

The Flowchart Lie

Or: Why do we still use Request/Response to handle asynchronous business processes?

I continuously spend time with improving my clients' application architectures and designs, and there is one thing which strikes me quite often. It seems to be the common approach to map business processes to technical functions using general request/response style communication between client and servers. This is done even though most of the underlying business processes are inherently asynchronous and would therefore easily lend themselves a mapping to asynchronous messaging.

The Flowchart Lie

After observing this issue in more than a dozen different applications, I came to the conclusion about what the common root cause for these decisions must be. The flowchart.

Do you remember the first diagrams you have seen when you've been taught how to develop programs? I bet they were some sort of flowcharts which instantly showed that an application's function always start at the top, pass several conditions and loops, and finish at the bottom.

The good news is that flowcharts are right. The bad news is that they are talking about discrete algorithms, and not about business process mappings to software. When designing your applications, don't give in to the flowchart lie: Even though these processes have a distinct starting

point and ending point, they don't necessarily have to be completed synchronously!

The Typical Business Process

Let's look for example at some typical business processes of a mail order business.

1. A new customer wants to register and orders the catalog by postcard, fax, email, internet or phone
2. An existing customer places an order by postcard, fax, email, internet or phone
3. A customer returns items
4. A customer wants to know the status of an order

The first three processes are asynchronous in their very nature. Nevertheless most application designers and developers will instantly map them to a synchronous remote procedure call, and a synchronous INSERT statement into the database.

This means that the application's user has to wait until the complete request has been processed. Unfortunately the fact that numerous call centers are running as highly distributed operations results in them not always having Ethernet quality connections to their backend servers. In practice, this simply leads to the well-known *"I'm really sorry. Please give me a second. Our com-*

Ingo Rammer

<http://www.IngoRammer.com>

Ingo Rammer is independent consultant, mentor, and architect helping companies to adopt Microsoft .NET and Web service technologies.

He can help you to

- **Decrease the time** necessary for initial adoption of .NET and Web Services
- **Go for the best solution** right from the start instead of spending time with trying various approaches
- **Get a deeper understanding** of reliability, scalability and performance in the .NET Framework and how they apply to your application
- **Increase your own proficiency** in application architecture and design
- **Understand SOA** and how it applies to your current and future applications



puter system is very slow today" response on the phone. Something we as a customers definitely don't want to hear. And we are not yet talking about database downtime or similar issues which happen in the best 24/7 call centers¹.

Listing 1 Adding a Customer Request/Response Style

```
MyWebService svc = new MyWebServiceProxy();
svc.AddCustomer(...);
```

The problem with synchronous method calls can easily be explained in the two lines of code in Listing 1 which will most likely be used when implementing the first of the business processes presented above.

This means that the user's application will block until the request has been fully processed at the server. This is however largely unnecessary because the client doesn't need any kind of return value from this method call. Instead it will only add to taking up the user's time which should be spent with selling products to the prospective customer.

One of the first ways when trying to solve the problem is to call `.BeginAddCustomer()` instead of `.AddCustomer()` to invoke the web service asynchronously.

Unfortunately, this will not actually solve the problem but might in fact make matters worse. Asynchronous Web Service calls like this come without any guarantees of execution. If the user shuts down his client application before the call has been processed at the server, the update might or might not be completed at the server side. In addition, this *asynchronous* web service call is in fact just a synchronous call which will be executed in a new client-side thread to simulate asynchronous operation.

Instead of relying on this futile approach to asynchronicity, you should design and develop your application with message exchange patterns different from plain old request/response in mind. Instead of calling an HTTP based web service, you might just want to send the message using MSMQ (Microsoft Message Queue Server) which is easily accessible using the `System.Messaging` namespace in .NET. To allow for platform independent message exchanges, using IBM's MQSeries (or nowadays WebSphere MQ) will provide you with similar features.

This will allow the client to create a new XML based message and send it to the server without having to care about the quality of the

transporting network. It can instead fully trust the underlying transactional message queue. The client doesn't have to wait at all and you will also not tie up any client-side resources by creating additional client side threads.

On the server side, you will have to - again using `System.Messaging` - implement a queue listener which processes the incoming messages in a transactional way. It is even possible to create distributed transaction which guarantee that a message will only be removed from a queue if the corresponding INSERT/UPDATE/DELETE operations on your database/s were successful.

Using message queuing in general helps you to increase the reliability and scalability of your application. You can for example throttle the number of messages processed per second in peak times to better utilize your servers' resources without affecting their ability to respond to RPCs in a timely manner. After all, there will still be cases when you have to use request/response messages as in the fourth business process presented above.

Summary

Messaging will become more and more important in the future, especially as a preferred means of communicating between different SOAP services - some of which are maybe even running in environments outside your control. Designing and developing your applications today with these trends in mind will allow you to easily utilize future middleware products which will provide similar features based on standard SOAP and WS.* protocols.

¹ A recent client of mine who is one of the world's largest mobile communication providers reached the maximum level of 0.00% unscheduled downtime for their main backend database during one year. They still were offline for 32 hours in the same year due to scheduled maintenance. This is nearly the equivalent of a full work week's hours each year! And we are not yet counting several hours of connection failure for their remote call centers. Network failures happen—design for them!