1.1 Nanoservices

There are many unsolved mysteries, you've probably heard about some of them like the Loch Ness monster, Bigfoot etc. However, the greatest mystery, or so I've heard, is getting the granularity of services right… Kidding aside, getting right-sized services is indeed one of the toughest tasks designing services – there's a lot to balance here e.g. the communications overhead, the flexibility of the system, reuse potential etc. I don't have the service granularity codex and deciding the best granularity depends on the specific context and decisions (e.g. the examples in the Knot anti-pattern above). It is an easier task to define what shouldn't be a service for instance, calling all of your existing ERP system a single service should definitely be shunned. The Nanoservices anti-pattern talks about the other extreme… the smaller services

Consider, for instance, the "calculator service" which appears in samples web-wide (I've personally seen examples in .NET, Java, PHP, C++ and a few more). A basic desk calculator, as we all know, supports several simple operations like add, subtract, multiply and divide and sometimes a few more. Implementing a calculator service isn't very complicated - Listing 10.1 below, for example, shows part of WSDL for a java calculator service that, lo and behold, accepts two numbers and adds them.

Listing 10.1 excerpt from a WSDL of a stateless calculator service example. The sample only includes the data needed for the "Add" operation. The add operation accepts two numbers and returns a result (http://cwiki.apache.org/GMOxDOC21/jaxws-calculator-simple-web-service-with-jax-ws.html)

```
  <wsdl:types>
        <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                    xmlns="http://jws.samples.geronimo.apache.org"
                    targetNamespace="http://jws.samples.geronimo.apache.org"
                    attributeFormDefault="unqualified"
elementFormDefault="qualified">

             <xsd:element name="add">
                <xsd:complexType>
                   <xsd:sequence>
                      <xsd:element name="value1" type="xsd:int"/>
                      <xsd:element name="value2" type="xsd:int"/>
                   </xsd:sequence>
                </xsd:complexType>
             </xsd:element>

             <xsd:element name="addResponse">
                <xsd:complexType>
                   <xsd:sequence>
                      <xsd:element name="return" type="xsd:int"/>
                   </xsd:sequence>
                </xsd:complexType>
             </xsd:element>
        </xsd:schema>
    </wsdl:types>

    <wsdl:message name="add">
       <wsdl:part name="add" element="tns:add"/>
    </wsdl:message>

    <wsdl:message name="addResponse">
       <wsdl:part name="addResponse" element="tns:addResponse"/>
    </wsdl:message>

    <wsdl:portType name="CalculatorPortType">
       <wsdl:operation name="add">
         <wsdl:input name="add" message="tns:add"/>
         <wsdl:output name="addResponse" message="tns:addResponse"/>
       </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="CalculatorSoapBinding" type="tns:CalculatorPortType">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>

        <wsdl:operation name="add">
            <soap:operation soapAction="add" style="document"/>
            <wsdl:input name="add">
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="addResponse">
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>

    </wsdl:binding>

    <wsdl:service name="Calculator">
        <wsdl:port name="CalculatorPort" binding="tns:CalculatorSoapBinding">
            <soap:address location="http://localhost:8080/jaxws-
calculator/calculator"/>
        </wsdl:port>
    </wsdl:service>
```

Calculator services can be even more advanced and have memory - consider listing 10.2 below, which shows an interface definition for a .NET (WCF) sample that uses workflow services and accepts a single value at a time

Listing 10.2 a Service contract definition for a statufil calculator service (http://msdn.microsoft.com/en-us/library/bb410782.aspx). The service accepts a single number at a time and remembers the former state from operation to operation.

```
[ServiceContract(Namespace = "http://Microsoft.WorkflowServices.Samples")]
public interface ICalculator
{
    [OperationContract()]
    int PowerOn();
    [OperationContract()]
    int Add(int value);
    [OperationContract()]
    int Subtract(int value);
    [OperationContract()]
    int Multiply(int value);
    [OperationContract()]
    int Divide(int value);
    [OperationContract()]
    void PowerOff();
}
```

The calculator service (both versions of it) is a very fine grained service. Naturally, or hopefully anyway, the calculator examples are just over simplified services used to demonstrate SOA related technologies (JAX-WS in the first excerpt and WCF and WF in the second one). The problem is when we see this level of granularity in real life services

1.1.1Consequences

Problem? Why is "fine granularity" a problem anyway? Isn't SOA all about breaking down monolith "silos" into small reusable services? More so, the finer grained a service is, the less context it carries. The less context a service carries the more reuse potential it has – and reuse is one of the holy grails of SOA isn't it? The calculator service above seems like the epitome of a reusable service. There's no doubt we can reuse it over and over and over.

Reuse is indeed a noble goal (I'll leave discussing how real it is for another occasion), the culprit of fine grained services, however, is the network. Services are consumed over networks – both local (LANs) and remote (extranets, WANs etc.). The result is that services are bound by the limitations and costs incurred by those network. Trying to disregard these costs is exactly what ailed most, if not all, RPC distributed system approaches that predated SOA (Corba, DCOM etc.) - The calculator service and other similarly sized services are nanoserivces.

> **Nonoservice is an Anti-pattern where a service is too fine grained. Nanoservice is a service whose overhead (communications, maintenance etc.) out-weights its utility.**

So how can nanoservices harm your SOA? Nanoservices cause many problems, the major ones being poor performance, fragmented logic and overhead. Let's look at them one by one

Every time we send a request to a service we incur a few costs such as serialization on caller, moving caller process to the OS network service, translation to the underlying network protocol, traveling on the network, moving from the OS network service to the called process, deserialization on the called process – and that's before adding security (encryption, firewalls etc), routing , retries etc. Modern networks and servers can make all this happen rather fast but if we have a lot of nano-services running around these numbers add-up to a significant performance nightmare,

Nano-services cause fragmented logic - almost by definition. As we break what should have been a meaningful cohesive service, into miniscule steps our logic is scattered between the bits that are needed to complete the business service. The fact that you need to haul over several services to accomplish something meaningful also spell increased chances of the Knot anti-pattern, mentioned above.

Proliferation of Nanoservices also causes development and management overhead. Just look at the amount of WSDL needed to define the calculator services in listing 10.1 above and for what? A service that adds a couple of numbers… There is a relatively fixed overhead associated with managing a service. This include things like keeping track of a service in a service registry, making sure it adheres to policy, writing the cruft (things we have to write around the business logic) for configuring it etc. Having nano-services around means we have to do this a whole-lot more times (i.e. per service) compared with having fewer coarser grained services.

The point of overhead out-weighing utility that appears in the Nano-services definition above is subtle but important. The fact that a contract does not have a lot of operations means we want to make sure we don't have a nano-service, but it doesn't automatically mean that it is. For instance, a fraud detection service contract might only accept transaction details and decide whether to authorize the transaction, deny it or move to further investigation. However the innards of this service involve a complex process like running the details in a rule engine checking for fraudulent behavior patterns, matching to black lists etc. In fact Fraud detection is such a complicated issue that these are actually systems and a SOA based one would be comprised of several services in itself.

The other side of the equation is also true a comprehensive contract does not guarantee a service is not a nano-service. For instance, in a system I designed on the initial iterations we developed a resource management service. It supported some very nice operations like getting status of all the services in the system, running sagas and of course allocating services. Allocating services meant that whenever an event went out that needed a (new) service instance to handle it, we had to make a call to the resource manager to get one. This provides for a neat centralized management and also for a performance bottleneck that slows the whole system. To solve this we went with distributed resource management but that't beyond the scope of this discussion. The point, however that is that the utility of the resource management (e.g. easy management of running sagas) vs. overhead associated with the service (the number of calls and performance hit on the system) was not worth it – Hench a nano-service.

1.1.2Causes

From a more technical point of view, we get to nanoservices from not paying attention to at least a couple of the fallacies of distributed computing. Mentioned in chapter 1, the fallacies of distributed computing are a few false assumptions that are easy to make and prove to be wrong and costly down the road. Specifically, we are talking here about assuming that

- Bandwidth is infinite – Even though bandwidth gets better and better, it is still not infinite within a specific setup. For instance in one project we were sending images over the wire and distribute them to computational services (a la map/reduce – see also Gridable Service in chapter 3). Things were working ok when we sent small images, but when we sent larger images we understood we were sending them as bitmaps and not as much more compact jpegs which caused a burden on the backbone of our switches which wasn't ready for that load.
- Transport cost is zero – As explained in the previous section every over-the-wire call incurs a lot of costs vs. a local call (also see figure 10.5 below).The costs of the transport can be considered both from the time it take to make each of these calls but even the real dollar value attached to making sure you have enough bandwidth (connection/routers, firewalls) to handle the traffic incurred

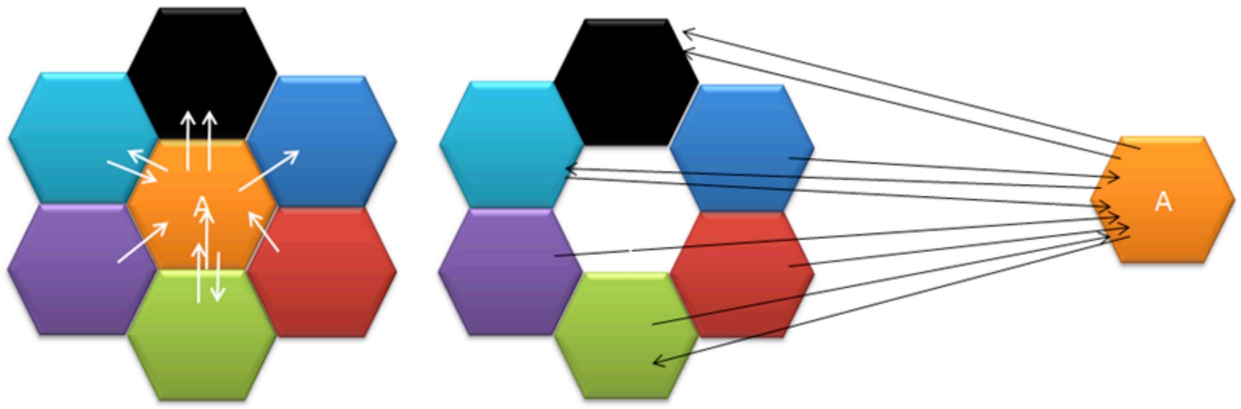

Figure 10.5 Local objects can "afford" to have intricate interactions with their surroundings. A similar functionality delivered over a network is more likely than not to cause poor performance because of the network related overhead.

Another reason to get Nano-Services, at least for beginners are poor examples – as, noted the calculator services above are taken from real examples provided by various vendors. SOA newcomers and/or people without a lot of distributed systems development experience can be easily take these samples at face value, and go about implementing services with similar granularity. The fact that that web-service framework mostly map service calls to object method calls makes this even more tempting.

Nano-services is also an inherent risk when applying the orchestrated choreography pattern. Adding an orchestration engine, capable of controlling flow and external to services tempts us to think that we can use it to drive all flow as little as it may seem. Couple this with the fact that the smaller the services are the more "Reuseable" they are (less context) and, again, you may end up with a lot of nano-services on your hands.

Lastly, since the nano-services boundary is soft (remember utility vs. overhead weight) behaviors that can look promising at design time can prove to be nano-services moving along (like the resource manager example above). This can be an acceptable if your SOA is developed iteratively (see 10.2.4 exceptions below) but it still mean that we have to come up with ways to refactor nano-services.

1.1.3Refactoring

There are basically two main ways to solve the nano-services problem. One, which is relatively easy, is to group related nano-services into a larger service. The second option, which is more complicated, is to redistribute its functionality among other services. Let's take a look at them one by one.

On one project I was working on we needed to send out notifications to users and admins via SMS messages. Since the software component that did the actual SMS dissemination was a 3rd party app we've decided to create a simple service (not unlike OO adapter) that accepts requests for SMS and talks to the 3rd party software. A nano-service was born, it even got a nice little name Post Office Service (ok, ok the original name was Spam Server but I thought it would look bad in presentations ☺).

Why is this a nano service? Well, it really doesn't do much and it would be even simpler to package this as a library that other services can use and it does have all the management overhead of maintain as another system service.

What we did about it was to add similar functionality to the service so it also learned to send emails, tweets and MMSs. A serendipitous effect of this was that now instead of sending a request like TweetMessage or SendSMS to this service we could now raise more meaningful events such as SystemFailureEvent and have the service make decisions on how to alert administrators based on the severity of the problem etc. So combining the related functionality helped make the overall service even more meaningful.

Unfortunately it isn't always possible to take the functionality of Nano-services and find suitable "other services" (nano-or right-sized) that can assimilate them. In those cases getting rid of a nano-service is more of an exercise in redesign than is a refactoring. For instance, in a project we've built we had a services allocation service (SAS). The SAS role was to know about other services location and health status and utilization and upon a request, such as beginning a Saga (see chapter 5 for the saga pattern) decide what service instances should be used. The service also provided "reporting" capabilities for active sagas, services utilization etc. This might not sound like a nano-service, and at first we thought so too, but as the project progressed we found that being a central hub, as seen in figure 10.6 below, made the SAS a performance bottleneck, incurring additional costs (in latency) on a lot of the calls and interactions made by other services. The utility of the SAS, of finding what service instance to talk to, was being diminished by the cost – yep it is a nano-service after all.

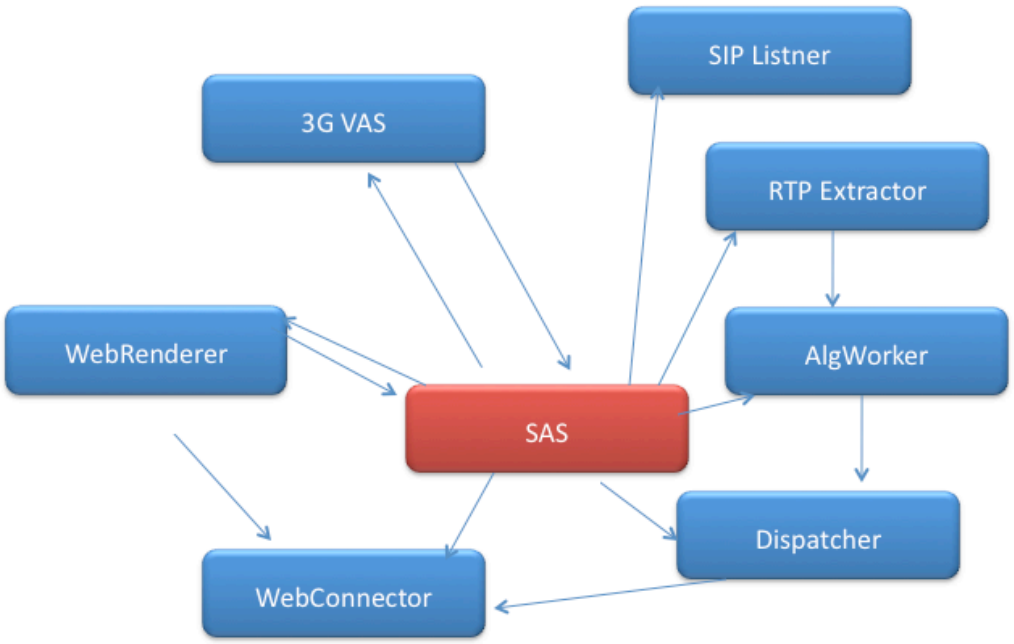

Figure 10.6 An example for a nano-service. The SAS service is a performance bottleneck as a lot of calls go through it. It provides an important service but the costs of its are too dear.

To solve the SAS problem we had to put in quite a lot of work. The solution, essentially was to move to distributed resource management, so that each service had some knowledge of what the world looks like so that it could decide what service instant to talk to by itself.

To sum this section, sometimes it is easy to notice that something is a nano-services, chances are that in these cases it would also be easy to take the functionality and group it with related functionality in other services. However on other occasions the fact that a service provides too little benefit is not as apparent and only becomes clear as we move along. In those cases it is also harder to fix the problem. One question we still need to cover is are there any situations where we would go with a nano-service even if we know it is one on the onset.

1.1.4Known Exceptions

When is it ok to have Nano-services? When you are starting out. When your approach to SOA is evolutionary and you don't plan everything in advance (something that rarely work anyway, but that's another story), there's a good chance that first versions of services you build will not show a lot of business benefit, but they will already need the full overhead of a service. The post office service in the example above is a good example for that as starting out it only dealt with a single type of message and it didn't do a whole lot with it either.

The post office service is also a good example for another reason to have a nano-service which is when you want to build an adapter or bridge to other systems be that legacy systems or 3rd party ones. In these cases you need to weight the advantage of using a service vs. building the same functionality as a library that can be used within services, but in many cases keeping the flexibility and composability of SOA can triumph over the overhead associated with having an additional service to manage.

Lastly, one point to keep in mind is that NanoServices is a rather soft pattern and the value of a small service can radically change from system to system or even in a certain system as time and requirements progress. It is worthwhile questioning our assumptions and looking at the services that we grow from time to time to validate the usefulness of what we're building.

Another SOA anti-pattern common for SOA newbies is what I like to call the 3-Tier SOA – trying to dress up 3-tier architectures of yore in SOA clothing.