SCA: Bringing Modern SOA Programming to Tuxedo

An Oracle White Paper
March 2010

Tuxedo

INTRODUCTION

Oracle, BEA, IBM, SAP, IONA, Siebel, and Sysbase introduced the initial SCA specifications in November 2005.

Oracle Tuxedo is one of the original SOA platforms. Everything in Tuxedo is a service and invoked as a service. Tuxedo also provides industry leading performance, scalability, and reliability. What Tuxedo has lacked is a simple programming model that lends itself to rapid development and easy reuse of services. By combining Tuxedo's robust native SOA infrastructure with the SOA programming model provided by the Service Component Architecture (SCA), Oracle SALT 10gR3 offers a compelling solution to developing high performance, scalable, and extremely reliable SOA based applications. This paper provides an introduction into this new programming model for Tuxedo based on SCA.

SCA – SERVICE COMPONENT ARCHITECTURE

In March 2007 version 1.0 of the SCA standards were published. In April 2007 OASIS took over the ongoing development of the SCA standards to allow broader input into their development.

Building SOA based applications has always been a challenge. While there are many standards defining how services should be invoked, interact, managed, and monitored, there have been few standards defining how services should be developed. In November 2005, Oracle, BEA, IBM, SAP, IONA, Siebel, and Sybase announced a new set of SOA standards focusing on how applications should be constructed and composed in order to build applications based upon reusable services. The goal of these standards is to simplify creating SOA based applications.

SCA Value and Principals

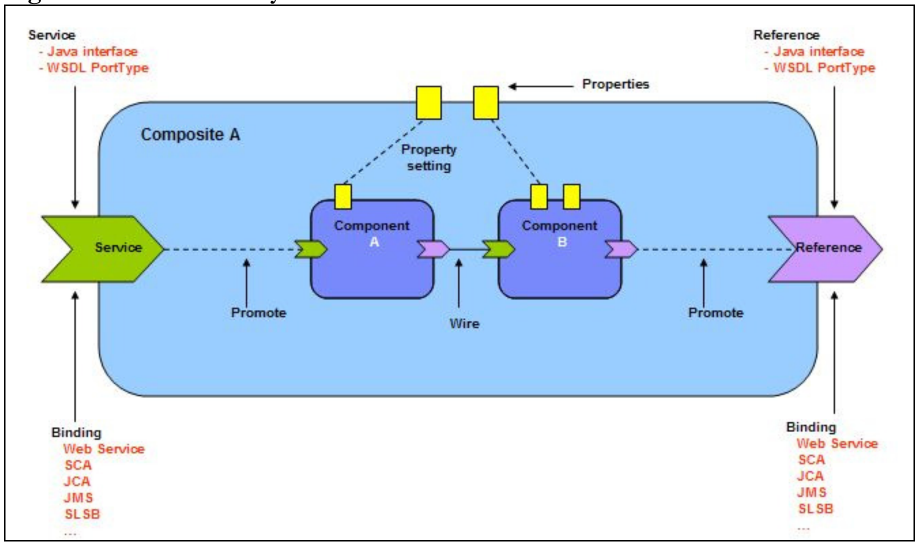
SCA supports the standard SOA principals of abstraction, loose coupling, service contracts, reusability, and composability. By separating the service definition from the service implementation and transport, different service implementations and transports can be chosen based upon the particular requirements such as performance, robustness, implementation dependencies, or other quality of service requirements. In the SCA model, the dependency of a service implementation on other services is described externally to the implementation allowing the choice of dependent service implementation and transport to be made without changing the service implementation.

SCA Assembly Model

Many different skills are required to construct large scale SOA based systems. Business Analysts examine the business processes within an organization and define business requirements for a system. Software Architects define how those business requirements should be mapped into a software system based upon existing and new software assets. Software Developers build the underlying software assets that will realize the business and technical requirements of the system.

These professionals work with different tools, different languages, and different processes. One of their major difficulties is in communicating the results of their work to each other. SCA provides a standard for defining components and how those components interact, independent of their underlying implementation technology or infrastructure. Thus a business process defined by a business analyst and implemented as a BPEL process can be represented and described in the same manner as web service defined by a software architect. Likewise the same component definition language can be used by the software developer creating a component in Java, C, C++, COBOL, or other programming language. This model is the SCA Assembly Model. Below is a diagram representing an SCA Assembly Model.

Figure 1 - SCA Assembly Model



This diagram shows a composite that consists of two components. Each component offers a service and uses a service. These are represented by the green and purple arrows respectively. Component A's service has been promoted to the composite, meaning that the service can be called from outside the composite. Likewise Component B's reference has been promoted to the composite, meaning the particular binding technology used to access the actual service is controlled by the composite's definition. The diagram also shows settable properties for the components that can be set externally to the components.

The SCA Assembly Model defines how components are described and related. Components can be providers of services as well as consumers of services and these relationships along with the service contract information is described in the Service Component Definition Language (SCDL). Using SCDL components can be assembled together into larger components or applications. Depending upon the particular SCA runtime used, these assemblies of components can be built statically or bound together dynamically at runtime.

SCDL – Service Component Definition Language is used to describe components, specify how components are wired together to form composites, and define the services offered and used by composites.

Service Component Definition Language

SCDL is used in SCA applications to describe components and assemblies of components called composites. It allows defining such information as the services referenced by the component, the services offered by the component, settable properties, and how components are wired together into composites. Composites in turn can be used as component implementations, thus allowing hierarchical construction of applications.

A component usually offers one or more services. These services can be defined using a variety of specification mechanisms such as Java Interfaces, C++ abstract classes, and WSDL. Access to the service depends upon the binding mechanisms that are supported by the SCA runtime. A binding in SCA defines how a component invokes the services of another component, and how the services of a component are offered. Common binding technologies include SOAP, JCA, and JMS. As well components usually make use of other services via references to those services. References also have an interface associated with them, and can be bound to a service through similar service invocation technologies.

SCA Implementation Models

Many if not most existing SOA standards deal with standardizing the communication between components. While these standards help ensure interoperability of components, they do little to help standardize the implementation of components. As well many distributed computing models tend to be API centric. This means that much of the model or standard is focused on defining a set of APIs the application developer can use within their application. Often these APIs are proprietary, although many such as the Tuxedo ATMI API have been standardized by various standards organization. All the services, options, features, capabilities, etc., are all exposed as a set of API calls or flags/parameters passed in API calls.

This API centric focus has several major drawbacks. While API standards can help with application portability, that is only true to the extent that the APIs are truly standardized, that there are multiple implementations of the standard available from which to choose, and the implementations adhere well enough. Instead what has typically occurred is that an application becomes tied to a single platform and becomes a prisoner of that platform requiring a major rewrite to move to an alternative platform. Good program design and modularity can help mitigate to some extent the dependence on these middleware APIs, yet the dependence is still present in some portion of the application.

The SCA specifications include Client and Implementation specifications that describe how users of SCA based technology can utilize SCA, while minimizing or eliminating the dependency on the specific platform providing the SCA implementation. This is done partially by shrinking or eliminating where possible the required APIs that must be used to implement a SOA based application. In the SALT SCA implementation, the only required API usage for a client is that to get a reference to the SCA runtime context and from that a reference to a service. It is possible that even these API calls will be optional in the future with the introduction of a dependency injection framework. For a service implementation, there are no APIs required in order to build a usable service.

One of the major advantages of moving the wiring of components out of the components themselves and into SCDL is that the same implementations can utilize different bindings without any change to the component. Thus instead of creating a component that offers a SOAP service creating using a set of SOAP specific APIs therefore tying the implementation to SOAP, the developer would create an SCA component implementation and not use any service related APIs. The decision of what underlying technology is used to make the service available is defined by the bindings in the SCDL.

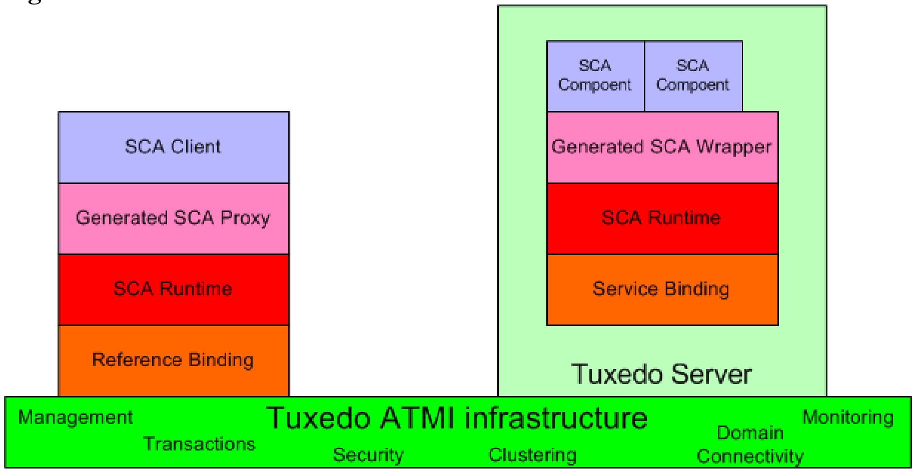
SERVICES ARCHITECTURE LEVERAGING TUXEDO – SALT

SALT was introduced in August 2006 initially as a Web Services gateway product that allowed SOAP/HTTP clients to invoke Tuxedo ATMI services. A year later in August 2007 the ability for Tuxedo applications to transparently invoke external SOAP/HTTP Web Services was added to the product. With the SALT 10gR3 release, SALT has been extended again to now support SCA. Unlike many products on the market, the support for SCA in SALT includes support for the SCA Assembly Model, the C++ Client and Implementation Model, binding extensions for Web Services and ATMI, and a robust runtime built on top of the existing Tuxedo runtime. This combination provides the best of what SCA offers with the market leading reliability, availability, scalability, and performance that Tuxedo has traditionally offered.

SALT 10GR3 SCA CONTAINER

The 10gR3 release of SALT includes a new runtime built on top of the standard ATMI runtime. This runtime effectively acts as a container to host SCA composites. SCDL is used to define which components and their corresponding component implementations make up each composite. The diagram below shows the relationship of the SCA container to the Tuxedo ATMI runtime.

Figure 2 SALT SCA Container



SCA Servers in SALT only need to implement the business logic to provide the intended service. All implementation details necessary to support any particular binding is provided by the SCA container.

SALT SCA Servers

Developing services using the SCA container is substantially easier and faster than developing standard ATMI services. This is largely due to the transparent handling of buffers by the SCA runtime. At a minimum a service must have a service definition or interface in the form of a C++ header file that defines an abstract base class containing only pure virtual functions. Each of the functions defined becomes a service. As well a concrete class that implements all of the pure virtual functions must be created. Together these are built into a component implementation in the form of a dynamic library using the `buildscacomponent` command. Here is an example interface definition for a simple four function calculator component::

```
namespace services
{
   namespace calc
   {
      class Calculator
      {
      public:
         virtual float add(const float addend1, const float
addend2) = 0;
         virtual float subtract(const float sub1, const
float sub2) = 0;
         virtual float multiply(const float multiplicand,
const float multiplier) = 0;
         virtual float divide(const float dividend, const
float divisor) = 0;
      }
   }
}
```

The concrete class or component implementation file for the interface above is:

```
#include <cctype>
#include "CalculatorServiceImpl.h"
#include "tuxsca.h"
using namespace std;
using namespace osoa::sca;
/**
* Calculator component implementation
*/
namespace services
{
  namespace calc
  {
    float CalculatorServiceImpl::add(const float addend1,
const float addend2)
    { return addend1 + addend2; }

    float CalculatorServiceImpl::subtract(const float
sub1, const float sub2)
    { return sub1 - sub2; }

    float CalculatorServiceImpl::multiply(const float
multiplicand, const float multiplier)
    { return multiplicand * multiplier; }

    float CalculatorServiceImpl::divide(const float
dividend, const float divisor)
    { return dividend / divisor; }
  }
}
```

There are no technical APIs in the concrete class. In fact, unless a service implementation needs to invoke another SCA service, there isn't a requirement to use any technical APIs. This dramatically simplifies the task of creating service implementations.

Once the components have been created, the `buildscaserver` command is used to create a Tuxedo server that will host the components. The Tuxedo server dynamically loads C++ SCA component implementations that make up the composites specified on the `buildscaserver` command line. `buildscaserver` parses the SCDL files contained in the application directory and determines which component implementations must be loaded and what services are to be offered.

Once the SCA server has been built, it can be added to the Tuxedo configuration as any other server. Upon startup the SCA server will advertise the SCA services that were defined in the composite used to build the server. Like any other Tuxedo server, multiple copies of the SCA server can be configured on multiple machines within a Tuxedo domain to support virtually unlimited scalability.

Tuxedo SCA Clients

SCA client in SALT invoke services as though they were local methods. The binding to a particular service implementation is controlled by the SCDL.

Clients are also assembled from components and described using SCDL. The `buildscaclient` takes a composite definition written in SCDL, source components described in SCDL, and component implementations written in C++ and creates a client executable. Below are the relevant snippets of code from a simple client for the four function calculator:

```
#include <iostream>
#include <stdlib.h>
#include "tuxsca.h"
#include "CalculatorService.h"
using namespace std;
using namespace osoa::sca;
using namespace services::calc;

int main(int argc, char* argv[])
{
  try {
  // Initialize the SCA context
  CompositeContext theContext = CompositeContext::getCurrent();
  // Locate the service
  CalculatorService* calcService =
        (CalculatorService *)theContext.locateService("CALC");
  try {
    // Perform the call
    float arg1 = (float) atof(argv[1]);
    float arg2 = (float) atof(argv[3]);
    char op = argv[2][0];
    float result = 0.0;
    switch (op) {
      case '+':   result = calcService->add(arg1, arg2);
                  break;
      case '-':   result = calcService->subtract(arg1, arg2);
                  break;
      case '*':   result = calcService->multiply(arg1, arg2);
                  break;
      case '/':   result = calcService->divide(arg1, arg2);
                  break;
    }
    cout << "Returned value: " << result << endl;

  } catch (...) {
    cout << "Failed to locate the service " << endl;
    }
  return 0;
}
```

The client code is nearly free of technical APIs. The first API call to `CompositeContext::getCurrent()` gets the current context for the composite. From that context a pointer to the CALC service can be obtained and then used to invoke the services simply as function calls. No other APIs are required to locate and invoke services.

SALT 10GR3 BINDINGS

In an SCA based application, components are wired together by way of bindings to specific transports. SALT 10gR3 supports both a Web Service binding as well as a bindings based upon the native Tuxedo ATMI infrastructure and Tuxedo workstation protocol for remote clients. A binding tells the SCA runtime how a reference or service is mapped to a specific transport. On the reference side a binding specifies how the method call is to be mapped to a specific transport's service client invocation mechanism. On the service side a binding specifies how a specific transport's service implementation mechanism is to be mapped to the SCA service implementation.

Web Service Binding

The Web Service binding allows communication for clients and with servers to occur via SOAP/HTTP. This binding uses the SALT SOAP gateway to handle the SOAP processing. The `buildscaserver` generates the required configuration and metadata information necessary to allow the SALT SOAP gateway to support the SOAP service as defined by the user provided WSDL. The client or server in this case may simply be a standard SOAP client or server and not necessarily an SCA client or service.

ATMI Binding

The ATMI binding allows clients and servers to utilize the native Tuxedo ATMI infrastructure to make or accept requests. The type of ATMI buffer to be used can be specified in the binding definition as well as how the parameters of the service are mapped to the fields of the buffer if appropriate. The example SCDL file below for the four function calculator client shows the binding specification for ATMI. In this case the ATMI transport will use Tuxedo FML32 buffers to carry both the service request and response. The transport will create fields in the FML32 buffer for each of the service parameters.

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  name="calc.client">
 <reference name="CALC">
  <interface.cpp header="CalculatorService.h"/>
  <binding.atmi>
   <inputBufferType target="add">FML32</inputBufferType>
   <inputBufferType target="subtract">FML32</inputBufferType>
   <inputBufferType target="multiply">FML32</inputBufferType>
   <inputBufferType target="divide">FML32</inputBufferType>
   <outputBufferType target="add">FML32</outputBufferType>
   <outputBufferType target="subtract">FML32</outputBufferType>
   <outputBufferType target="multiply">FML32</outputBufferType>
   <outputBufferType target="divide">FML32</outputBufferType>
  </binding.atmi>
 </reference>
</composite>
```

Interoperability with existing ATMI Clients and Servers

SCA components in SALT are Tuxedo components. An option in the ATMI binding allows normal ATMI clients to call an SCA service as it would call any other Tuxedo service. Similarly the ATMI binding allows an SCA client to call an existing Tuxedo ATMI service. This allows the developer to freely mix SCA and non-SCA components in the same application and reuse their existing ATMI services.

PULLING IT ALTOGETHER WITH SCDL

Once the service interfaces have been defined and the service implementations written, the remaining step is to pull everything together in a single description. This would be the top level composite which often consists of a client component and a server component. The server component is normally built out of a number of different components. Here is the top level composite for the simple four function calculator:

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
      name="calculator.app">

      <component name="CalcClient">
            <implementation.composite name="calc.client"/>
      </component>

      <component name="CalcServer">
            <implementation.composite name="calc.server"/>
      </component>

</composite>
```

METADATA AND SOURCE DRIVEN DEVELOPMENT

In creating SOA applications, developers often use one of two common approaches, metadata driven and source driven development. They primarily differ in the starting point for the definition of a server. In metadata driven development a developer starts with a service definition created in some form of metadata repository. This metadata is then used to create the initial artifacts that are required to develop the service. Source driven development uses a standard programming language source file as the service definition. This source file is often in the form of a header file that defines an interface or in C++ an abstract class that consists only of pure virtual functions. From this source file other required artifacts are generated including a metadata definition of the service.

Metadata Driven Development with SALT

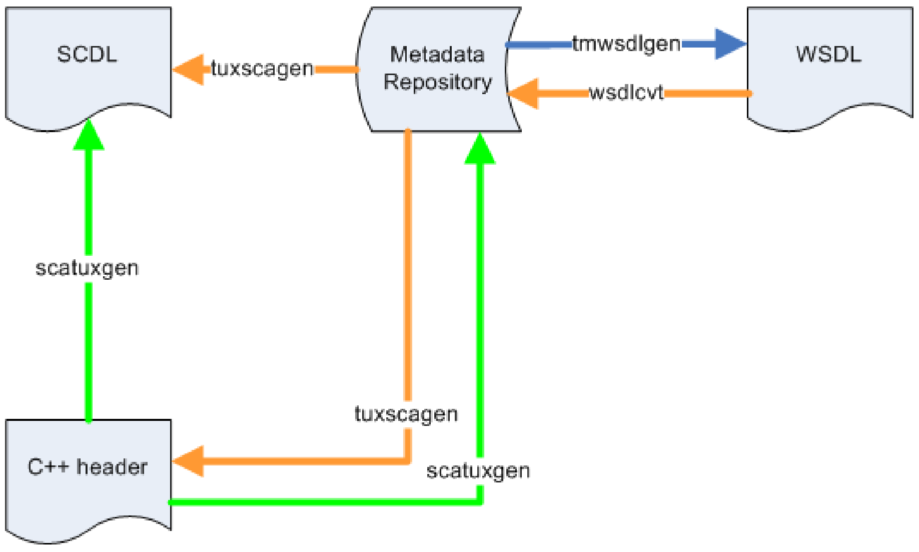
SALT provides the necessary tools in order to support metadata driven development. The developer starts with a service definition, usually in the form of a Tuxedo service metadata repository input file. This file describes the characteristics and contract for the service. It is a plain text file that contains a number of elements including the name of the service, the types of Tuxedo buffers to be used for the request, reply, and errors replies, and the names and types of each of the fields in those buffers. From this file, a developer can use the `tuxscagen` command to produce the C++ header files and SCDL files for the service. The developer then needs to create a concrete implementation classes associated with the abstract classes contained in the generated C++ header files. This greatly simplifies the development of SCA services in SALT.

Many organizations have standardized on WSDL as the preferred means of describing a service contract or interface. SALT supports the development of SCA services with WSDL as the starting point. Using the wsdlcvt command, a developer can start with a WSDL file, convert the WSDL into a Tuxedo service metadata repository input file, and then using the procedure described above, generate the C++ header file and SCDL files and then create the concrete C++ implementation class that implements the interface.

Source Driven Development with SALT

The 11gR1 release of SALT provides support for source driven development. The service developer creates SCDL files containing one or more binding.atmi bindings defining the services or C++ header files that contain abstract virtual classes containing only pure virtual functions. These become the starting point for the definition of the service. Using the new `scatuxgen` command, the developer can then generate the metadata input file necessary to populate the Tuxedo service metadata repository. This is especially helpful when taking existing C++ applications and turning them into SCA and web services. As SCA services, it is simply a matter of adding binding.ws bindings to the SCDL files to make these existing C++ applications available as web services.

The diagram below shows the two styles of development. The orange arrows show the path taken for metadata driven development while the green arrows show the path for source driven development.



DYNAMIC OBJECT ORIENTED SCRIPTING LANGUAGES

Application developers are more and more turning to dynamic object oriented scripting languages such as Python and Ruby to develop applications. These languages help decrease the development time required to create applications. New to the SALT 11gR1 release is the support for both Python and Ruby as component implementation languages. This allows developers to create SCA clients and services in Python and Ruby. The components developed with these languages can be used by other SCA components written in C++ as well as any ATMI based application. Since SALT supports SCA components invoking standard ATMI services, Python and Ruby components can freely call existing ATMI services.

Developing SCA Components in Python and Ruby

The creation of SCA components in Python and Ruby is done in much the same way as the development of C++ SCA components. The developer creates the SCDL files describing the components and includes either an implementation.python or implementation.ruby element to indicate that the component implementation will be in Python or Ruby. These elements define the Python or Ruby source file that will be used as the implementation of the component. The developer then creates the appropriate Python or Ruby script and adds a new system server called SCAHOST to their Tuxedo configuration. SCAHOST provides the runtime container for Python or Ruby components.

Services developed in Python or Ruby need not be aware they are being called as SCA services. As in C++ SCA service components, unless there is a need to invoke another SCA service, no SCA APIs are required. The scripts simply implement the appropriate business logic. Client components on the other must use a trivial API to look up a service from the component context before they can invoke the service.

CONCLUSION

Creating SOA based applications that provide extremely high reliability, availability, scalability, and performance has never been easier than it is now with SALT. By providing support for the Service Component Architecture, SALT allows customers to quickly develop and compose SOA based applications running on the most robust infrastructure in the industry. The addition of support for Python and Ruby in the 11gR1 release gives customers the option of leveraging the rapid development these languages provide, while still allowing near unlimited scalability, all in a SOA environment.

SCA: Bringing SOA Programming to Tuxedo
March 2010
Author: Todd Little
Contributing Authors: Maurice Gamanho

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com