asynchronous/procedural.

SYNCHRONOUS

Advantage—practically immediate response

Disadvantage—cannot scale up as well

ASYNCHRONOUS

Advantage—can scale well, especially with document queues

Disadvantage—documents should be validated on client before transmitting

Stateful versus Stateless

With remote interfaces, the distinction between stateful and stateless interfaces is more critical. A server that keeps state for clients may not be able to handle as many clients as a server that does not keep state.

For example, many web sites have shopping carts. In a stateful interface, the contents of the shopping cart are kept in a semipersistent storage on the server. Each new connection from a client (i.e., a browser) creates a new shopping cart that is assigned a SessionID. The SessionID is the key that identifies the data for a particular client on the server. The browser returns this SessionID with each request for another web page. The server uses the SessionID to retrieve current contents of the shopping cart.

In a stateless interface, the server does not keep any state. For example, with a Google search, the URL passes the search parameters every time. If Google keeps any information on a search, it is for performance reasons, rather than for interface reasons.