A Matter of Definition

July 22nd, 2008

Steve Vinoski has been busy trying to convince the world at large that RPC is "fundamentally flawed". I think it is interesting to take a look at RPC and see what those fundamental flaws are (and whether there *are* flaws, for that matter). Doing this will definitely take more than one post, so don't expect the answers all at once. I will deal with various aspects of the topic over a number posts over the next few weeks, so please bear with me.

Before we can delve into the details, let's take a look at a definition of RPC. In one of the comments to his blog post, Steve states that RFC 707 defines RPC. Having just read trough that RFC again (twice), I cannot find a *definition* of RPC anywhere in that document. What I *can* find is an outline of a protocol that allows a client to ask a server to perform some work and to get results back from the server. But that's pretty much it. There is no mention of specific APIs, there is no mention of interface definition languages, and there is no requirement for specific interaction models. (RFC 707 explicitly states that interactions need not be synchronous and that the run-time environment can provide a non-blocking interaction model.)

The protocol that is described by RFC 707 is remarkably simple: just two message types, request and reply. This is attractive but, as we have found out in the intervening time, inadequate. For example, for a connection-oriented transport, we also need a message that confirms the acceptance of a connection; without such a message, the protocol can violate at-most-semantics, which are important for operations that are not idempotent. The importance of at-most-once semantics and idempotent operations is another thing that we have learned in the intervening time. (RFC 707 does not mention either.)

How much time? RFC 707 was published in January 1976, more than *thirty-two* years ago. To put things in perspective, that was at a time in computing history where RPC (and networking, for that matter) were truly in their infancy. People were taking their first few hesitant steps toward distributed computing back then: TCP/IP had only just been invented and there was no such thing as distributed computing as we understand it today—what counted as distributed computing back then were protocols such as telnet and ftp. (DCE did not exist, and even UUCP and email had not been invented yet.)

While RFC 707 defines one of the earliest protocols for RPC, we today know this protocol to be inadequate. Moreover, a lot has happened in the intervening years; RPC today is a far cry from "RPC" back in 1976, and RFC 707 is of interest mainly as a historical document that has essentially no relevance to modern middleware. So, let me see whether we can find a more current definition.

Wikipedia has an entry for RPC. That entry also refers to RFC 707, but states that RFC 707 "describes" RPC (which is not the same as defining it). The entry then offers the following explanation:

> *An RPC is initiated by the client sending a request message to a known remote server in order to execute a specified procedure using supplied*

> *parameters. A response is returned to the client where the application continues along with its process. […] While the server is processing the call, the client is blocked.*

"While the server is processing the call, the client is blocked." Huh? Only a few lines earlier, the same article references RFC 707, which explicitly states that the interaction can be non-blocking. I guess we are running into the limitations of Wikipedia—there is only so much quality control that can be applied to the various articles.

So, let's turn to whatis.com. It says:

> *Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. (A procedure call is also sometimes known as a function call or a subroutine call.) RPC uses the client/server model. The requesting program is a client and the service-providing program is the server. Like a regular or local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned.*

Hmm… The same error as with Wikipedia here: the entry claims that interactions are synchronous when, according to RFC 707 (and lots of past and current RPC implementations), they need not be. (One wonders whether one copied from the other.)

I did many more searches and checked a number of books, and what they turned up was very similar to the preceding "definitions" for RPC. The common theme is:

One program is active and issues a request for service.

Another program passively listens for requests and acts on them.

When the service is complete, the service-providing program can return results back to the originator of the request.

Now, this is nice as far as it goes, but it hardly is a *definition* of RPC. Instead, it is a description of basic principles. But that description is so loose, just about anything fits it, including Ice, DCE, CORBA, SOAP, and REST.

Yet, we all seem to somehow know what RPC is and is not:

- RPC is about procedure (or, for object-oriented RPC, method) calls. In other words, at the API level, the interaction feels like a procedure or method call (whether synchronous or asynchronous).
- RPC platforms require a contract that defines the procedures or methods, including the types of data that are exchanged as parameters and return values.

It seems to me that this gets a little closer to what RPC is about: we have APIs that mostly hide the grunt work involved in communicating over a network, and we have a formal contract that establishes a type system.

Typically, a compiler that creates stubs and skeletons from an interface definition language generates the API and enforces the contract; however, other ways to define the contract (such as reflection) can be used. Either way, the generated code or reflection takes care of marshalling chores.

Note that, for modern RPC, a static contract is only one way of doing things. For example, Ice and CORBA provide dynamic invocation and dispatch that allow you to use RPC without an interface definition and without generated code.

Of course, the preceding description is still a far cry from a proper definition. But at least it gets us in the right direction: RPC systems take care of networking chores and—at least much of the time—use generated code that gets data onto the wire and back off the wire again. (If anyone is aware of a more detailed and/or rigorous definition of RPC, please let me know; I'd be very interested to see it.)

To what extent various technologies, such as Ice, SOAP, and REST fit (or do not fit) that description is something I will explore in future postings.

Cheers,

Michi.