Introduction

Service Oriented Architecture (SOA) refers to attempts towards making software services instantaneously available as if the services reside on the requesting computers. One has to distinguish between SOA and the actual services themselves. As an analogy, the railroad is not the same thing as the goods that an engine carries to distant locations.

C++ is the most suitable choice for fabricating services on any particular platform. Except for the most trivial ideas the implementation of services result in large and complex programs. Even the facilities of C++ are not adequate to harness such complexities, let alone the minimal linguistic constructs of pseudo-C++ languages. **The design of such languages concentrates on their GUI capabilities, but treats computational expressiveness as an add-on.**

There are two ends in providing a service, the consumer and the supplier. SOA is the medium that connects the two ends. This medium must be reliable and efficient, neither of which are met by XML proposals. Transformational requirements for adapting existing services for XML transportation are outrageous.

Z++ provides an evolutionary solution for SOA predicament. First, the existing services implemented in C++ need no change. Second, new services can be written in Z++, which is a superset of C++ with a profusion of scientifically engineered mechanisms for taming the complexity. The accompanying papers present several unprecedented features of Z++. In this paper we briefly introduce the modular capabilities of Z++ for providing sophisticated services.

Z++ example presented further down, illustrates the simplicity of implementing the ideas discussed in this paper. Despite its expressiveness, Z++ is an intuitive language.

Modules and SOA

A Z++ executable can run as a standalone program, or it can be invoked as a remote or local module. That is, every executable can invoke any executable in a manner similar to loading a DLL, a loadable module etc. The set of entry points of a module constitutes its boundary with the outside world. Technically, an entry point is like an exported function in a DLL. Conceptually, the set of entry points is the module’s interface.

A module can be a very large program. One can either load the module into the local machine or have the remote machine execute it. Thus, a server can distribute its services in order to achieve an efficiently balanced load. Data transport is fast, in small binary packets. Furthermore, arguments to an entry point can be any class type, including those derived via multiple-inheritance.

As for exposing existing C++ services, the Z++ compiler automatically generates a C++ DLL from a set of C++ DLLs. One merely has to compile the generated DLL using a

C++ compiler. The result is a Z++ wrapper that exposes services to the outside world. Here, we only mention DLL libraries because they seem to be most widely used.

Examples

In this section, we illustrate the simplicity of exposing and using services in Z++. Suppose we wish to use services of a module called Math, residing on a remote server at some URL. The Math module may have hundreds of entry points. For our purposes, we consider two of them, say Add and Multiply.

We write a Z++ program, and define the following type in that program.

```
class MathUserType = “URL/Math”<remote>
    // members and private methods
public:
    // constructors, other methods

    external double Add(double, double);
    external double Multiply(double, double);
end;
```

The specification remote means that we want the remote server to execute the calls. Omitting it, or specifying local instead, would result in receiving the module on the local machine. Note that Z++ is platform independent so that a module can be loaded into any client (in binary). The remote option is suitable for large modules that provide many services, or when the server disallows downloading the module.

The equality symbol after the name of class `MathUserType` associates this class with the module Math. The specification “external” identifies the services. These are prototypes of entry points of Math module. The compiler generates the code for the body of external methods. Thus, **you only need to know about the services, and the URL to the server**.

Now let us look at a code fragment to use the services. You simply declare an instance, and then invoke the methods on the instance, just as you do with any class. All protocol negotiations happen at the point of declaration between Z47 processors. Requesting a service is just a function call. Cleanup is done for you when object goes out of scope.

```
MathUserType MathInstance; //connects to server, etc.
double result = MathInstance.Add(5, 3.2);
output << “Sum is: “ << result << ‘\n’;
output << “Product is: “ << MathInstance.Multiply(5, 3.2) << ‘\n’;
```

This is as good as it can ever get. For a C++ programmer there is no learning curve. For the software company there is no tools to buy. Furthermore, regardless of changes to operating systems, or even as newer PDA emerge, this code is permanent and requires no maintenance resulting from changes in the underlying infrastructure.

Conclusion

Initially, the services provided via SOA may be limited to simple cases so XML can be deployed. Before long, XML will create unmanageable complexities resulting from its well-known limitations as a solution for SOA domain problems. However, until legacy programs written in languages other than C++ are retired, XML and CORBA are among viable options. The solutions will be expensive and murky bundle of patches, though.

The implementation of contemporary services already demand linguistic facilities beyond C++. Furthermore, the details of SOA must be transparent to the services for several reasons. First, **such details influence the implementation of the services**, and create a distraction from their intended design. Second, the services will most likely require expensive maintenance whenever the SOA implementation needs upgrading.

The Z++ solution presented here is an upward evolutionary extension of C++. The SOA is transparent to the intended services and requires no tools or maintenance. In other words, there is no need for language and technology experts for the delivery of services, beyond the C++ engineers who craft the services.