RECURSION SOFTWARE, Inc.

Software for Software Developers™

VOYAGER®

The Voyager SOA Platform

SOA Without all the Framework Baggage

By Daniel Brookshier
Java Architect
turbogeek@cluck.com

Recursion Software, Inc.

September 9, 2005

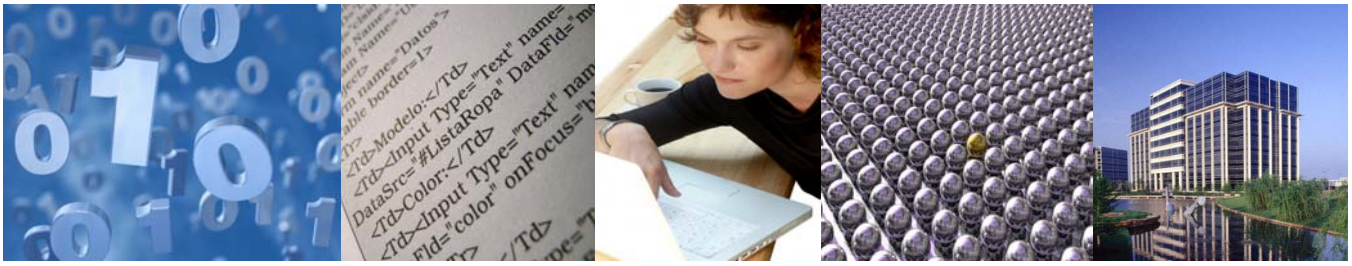

TABLE OF CONTENTS

The Voyager™ SOA Platform

SOA without all the Framework Baggage

Solve Problems without wasting your time learning overly complex SOA code and configurations.

Service Oriented Architecture (SOA) is now the framework for most new development. SOA is being added to almost every application. We have a lot of choices to accomplish the goal, but which one gets you there with the least risk of failure?

You may have noticed that most SOA code is complex, difficult to maintain and debug, and is hard to integrate with legacy systems. In most cases there is a lot of glue code and/or a lot of XML configurations or code injection required to make these things work. In exchange for simple services we have frameworks from hell.

Could Voyager® provide a simple and better SOA? In my case it has. I have not had to write XML descriptors for objects or use code injection or any other hack with Voyager. I also have faster, cleaner, and more maintainable applications. I can also use Voyager remote objects and messaging or CORBA and SOAP, all without the normal hassles other frameworks require.

I have found my way to an architect's ideal SOA, but is Voyager your solution? I can't say for sure, but you can probably make a choice by the end of this article. I hope you find Voyager a great choice as I have.

Finding Voyager

Voyager has every thing I was looking for. Using run-time discovery, run-time class proxies and other magic, Voyager lets me create a service in a couple minutes – literally. My code is clean and uncluttered and I did not have to learn an annotation system or complex config files. It looks like my original design instead of a patchwork Frankenstein's Monster!

I can also use Cinergi™, another Recursion Software product to do all sorts of quick and worry free legacy integration, even between languages like .NET and C++ or Java and .NET. Cinergi uses the same approach as Voyager, so most of the work is done for me.

Voyager supports transactions, security, multiple message types, load balancing, and much more. The system also supports .NET, SOAP, CORBA and others.

But the magic of Voyager's proxy objects and automated runtime mean that all of the capabilities are abstracted to a common framework API. Again, because of Voyager automated nature, the code is cleaner than a normal API that has to expose many of the details to the developer.

When you have found Voyager, you have found a system that works now, and in the future. I am right now working to integrate with Peer-to-Peer technology, which was not conceived of when Voyager was first created. But like the SOAP features it now supports, P2P is just a little coding to add its capabilities to all Voyager systems with little or no work by the end developer.

Easy to Learn, Easy to Use

Imagine learning EJB or SOAP in 5 minutes? Can't be done, right? I learned the basics of Voyager in 5 minutes. Easy is nice, but is it powerful? Let's see why it is the best SOA solution on the market.

- Use un-altered Java objects (POJOs).
- No messy wrappers or compile time code injection, or complex XML configuration files.
- Add/Modify/Remove Services at Runtime.
- Use remote calls or a rich messaging API that support async, sync, broadcast, and more.
- Easy to learn and use.
- Under-the-covers magic means Voyager doesn't require lots of boilerplate code or complex XML files.
- Stable, proven and robust.
- Connect with C, C++, CORBA, DCOM, .NET, and SOAP Services and even legacy Java binary code.

Voyager does all this faster and scales as far as you can imagine. Remember that I also said it was easy to learn and use. Let's take a moment to see how easy with a few code examples.

One Minute Voyager

Voyager is one of the simplest frameworks I have ever used. To prove it, here is a hello world example that takes about a minute to describe.

First, an interface and its implementation. Note that there is no Voyager framework referenced in the code. This is all just plain Java code.

Hello.java
```
public interface Hello{
  String sayHello();
}
```

HelloImpl.java
```
public class HelloImpl implements Hello{
  public String sayHello(){
    return  "Hello  World";
  }
}
```

Now, to add this to a distributed server (note that I am registering it with the name 'hello'. The code is run to add it to the Voyager server at the host URL.

```
Voyager.startup(8000);
Hello hello = new HelloImpl();
Namespace.bind(host+"/"+"hello", hello);
```

Now, we get access to the class from a remote client by looking it up by its registered name, and run the method:

```
Voyager.startup();
Hello helloProxy;
helloProxy=(Hello)Namespace.lookup(host+"/hello");
System.out.println(helloProxy.sayHello());
```

There were no config files and all I did was compile the code and run it. That's all there is to it.

Can you imagine doing EJB or SOAP in that few lines?

The HelloImpl is now available to the server and any client. I can also invoke it from a remote client via SOAP, Voyager, RMI, or CORBA.

Voyager Messaging

I could also invoke the call synchronously or asynchronously. Here is a simple invocation of the sayHello() method. I'll make the call asynchronously (without blocking). When I am ready, I get the result that returns the value, or blocks until ready.

```
Result r;
r = Future.invoke(helloProxy,"sayHello",null);
… do something …
System.out.println(r.readString() );
```

This little bit of code solves huge problems that we always face with remote objects. Simply put, this allows us to write code that is not blocking on long processes or slow communications. More importantly, the code is uncluttered by the framework and I don't need to create my own threads to manage the process.

I can also do callbacks, blocking and non-blocking messaging. All are written in the same way. Publish and subscribe is just as easy.

Voyager Mobility

Voyager has had support for mobile agents built in as a core feature since 1998. Mobile agents are not commonly used, but what if you could move services around to balance load or improve efficiency in a cluster of servers or grid? Here is an example of moving the service:

```
IMobility mover = Mobility.of( helloProxy );
mover.moveTo( anotherHostURL );  // move service to remote host
```

The same service has been moved, including the class's code, in just a couple of lines. This can really change the way you work with SOA.

Dynamic Proxy Objects

How does it work? How can we get such unprecedented dynamic behavior without burdening a developer with a complex framework or dozens of XML configuration files?

Voyager creates proxy classes that implement the interface(s) of the service object. The proxy class uses introspection to configure itself. Voyager does all the dirty work. In addition, Voyager uses distributed garbage collection to automatically track remote references. You don't need to manually "release" references.

Voyager & SOAP

The promise of SOAP and .NET are said to be the future of SOA. They indeed do have a place but many applications are not suitable because of many factors including the inefficiency of XML and the static nature of SOAP. Voyager is both an alternative to SOAP and can be integrated with existing SOAP services.

Use Voyager in high volume applications where scalability and performance matter. Integrate with SOAP when working with external services and use SOAP to expose Voyager services when your customers and partners want SOAP.

Voyager And the Future

Voyager would not be here if it was a static framework. Voyager's core has not changed since 1998, but because of its proven core architecture concepts it has stayed current. For example, as SOAP became available it was added as a protocol option with little effort and no impact to existing users.

Voyager does all the work of creating the SOAP configuration and can even update the lookup service. Within seconds of having a Voyager object we can have a SOAP service. The same is true for CORBA and interfaces to other SOA frameworks.

Voyager In The Real World

One of the best ways to show how a software product works is to use a real world example.

Voyager On Wall Street

My first example is also the biggest, a Wall Street trading company's infrastructure. This is a little different from talking about something like J2EE as middleware. Instead, imagine many computers and dozens of applications with thousands of live objects that the applications use, share, and interact.

Here is what the customer was faced with:

- Dozens of independent applications.
- Common live data shared between applications.
- Near-real-time equities trading.
- Event generation from changes to live objects.
- Had to be tight enough for Wall Street auditing and FCC scrutiny.
- Compatible with legacy systems.

There were also practical considerations:

- This was a core infrastructure change and needed to be the right choice for many years.
- The project was in-house so little outside help was wanted and in fact not desirable.
- Ease of use to ensure quick time to market for any new application or modification.
- This was the core to all trading and their front/back office applications. In other words, this is the whole ballgame for a Wall Street icon.

The customer had many options. By looking at what they needed, you might be thinking Object Oriented Database but that would only solve persistence. EJB was also the wrong choice. First, it was hard to mix events and messages with remote objects, and second, application servers did not have the requisite performance and scalability. This application needed to operate on live in-memory objects across systems with a rich messaging model or it would never meet demands.

Most of the competing solutions had other issues caused by architecture design verses implementation. These solutions were hard to re-use repeatedly.

A couple of close contenders were RMI and CORBA. They were considered, but abandoned. RMI because it did not scale well plus did not support transactions and various messaging scenarios, and CORBA because it was not dynamic and hard to integrate.

Voyager won on several top points:

- Ease of use.
- Live Objects with dynamic messaging.
- Ease of understanding by developers.
- Stable technology with no surprises.
- Low design to implementation impedance.

- Dynamic runtime configuration.
- Supported ACID transactions and security.
- Rich Messaging capabilities.

Let's take a simple piece of the completed system to understand how it is used. Start with the data in a customer's account with a portfolio and trading triggers.

The application is monitoring the state of live objects like stocks. The objects are shared by the automated trading application that registers via publication and subscribe messaging a specific trigger value.

When the trading program gets the trigger, it then asks other systems to do the pre-defined trade. This causes a cascade of queries and events like checking the user's funds to ensure there are enough funds.

Things get complicated just prior to when the trade is committed. The negotiation of the trade has to be finally locked and the conditions validated. With thousands of trades per second, conditions can change quicker than the setup. Once validated the trade is finally transacted and locks released.

The completion of the trade is only the start. The live objects trigger other applications to cause them to take further action. The live objects themselves either sends messages directly or connected via publish and subscribe. This reduces both the complexity of the trading system and all the applications that depend on its actions.

An example of a few other applications triggered by the trade:

- Customer notification
- Customer accounting
- Transaction logging

Remember that there are dozens of applications in the current system, so there are a lot of possibilities from online trading to portfolio management and institutional investment. All applications are running on the unified middle office software with all objects in Voyager's domain.

This is probably one of the largest and most complex systems running Voyager. It has the most objects and is capable of transaction speeds far greater than average frameworks and much greater than pure SOAP SOA.

The average architect is confronted with far less, but it is nice to know Voyager is up to the biggest tasks.

SAN Management System

The second system to talk about is a little less complex. It does have many of the same requirements of the trading system.

Storage Area Network (SAN) is a high-speed network of shared storage devices. SAN is used with large arrays of drives and their controllers. The basic idea is to treat these as one device that is shared by many servers on a high-speed network separate from the LAN.

SAN is relatively complex to configure and manage. A SAN device is essentially a server that only concerns itself with storage and retrieval, but there is still a lot to worry about.

SAN does everything from managing backups, replicating data, managing users and application space usage. The SAN also has to report on usage, errors, and be able to tune and reconfigure parts of the SAN.

You can see that there is a lot going on here. There are many requirements that point to Voyager's strengths.

- Multiple management consoles needed to be run for the same network.
- Consoles needed to work within the customer's network or by system operators from their homes or remote offices.
- External management of the SAN for the customer as an additional service.
- Consoles associated with a single SAN needed to coordinate to share data and to avoid clashing commands.
- Different types of consoles were also needed from a vendor, operator, and manager points of view, but with many live objects in common.

- The SAN software and the consoles need to be able to dynamically load modules for updates and to work with new SAN equipment.
- Dynamic software to aid in monitoring and to add services or unique customer driven applications.
- Dynamic enough to be able to add new drivers and services for new equipment yet to be invented
- Compatible with other technology for integration with partner software.

They also have the same general requirements as many other customers:

- Mature technology
- Easy to learn framework
- Low code injection requirements
- High speed, scalable operation
- SOA compatible architecture

Voyager of course was chosen. Most of the distributed technology stack is Voyager that covers it all from remote objects to messaging. The implementation was delivered in the expected time and is very stable.

The architects are happy, as are the developers who built the system. Management is pleased with the results and even more pleased by a very customizable solution delivering value for many years.

The SAN control system has been a success for their customers too. It is now the flagship product of the company and one of the best reasons to buy their equipment.

Voyager and You

Voyager wins on many fronts. It mixes well with designs. It is scaleable and maintainable and even leaves room for the future. Voyager is also kind to the programmer by being easy to learn and simple to debug and maintain.

I hope you can see why I love Voyager. You can see that there are many reasons why Voyager fits well in your SOA strategy or in a general distributed architecture strategy.

About the Author

Daniel Brookshier, is a freelance consultant, mentor, architect, speaker, author, and Java Geek since Java 1.0. Daniel specializes in UML, SOA, and peer-to-peer computing. Daniel's latest book is “JXTA: Java P2P Programming” and working on his next book: “P2P & Distributed Computing - Patterns and Anti Patterns”. He can be contacted at turbogeek@cluck.com.

Recursion Software, Inc.
2591 North Dallas Parkway
Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com

RECURSION SOFTWARE, Inc.