



Simplifying Web Infrastructure with SmartVariables

Lee Hounshell
SmartVariables.com

Abstract

“Efficient Communications.” Now, that’s important. It is also something we often take for granted. Gaps in information efficiency and integrity arise when one system’s information should really be integrated with another system’s data, but they can’t be due to some incompatibility. Integration would be much *simpler if the data worked more like “Lego pieces.”* But often systems development tools are overly complex and integration is difficult. The idea of “simple data sharing” has shrunk leaving only an elite few with knowledge to build interconnected applications. Intelligent information sharing is often not simple to implement; “web services” do not represent a mature industry.

Imagine all information can be box-classified and placed into either a smart list, or inside a smart data container. Information becomes one of: “containe¹,” “list” or “element².” We all use an email address to communicate with online friends. What if we could attach a “machine address” *name* to these collections? By “naming” data using syntax like “key@hostname,” the information (inside a smart container or list) could persist itself, and also live-update active copies in other computers. By behaving like “distributed shared memory,” a network aware “smart container” or “smart list” object could provide a convenient, simple and safe method of “hooking systems together” that need to communicate or share. For ease of use, make these containers and lists also behave like ordinary program variables and arrays of program variables. By doing that, the ordinary working (and active) computation data becomes directly and automatically network shared. Application logic simplifies, as data buffers, complexity and infrastructure disappear. When any named data value changes, all network copies (inside all active programs on all machines accessing that) transparently change to reflect the new value(s). Applications could even be explicitly notified of changes, asynchronously.

Now stop imagining... **SmartVariables** is a new open-source product that does exactly that. Simplification in design and implementation significantly reduces development, training and maintenance costs, and reduces time to market while yielding higher quality products. Legacy systems using this new architecture are easily modernized without being rewritten. This paper shows how SmartVariables technology simplifies data sharing tasks.

¹ The data is self-contained as a unit, and can be conceptually put into a single box, called a *container*.

² Each list-element holds either a container or another complete list, including all items contained inside that list.

Introduction

Today's networked systems are often highly complex, and usually difficult to build and maintain. Much of the complexity exists in code to "handle the information," and to acquire and manage that. A non-trivial amount of programming in each application is for:

- locating data that meets some criteria
- moving data across buffers
- figuring out what it means
- designating local and/or remote storage
- transforming the data for security
- notifying systems that might care
- handling errors
- and navigating internal memory limitations

It takes too much technical knowledge³ and training to build today's networked systems. We need a way for scientists and junior developers to directly, simply and quickly implement ideas. If we could "strip-away" code details for finding, storing, accessing and sharing data, then application complexity and code size shrinks. SmartVariables "hides" the framework so that we can get on with building complex systems, easily. Transparent program logic leads to more maintainable systems, so engineers focus on solutions, instead of managing pieces. Simplification lowers costs.

Existing products are fragmentary solutions; none of them automatically manage data end-to-end, automatically transporting it into multiple distributed executing processes. An application's working in-memory variables and arrays should self-propagate directly into programs running on other machines. Because that product didn't exist, we built SmartVariables. It is useful for Distributed OO Databases, GRID programming, Web-Services, Directory, B2B, P2P, Legacy System Integration, Spiders, Distributed Neural-Nets, Messaging and Distribution systems.

The SmartVariables environment is designed with the idea of removing the "buffers" separating common data in different systems. SmartVariables makes information "shared" so that all copies update their values transparently. Objects dynamically share raw, working-data across multiple machines and processes in a way that abstracts-out: the machines, processes, network, database, types, and persistence code. SmartVariables objects have novel functionality: information is auto-shared and objects behave like they exist *inside futuristic "global" network-shared RAM*⁴.

The environment is a well-crafted, asynchronous "push" technology⁵. Objects *remember* who has copies of their information, and so the objects themselves are responsible for forwarding and integrating changes to those working with that data – regardless of physical location. The system also hides locking code, applying queued object changes securely in a first-come, first-serve manner where no object updates are lost. Our design even encapsulates *Web-Service*⁶ and *Directory-Service*⁷ functionality using only *Container* and *List* objects. A simple object "finder" provides ability to search for information patterns. Our intention: simplification on all fronts. *Eliminate most, if not all of the data-management code in applications.*

³ After dealing with MPI and PVM style cluster systems, and XML, SOAP, WSDL, UDDI, CORBA and EJB, we realized there must be some way to reduce complexity and at the same time increase maintainability.

⁴ Random Access Memory (RAM) that emulates the NUMA (Non-Uniform Memory Access) shared-memory architecture.

⁵ An Internet technology that sends prearranged information to users before they actually request it.

⁶ Web-Services allow business to exchange data, application-to-application, without detailed knowledge of the other's systems.

⁷ A Directory Service is middleware that locates the correct and full network address from a partial name or address.

Generic container objects hold and work with any piece of information. Lists of these containers are indexed associatively – with strings, intrinsic types, or even other SmartVariables objects. The objects use a natural syntax to access heterogeneous, distributed information. In SmartVariables, the characters [] index list-elements, just as with standard C++ arrays; however, indexes are not limited to scalar values, and will *create* non-existing elements. Information network-persists when named, updating all copies. To access remote data, the name contains a suffix of the form “@machine.” Making the object’s name look like an email address is natural. Using the syntax: “name@machine” also provides a database key. Objects can be easily “subscribed” to, by event-driven applications. The platform uses “plug-in” libraries to incorporate 3rd party low-level physical databases (for object-persistence), and for network data transport. The database plug-in⁸ contains a tiny, common set of generic database methods that are easily rewritten to add support for new physical databases. The transport plug-in uses two methods: send, and receive. Using plug-ins allows complex new object behaviors to be introduced *without needing to recode applications, or even recompile* them!

Our primary design consideration in developing this generic network programming environment was and remains *API simplicity*. We made SmartVariables so easy to work with that junior programmers can quickly learn to build complex networked applications. It is versatile enough for scientists to construct powerful GRID⁹ and parallel-processing systems, as well.

⁸ The SmartVariables database plug-in, named libPlugDB, uses 9 generic methods to manage database information.

⁹ GRID computing harnesses unused processing cycles in a network for solving problems too intensive for one machine.

Programming Basics

Below is a simple example of a distributed system consisting of three computers named: Alice, Bob and Charlie. To begin, our program running on “Alice” will function to continuously print out the contents of a remote container object named “greeting@Charlie.” Here is the code for Alice:

```
Var greeting;
greeting.Name( “greeting@Charlie” ); // attach to and subscribe to the remote object

while (1) {
    cout << “greeting=” << greeting << endl; // note that ‘greeting’ can change values here
}
```

Note that Alice’s display code is in a tight-loop, and there is no code that explicitly connects to machine “Charlie” to retrieve the “greeting” object or any changes made to it. Next, we run another program on machine “Bob” that simply changes the value of the remote “greeting@Charlie” object to be the string “Hello, World!.” Here is the code for Bob:

```
Var greeting = “Hello, World!”;
greeting.Name( “greeting@Charlie” ); // modify all copies, everywhere.
```

Now, when the above program on machine Bob gets executed, it transparently connects to Charlie and modifies the “greeting” object to have its new value: “Hello, World!.” Because SmartVariables containers “know” who have copies of their data, the environment transparently propagates the change to Alice. SmartVariables propagate themselves into process-level code automatically. This means that the program still looping on Alice will now begin printing its new value of “Hello, World!.” The code on Alice appears to be a “tight loop,” with no opportunity for the object to be modified; however, *it does change*. Modifications to the “greeting@Charlie” object become automatically reflected by Alice’s program output. This behavior is quite amazing to see.

A “callback” method can be attached to objects. Callbacks allow programmers to write efficient interrupt code that executes when an associated named-object changes values. The *old* value is also accessible, should that be needed. This environment has many other capabilities also. For example, containers can do *infinite precision* integer math. Read the tutorial section of the smartvariables.com web site for a complete discussion of programming capabilities with examples.

In all, there are 6 SmartVariables object-types to learn¹⁰:

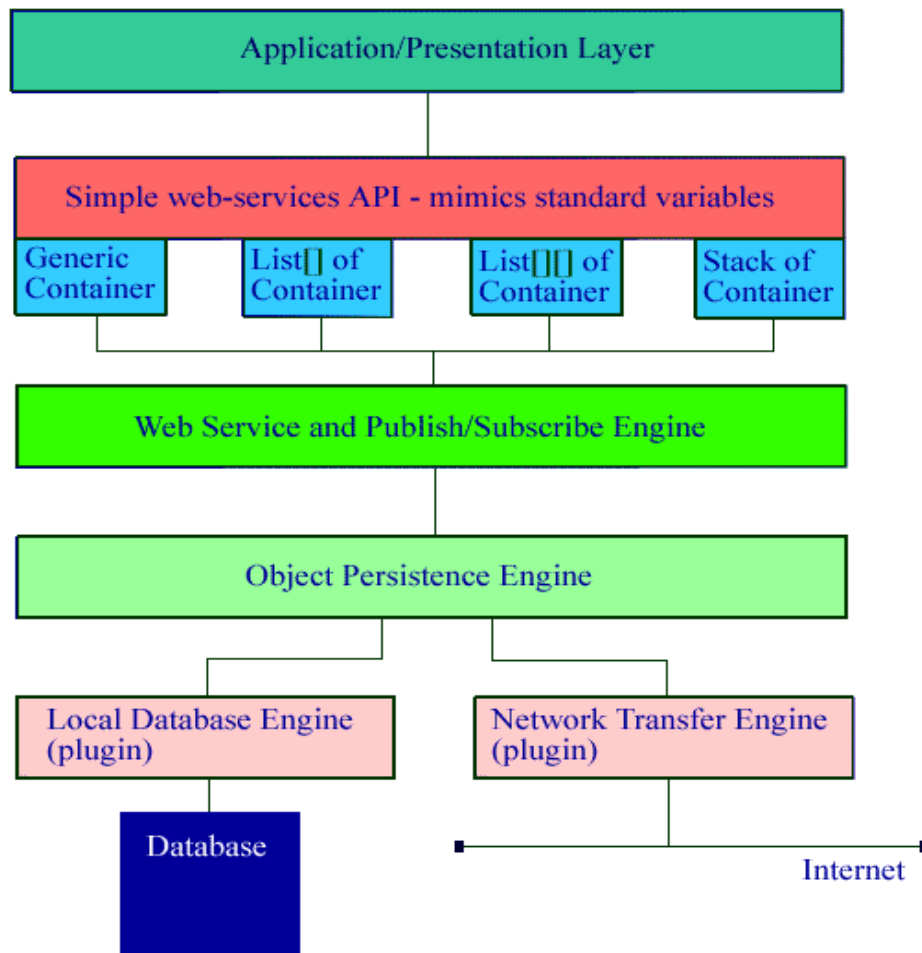
- **Mem** a fast non-persistent universal container
- **Var** a network-persistent universal container
- **Stack** a network-persistent “stack” of containers
- **MemArr** a network-persistent associative array of *Mem*¹¹
- **VarArr** a network-persistent associative array of *Var*
- **List<>** a template for making persistent lists of *Var*, *Stack*, *MemArr*, or *VarArr*

¹⁰ All six objects can be mastered by an experienced C++ programmer after only 1 day of study.

¹¹ Standalone Mem objects can not persist; however, the MemArr collection will persist as a single entity (including contained Mem objects).

Architecture

SmartVariables uses a layered architecture, intended to efficiently hide complexity, and provide extensibility. Access to the physical object database and the network layers are implemented via shared, *dynamically loaded*¹² libraries. Decoupling that functionality allows “swapping” components during deployment, without recompiling or relinking an application. SmartVariables encapsulates nine (hidden) methods that implement transparent support for any new physical database. These are: *Open (constructor), Get, GetNext, Set, Delete, Commit, Lexicographic, Recover and Detach (destructor)*. The database ultimately reduces everything into simple **key** and **value** pairs. The default network transport uses TCP/IP¹³ with customized auto-correcting error recovery/retry. Two methods (send and receive) implement a supplemental transport, like Sun’s JXTA¹⁴ environment. Out-of-service conditions are easily trapped and handled by the application.



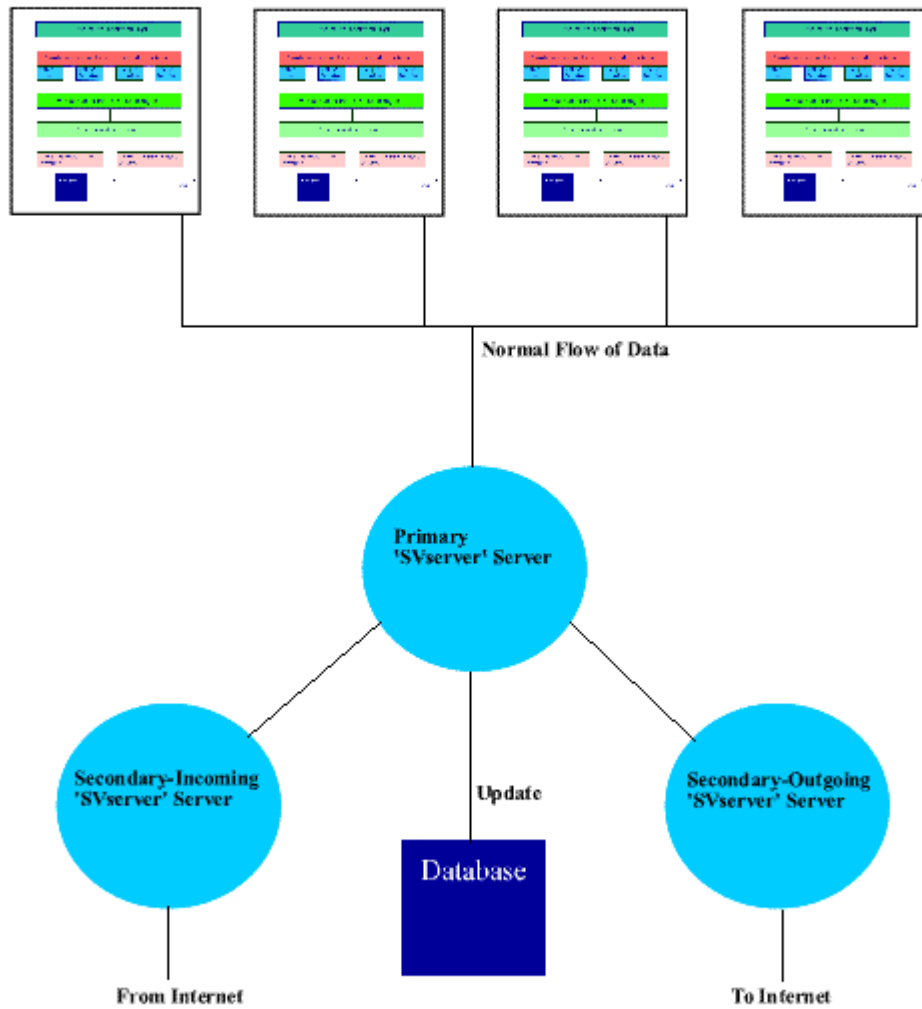
¹² Dynamically loaded libraries are code libraries that are loaded at times other than during the startup of a program.

¹³ The Transmission Control Protocol / Internet Protocol (TCP/IP) allows computers to communicate over the Internet.

¹⁴ JXTA is a network transfer technology that lets network connected devices communicate and collaborate.

SmartVariables transparently uses 3 background server modules to handle data storage, access and transfer. The primary server is responsible for managing the local database, including access to that and managing associated object subscriptions. Two secondary servers respectively handle incoming objects and object access requests as well as outgoing objects and access requests. This 3 server design will be expanded in future versions of SmartVariables to include additional dynamically allocated servers and support for dedicated private connections. This will improve efficiency when many machines all require access to the same set of remote objects. The current data transfer design looks like this:

single computer with 4 hypothetical applications



MPI and PVM style Cluster Systems

Modern *super-computing clusters* often use the MPI¹⁵ or PVM¹⁶ technologies for communications and parallel operations. Both of these message-passing technologies are complex to program and deploy. Programs written for either architecture are often cryptic and difficult to debug.

MPI lends itself to most distributed-memory parallel-programming models. It is commonly used to implement shared memory models, such as Data Parallel¹⁷, on distributed memory architectures. All parallelism is explicit, as the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using special MPI programming constructs. These constructs are not necessarily intuitive, but they do provide low-level parallel data access using a master/slave style architecture and explicit message passing.

PVM is a message passing system that enables a network of UNIX¹⁸ computers to be used as a single distributed-memory parallel computer. At the highest level, the *transparent* mode, tasks are automatically executed on the most appropriate computer. In the *architecture-dependent* mode, the user specifies which type of computer is to run a particular task. And in *low-level* mode, the user may specify a particular computer to execute a task. In all of these modes, PVM takes care of necessary data conversions from computer to computer as well as low-level communication issues. Task-to-task communication is done with explicit message passing.

SmartVariables technology works well as a replacement for both MPI and PVM based systems, simplifying such applications greatly. Systems built with SmartVariables don't need to worry about explicit message passing. New tasks can be invoked by using *Web-Service modules*. Programs always work directly with named-data, in parallel. Tasks are easily sub-divided and farmed out to additional web-services, as needed – without worry of breaking the natural parallelism. If two or more tasks ever access data of the same name and location, then that data is automatically shared between them – without need for additional parallel programming constructs. Instead of using configuration files with lists of available machines, a shared SmartVariables *List* object (with a commonly accepted name, like "machines@localhost") could easily hold the available host names, which can then be used for dynamic task allocation. The end-result is that SmartVariables-based parallel systems need only reference and work with distributed data, and don't need to manage it. Automatic sharing means there is no need to worry about explicit connection, infrastructure, or message-passing code. Instead, *applications only need agree on the names*¹⁹ used for their data. Names and object locations are easily managed by using a SmartVariables based *Directory-Service* as an additional layer of object indirection. The next sections of this document deal with SmartVariables based services.

SmartVariables is particularly effective for systems that present high *locality of data*. Locality is a measure of how much the calculation of one piece of data depends on other pieces "close to it." This is easily thought about for **GRID based computations**²⁰, where to update the values at one point in the GRID it is necessary to know the values at each of its neighboring points. Distributed **neural-networks**²¹ represent another example of a system class with high locality of data.

¹⁵ The Message Passing Interface (MPI) is a specification for the developers and users of message passing libraries.

¹⁶ Parallel Virtual Machine (PVM) allows a network of UNIX computers to function as a single large parallel computer.

¹⁷ Data Parallel is a programming model in which each processor performs the *same work* on a unique segment of the data.

¹⁸ UNIX is an operating system designed by Bell Laboratories that efficiently supports multi-user and multitasking operations.

¹⁹ Tightly bound code segments are replaced by a loose association of data that follows a strict naming convention.

²⁰ GRID computing links many computers on a network and divides a computing task among them for speed, creating a super computer.

²¹ Neural networks perform automatic *learning* and advanced pattern matching by mimicking a biological brain's function.

Web Services

The term *Web Services* describes programmatic interfaces made available to integrate internet-based applications, usually with XML²², SOAP²³, WSDL²⁴ and UDDI²⁵ open standards. CORBA²⁶ and EJB²⁷ may also be used for implementation, along with other technologies²⁸. Web services are primarily used by businesses to communicate with each other and with their clients. Web services allow organizations to exchange data, application-to-application, without intimate knowledge of the other's IT systems behind the firewall. Let's briefly examine some of these technologies:

CORBA is basically a remote-object access and control mechanism bundled with basic services. It handles information transport and operation execution in a distributed environment. Unfortunately, CORBA is deficient for use as a distributed object platform, mostly because the programmer API is far too complex and cumbersome to work with directly. Data is strictly typed and must be defined using the IDL²⁹ meta-language. When compiled, IDL produces copious, complex code that must then also be compiled and linked into an application in order to provide object distribution services. CORBA does not provide an integrated default database or transaction support. The primary advantage is that it works with a variety of languages on numerous architectures. Experienced programmers need weeks of training to become proficient in CORBA.

XML is an abbreviated version of SGML³⁰ used for tagging complex data structures and describing documents. It is designed to make it easier to define your own document types, and easier for programmers to write programs that handle them. XML omits all the options and most of the complex less-used parts of SGML for the benefits of improved understandability, and easy delivery and interoperability over the Web. XML is very useful for organizational purposes. Documents are parsed by the application using them, and must conform to a set of standard naming conventions. Note that *SmartVariables containers and lists can readily hold and share XML documents*.

SOAP is the specification for an XML based communications protocol that allows applications to exchange information, usually with HTTP³¹. All SOAP objects consist of three parts: an envelope that defines the framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP is a lot less complex than CORBA; however, SOAP is neither as simple to use nor as powerful or intuitive as SmartVariables because SmartVariables objects need only be *named* (1 line of code) to implement SOAP functionality.

EJB is a component model that promotes reusability by providing infrastructure and high-level object management services. The EJB specification defines a standard model for building *Java based* server applications. The programmer API is more complex than SOAP's, but not as bad as CORBA's. 3rd party add-ons are required to implement database persistence, object-relational mapping and data communications. EJB's primary advantage is that it supports high-level services, like transactions and event processing. A SmartVariables component model is much less complex.

²² Extensible Markup Language (XML) is used to tag complex documents, specifying what it is and how it should be formatted.

²³ The Simple Object Access Protocol (SOAP) is used to transfer data between machines, often using HTTP with XML messages.

²⁴ The Web Services Description Language (WSDL) is used for describing the available network services as a set of SOAP messages.

²⁵ Universal Description, Discovery and Integration (UDDI) protocol is a web directory used for listing what services are available.

²⁶ The Common Object Request Broker Architecture (CORBA) creates, distributes and manages distributed objects in a network.

²⁷ Sun's Enterprise Java Beans (EJB) is a Java API that defines the component architecture for multi-tier client/server systems.

²⁸ For example, **.NET** by Microsoft which simply incorporates the technologies: XML, SOAP, WSDL and UDDI to provide web services.

²⁹ Interactive Data Language (IDL) defines interfaces for enabling communication between language-independent modules.

³⁰ Standardized Generalized Markup Language (SGML) is the standard meta-language for describing the structure of documents.

³¹ HyperText Transfer Protocol (HTTP) is used to transfer hypertext files across the Internet.

CORBA, SOAP and EJB all provide a mechanism for invoking remote procedures, and sharing information. As we shall see, SmartVariables provides the same functionality, but is more loosely coupled, with less application complexity. Later, we also explain how SmartVariables associative, named *List objects* also make XML, WSDL and UDDI technologies unnecessary.

SmartVariables objects provide a powerful web-services environment by naturally decoupling new “public operations.” The environment does not directly “plug-in” server-side object behaviors, RPC³² style. However, any client is allowed to *attach behaviors* to any named object, in the form of a “change-notify” **callback** method. This means that we can attach a program function³³ that provides full web-service functionality to any SmartVariables named-object. When the object changes value, the callback automatically gets executed. The callback receives the changed object’s new values, which for web-services usually consist of command parameters and the *name of a results object*. The callback extracts the command information, executes the appropriate service code, and then places any results into the named results object. Execution results get returned privately to the invoking remote process by SmartVariables.

If object names and command-values are carefully chosen, using *named objects* and *callbacks* to implement command-request functionality works just like creating new “public operations.” The key difference is that with SmartVariables, nothing gets directly “plugged-into” the server. The client program providing the web-service’s “server” functionality must be executing³⁴ *somewhere* on the network; execution is not tied to the same machine as the named-object used for invoking it.

Putting command information into a SmartVariables *directory object*³⁵ is the same as publishing it. If the names and control-values for all “new operations” are stored in a directory object, such as a **list-of-lists of containers**³⁶ named “directory@someserver,” it then becomes simple for a remote client to retrieve the directory, and examine it to dynamically determine what “operation-objects” are available, and what “control-values” they currently use. Results will be privately returned if the command-request includes a parameter identifying the name of a results object. Prior to invoking a service the invoking process will have **subscribed**³⁷ to this results-object (thereby receiving execution results asynchronously via a container or list). This makes for a simple and powerful mechanism for anyone to implement and publish web-services *using only SmartVariables*.

Be aware that CORBA, SOAP and EJB do not provide for a built-in list service. Not only does SmartVariables provide an intelligent, intuitive, and fast list service, but also the list service is inherently heterogeneous. A single list object in the SmartVariables environment can be used to contain numerous types of objects simultaneously: numbers, strings, user-defined objects, binary encoded files, web pages, pictures, audio, etc... may all exist *inside the same list*. List elements can also index other (logically contained) lists, by referencing their names. This makes for powerful distributed-list processing to any level of containment or object-distribution desired. Although XML can be used, if you really want that, it is not necessary because list objects naturally provide a powerful means of tagging data and defining document structure: SmartVariables list *indexes* are associative, and naturally replace XML *tag* references, *eliminating the need to parse documents*.

The next two pages show an example of a server program providing a “ls” (file listing) web-service, and also of a client program that invokes this service, and then receives/displays the results. The two programs are complete, and will compile and run as shown.

³² Remote Procedure Call (RPC) is a mechanism for invoking methods on remote computers.

³³ In C++, the callback method uses the signature: `void myservicename(PStore *obj) {}`

³⁴ The client has subscribed to the command-request object and acts as a “server” for the new public operation’s execution.

³⁵ SmartVariables directory objects take the form of either lists of containers, or lists-of-lists of containers.

³⁶ **List<VarArr>** is the type for a SmartVariables list of lists of named containers.

³⁷ To subscribe to any object, declare a container (or list) and then **Name()** that object. Publishing is automatic.

First, let's examine the server code that is providing our example "ls" web-service:

```
#include <SmartVariables/SmartVariables.h> // one include gets everything we need

int done = 0; // if set, the service will terminate
void ls_service ( PStore *invoker ); // this method will service a client's "ls" request

int main ( int argc, char **argv )
{
    Var ls_service_entrpoint;
    ls_service_entrpoint.SetObjectChangeNotifyCallback ( ls_service );
    ls_service_entrpoint.Name( "LS-REQUEST@localhost", SUBSCRIBE );
    ls_service_entrpoint.DontPersistOnDestruct();

    while ( ! done ) {
        ls_service_entrpoint.PauseNotify(); // suspend until "LS-REQUEST" changes.
    }

    PStore::Cleanup(); // terminate normally
    return 0;
}

void ls_service ( PStore *invoker )
{
    Var * ls_request = (Var *) invoker; // upcast to get the changed "LS-REQUEST" object
    Mem unique_id = strtok(( *ls_request ), "|"); // the vertical-bar separates passed parameters
    Mem command = strtok( NULL, "|" ); // this should be the string "ls"
    Mem results_object = strtok( NULL, "|" ); // and here is where we send the service results

    if ( "ls" == command ) {
        Mem thepublicarea = GetSmartVariablesTmpDir();
        thepublicarea += FILE_DELIMITER;
        thepublicarea += "pub"; // this is the directory we will provide the "ls" service for

        Stack ls = myscandir( thepublicarea );
        ls.Name( results_object, SETVALUENORETURN ); // return the file listing here!
    }
    else if ( "quit" == command ) { // allow the service to terminate
        done = 1;
    }
}
```

The above service program registers itself with the container named "LS-REQUEST@localhost." The "PauseNotify()" method suspends server execution until some client modifies "LS-REQUEST." Whenever that container changes, the "ls_service" callback method will execute. To properly invoke the "ls" service, this container must be populated with a new value containing three "parameters." Here, we arbitrarily delimit these parameters using a "|" character. The first parameter is a "unique id" that just ensures "LS-REQUEST" actually changed values – so that the "ls_service" callback will be invoked. It is needed because SmartVariables will only notify the server when "LS-REQUEST" is assigned a value that is different from the current database value. Putting the "unique_id" into the object's new value ensures that the callback always executes. The second parameter we have hard-coded to "ls," which is the name of the service we expect to invoke. The last parameter contains the **name** of some SmartVariables object where the client expects to receive results in. Notice that strings can be directly compared to containers, and that standard C++ functions, like "strtok()," operate as expected. The "myscandir()" method is one of SmartVariables' low-level support methods (see Misc.h). It returns a "Stack" of file-names for the requested directory. Naming this Stack using the invoking client's "results_object" name returns the Stack object's contents to the invoking client. Now let's take a look at the client code used to invoke this service and retrieve the file listing..

The client code shown here is used to invoke the above “ls” web-service and print the service results. This code assumes that: the service is running on the machine named “localhost” and that the **name** of the service invoker object is “LS-REQUEST”; however, if the service hostname and invoker object’s name were not already known by the client program (a common situation), then a SmartVariables “*Directory Service*” could be easily used to *discover* that information. See the Directory section of this document to learn how to implement discovery using a SmartVariables directory service. Here is the client code used to invoke our “ls” service:

```
#include <SmartVariables/SmartVariables.h>

void display_ls_results ( PStore *results ); // this method will receive results asynchronously

int main ( int argc, char **argv )
{
    Mem results_object_name = “LS-RESULTS-“;
    results_object_name.Concat( smartgetpid() ); // the results name is unique to us.
    results_object_name.Concat( “@” );
    results_object_name.Concat( getdefaultipaddress() ); // full path to the results object

    Stack ls_results; // here is where the service results will eventually be returned.
    ls_results.SetObjectChangeNotifyCallback ( display_ls_results );
    ls_results.Name( results_object_name, SUBSCRIBE ); // first subscribe to the (future) results object

    Mem thecommandrequest = makemyuniqueid(); // first “parameter”
    thecommandrequest += “ls|”; // second “parameter”
    thecommandrequest += results_object_name; // third “parameter”

    Var invoker; // we will use this to invoke the “ls” service
    invoker.MakeServiceInvoker(); // service invoker objects only request from a single server
    invoker = thecommandrequest; // assign the service request “parameters”
    invoker.Name( “LS-REQUEST@localhost”, SETVALUENORETURN ); // INVOKE THE SERVICE!!!

    while ( ! ls_results.ChangeNotify() ) { // wait for the service results to come back
        if ( invoker.ChangeNotify() < 0 ) {
            cout << endl << “SORRY: the ls service is not available!” << endl;
            break;
        }
    }

    PStore::Cleanup();
    return 0;
}

void display_ls_results ( PStore *results )
{
    if ( results->IsOutOfService() ) {
        cout << endl << “SORRY: unable to retrieve out-of-service results data!” << endl;
    }
    else {
        Stack * ls_results = (Stack *) results; // upcast to get the “ls” service results Stack
        cout << “the service provider=” << ls_results->LastModificationUser() << endl;
        // ls_results->EnablePrintXML(); // uncomment this to see XML formatted results data
        cout << “LS SERVICE RESULTS=” << *ls_results << endl; // print the Stack of file-names
    }
}
```

The above program first builds the **name** for a results object and subscribes to that. Then it builds a command request with 3 parameters and assigns the command to a service-invoker object. Finally, once the invoker is named, we just wait for the callback to trigger - where the results get displayed. In the next section, we show you how old-style Internet *Directory Service* functionality (including WSDL and UDDI) can be easily replaced by using SmartVariables list objects.

Directory

A web Directory Service allows your application to find remote information efficiently by looking it up, like in a telephone book. But the traditional programmer interface for implementing directory can be both cumbersome and error prone. Let's briefly examine the UDDI directory protocol:

UDDI is an internet directory-service mechanism for finding existing web services. Each UDDI directory entry is an XML file that has three parts: a "white pages" entry that describes contact information for the company offering the web service, a "yellow pages" entry that organizes the web services based on standardized industry categories, and a "green pages" entry that describes the physical interface to the web service for applications trying to connect to that service. UDDI "green page" entries reference a WSDL file that is another XML document describing binding information as a set of SOAP messages, along with service specifications for how those messages must be exchanged. WSDL *unambiguously* specifies what a request message must contain and what the response message must look like. *Special tools*³⁸ must be used to read the WSDL file and to generate the code required to communicate with a published XML service. You first use the UDDI business registry to find business information registered by the appropriate business partner advertising the web service. After you have that information, you can get more details about a specific business service or about the business in general. Then, using the binding information, you can write an application to connect to the published web service. When the application runs, it uses the cached location and binding information to invoke the web service and to format the information in the way that the web service requires. UDDI exists for the purpose of letting your customers *discover*³⁹ and find out how to *interact* with your business; however, it is very complicated to use.

SmartVariables can implement your application's Directory Service far more simply. Like UDDI, a SmartVariables based directory requires that a developer standardize on **names**, contents and the purpose of directory objects. After that, it gets easy; everything can be done using only two named **list-of-lists of containers**: The first list we use for discovery. The second list contains our selected business's entire web service's directory. To begin, we show how discovery might work for the company "Acme Widgets." To do discovery, we initially need to get a *master directory* object. The master directory is a *directory of directories*, and for our example it holds only the **names** of various businesses' web-service directories, each indexed according to standardized industrial categories. Imagine that "discoverydirectory@discoveryserver.com" represents the name of a master directory object containing information about lots of businesses. We want to get the directory information, but we don't really want to subscribe to that. The following two lines of C++ code retrieve our master directory:

```
List<VarArr> masterdirectory;  
masterdirectory.Name( "discoverydirectory@discoveryserver.com", GETVALUENOSUBSCRIBE );
```

SmartVariables is asynchronous. This means our application must wait for the master directory data to arrive. There are two ways this can be done: either by using a *callback*, or by writing code that explicitly waits for the object to arrive. We show the explicit way first. Our C++ code will explicitly wait for our master directory information, and also check for any *out-of-service condition*. This example discovers every business that deals in "widgets." Once we have a list of businesses, we need to select one of them. In our example, we just take the first business in the list; however, a real application would probably perform additional discovery – perhaps looking for the company with the lowest priced widgets.

³⁸ Microsoft's Visual Studio® .NET is one of the available tools for generating the required communications code from WSDL.

³⁹ Discovery is a way to other businesses find out about your business and its web-services, in an automated fashion.

Here is the code:

```
while ( ! masterdirectory.ChangeNotify() ) ;

if ( masterdirectory.ChangeNotify() == -1 ) {
    cout << "the master directory object is currently out-of-service!" << endl;
    exit(1); // a real application would not exit here, but instead retry... maybe using a different server
}

VarArr widgetcompanies = masterdirectory[ "widgets" ]; // get a list of all companies that sell widgets

Var onecompany, onedirectoryname;
onecompany = (widgetcompanies.First()->Key()); // the name of the first company in that list: "Acme Widgets"
onedirectoryname = (widgetcompanies.First()->Value()); // index of the web-services directory for "Acme Widgets"

cout << "name of the first company in our master directory=" << onecompany << endl;
cout << "name of the web-services directory object for that company=" << onedirectoryname << endl;
```

The while-loop at the top of the code block will terminate, once we either have the master directory information, or SmartVariables determines it can not connect to machine *discoveryserver.com* to retrieve that object. The list object *widgetcompanies* is assigned a list of all "widget" companies. The container *onecompany* gets assigned the name of the first entry in that list: "Acme Widgets." The container *onedirectoryname* gets assigned the name of Acme Widget's web-services object. In this case, *onedirectoryname* will have a value of *webdirectory@acmeserver.com*

The sole purpose of the above code was to discover the name: *webdirectory@acmeserver.com*. Once we have that name, we use it to retrieve the actual web-services directory for the company "Acme Widgets." Although the explicit-wait method could be used again, this time, we show how to use an asynchronous callback. That way, our code can go off and do something else while waiting for the directory to arrive. The internal structure of our Acme directory object is arbitrary, just so long as everyone agrees on that structure and the *index names* used beforehand. For the purpose of this example, our directory contains a list of services, plus "white page" and "yellow page" data. Each of these services in the list contains another list describing service invocation parameters and possibly the data *type* (container or list) any service *invoker* should expect for returned results. We are just going to use a simple container for results in this example. Our directory describes any number of services, with each service containing as many command parameters as needed. Here, we store "white page" information and "yellow page" information at the same level as the services-list. All of the services in this example are for the same company, Acme Widgets. This is the code:

```
List<VarArr> directory;
directory.SetObjectChangeNotifyCallback( directorycallback ); // the callback method is directorycallback

// remember: "onedirectoryname" contains the value "webdirectory@acmeserver.com" here.
directory.Name( onedirectoryname, GETVALUENOSUBSCRIBE );

while (1) {
    // next, the program would continue execution and do "whatever" processing it needs to.
    // when the "webdirectory@acmeserver.com" directory arrives, the "directorycallback" will execute;
    // however, for SmartVariables to check the network for incoming messages, an object must be referenced.
    // here, we just construct a dummy container, so that SmartVariables continues to remain active.

    Var dummy;
}
```

In the above code, *directorycallback* is the C++ function we want to attach to our directory object for change-notify processing. When the directory information for Acme Widgets arrives at the invoking computer, that function executes, and receives one passed parameter: a pointer to the directory object. In this example, we extract and print some of the "white page" information, and then use a web-service specification for our "standard" service named "**Order**" to place an online order for some widgets. Let's examine the *directorycallback*.

Below is code for the asynchronous “*directorycallback*” method. A real application would probably do more discovery work before placing any order, but since this is just a simple example, we are going to go ahead and invoke the “*Order*” web-service from inside this callback, directly:

```
void directorycallback ( PStore *directory_ptr )
{
    List<VarArr> directory = *(List<VarArr> *) directory_ptr; // upcast the parameter object
    Var name, address, contacts;

    name = directory[ “white page” ][ “company name” ];
    address = directory[ “white page” ][ “company address” ];
    contacts = directory[ “white page” ][ “company contacts” ];
    cout << “name=” << name << endl;
    cout << “address=” << address << endl;
    cout << “contacts=” << contacts << endl;

    VarArr order = directory[ “Order” ]; // this becomes our service-invoker template
    Var results; // this object will eventually hold Order service-invocation results

    results.Name( “WidgetOrderResult@smartvariables.com” ); // subscribe to future results here

    // next, let’s fill-out the order details:
    order[ “ship from” ] = name; // Acme Widgets
    order[ “ship what” ] = “widgets”;
    order[ “ship name” ] = “SmartVariables.com”;
    order[ “ship address” ] = “550 Battery St., Ste 905”;
    order[ “ship city” ] = “San Francisco”;
    order[ “ship state” ] = “California”;
    order[ “ship zip” ] = “94111”;
    order[ “ship quantity” ] = 1024;
    order[ “confirmation” ] = results.FullName(); // the name of our expected confirmation object

    // and now that we are ready to order... let’s invoke the Order web-service:
    Var service_location = order[ “service invoker” ]; // service_location contains Order@acmeserver.com
    order.MakeServiceInvoker(); // service invoker objects only request from a single server
    order.Name( service_location, SETVALUENOSUBSCRIBE ); // and we just ordered 1024 widgets!

    // next we wait for the confirmation of our widget order – a callback could have been used here, also
    while ( ! results.ChangeNotify() );

    cout << “final result of our online web-service order=” << results << endl;
}
}
```

Notice that the “*directory*” object is indexed just like a doubly indexed C++ array[][] object. The difference being that SmartVariables lists are associative⁴⁰. The first index, “white page,” retrieves the *contained list* holding the Acme Widgets’ company white page information. The second index gets a specified container object from that, using pre-determined index-names for our white pages data. For example, the 2nd index “company name” gets the container object holding the company’s name from its “white page” list. We extract and display that information here, although a real application would not do so, except perhaps during debug phase. If we wanted, we could also print out the entire “*order*” object to examine and discover what Acme Widgets expects for invocation parameters. All companies have previously agreed that the index string “*confirmation*” will be used to tell the service provider the **name** of our invoker’s expected results object. They also agree that a directory’s web-service objects use the special index string “*service invoker*” to hold the **name** of (Acme Widget’s) ordering object. Likewise, the “ship” index names are also prearranged by a set of naming standards. Although the web-service-name “*Order*” might be pre-arranged, that could be discovered. Naming conventions are required for SmartVariables based web-services to function.

By now, you should be able to see how simple this method is; especially when compared to the traditional directory and web-service framework that uses UDDI, XML, WSDL, SOAP and HTTP.

⁴⁰ Associative arrays can index elements using arbitrary character strings, instead of being restricted to just scalar values.

Peer-to-Peer File Sharing

Peer-to-Peer (P2P) is a networking strategy where the different computers across a network are considered to be equal. Instead of a central server providing services for clients on the network, each computer makes its resources available to others. And, each computer uses resources which others have made available. A contrasting strategy is client-server. P2P has been in the news quite a lot recently. Most of us are aware of the controversy⁴¹. SmartVariables, a generic networking and communications technology, could be used to build P2P and B2B⁴² style systems. Our company has tried to avoid this controversial use of networking technology. We feel that SmartVariables is best suited for scientific research systems, such as solving complex N-body⁴³ problems or implementing large-scale distributed neural networks; however, inevitably one day someone will use SmartVariables as the backbone of a P2P system. Just like the C++ compiler is a generic technology that can build any type of system – good or bad, SmartVariables technology represents a generic distributed-data networking technology that allows construction of any kind of distributed information system. The problem is not with the technology, but in the use of that technology.

That said, SmartVariables makes it easy to implement fast, optimized P2P network structures like “parallel index clustering” and also easy to move files around a network. However, SmartVariables is quite *restrictive about the location of files that it can access*. Access is permitted only from 1 directory on the system, named “**pub**,” and located in the same directory as is the SmartVariables database. Only files residing under the “pub” directory can be manipulated or loaded, and files can not be placed anywhere except into the “pub” directory on other systems. Let’s return to our network of three computers: Alice, Bob and Charlie. If we were executing a program on machine “Bob,” and wanted that program to copy a file on Charlie named “*backup.tar*” from Charlie into the “pub” directory on Alice, then the following code would affect that 3-way transfer:

```
Var thefile;

thefile.Name( "backup.tar", DONTGETVALUE ); // don't attempt to read local database for this

thefile.Load( "backup.tar@Charlie" ); // attach to Charlie and begin loading the file "backup.tar" from "pub"

while ( ! thefile.ChangeNotify() ); // wait for "backup.tar" to load from the "pub" directory on Charlie

if ( thefile.ChangeNotify() == -1 ) {
    cout << "SORRY: the file 'backup.tar' on Charlie is currently out-of-service!" << endl;
    exit(1);
}

thefile.Save( "backup.tar@Alice" ); // save "backup.tar" into the "pub" directory on machine Alice
```

As you can see, it is easy to move binary files about the network using SmartVariables containers. Like the *ftp*⁴⁴ protocol, there are many legitimate uses of such file-transfer technology. To locate information, SmartVariables has the ability to *search for objects* that match a name-pattern, location, and a structural type by using the Stack object’s *Find()* method. Find returns a list of object names, in the form of a Stack that matches a regular-expression⁴⁵ pattern. Once you have an object’s name, you can easily get the object. When combined with *Directories* that contain indexes (host names) of appropriate machines for searching, *Find()* becomes very powerful.

⁴¹ Beginning with Napster, and music file sharing, the RIAA has aggressively sought to make P2P file sharing technology illegal.

⁴² Business to Business (B2B) provides information exchange and other services between businesses.

⁴³ The problem of finding, given the initial conditions of *n* bodies, their subsequent motions as determined by classical mechanics.

⁴⁴ File Transfer Protocol (FTP) is commonly used to move files between machines on the Internet.

⁴⁵ A regular expression is a string that describes a whole set of strings, according to certain syntax rules.

Connecting Legacy Systems to the Internet

Many businesses have a large financial investment in legacy C++ systems. Yet, such systems were never originally intended to work over the Internet or in a distributed database environment, and so are often quite costly to rewrite for that. Typically, legacy applications are database management systems running on mainframes or minicomputers. As new technologies, standards and applications emerge, old standards naturally become obsolete. As a result, valuable information assets become trapped inside proprietary systems and old file formats. The challenge organizations face is to get their content into a new environment, and to do so quickly, cost-effectively and preferably without losing data quality or integrity, and without any significant “down” time.

Most 3rd party conversion software and conversion service providers focus on text data. Graphics information remains a problem, especially with technical documentation, publishing, architectural and data management environments. Assets like engineering blueprints can be difficult to extract or reproduce, yet often these are the organization's most powerful knowledge representation. Additionally, legacy software is often extremely difficult to modify for a B2B environment, where businesses need to share selected information. In the past, a system rewrite is generally involved to do that. While many software companies offer conversion services to help business migrate their legacy applications to the Internet or intra-net, such services are often prohibitively expensive, and take time to plan and implement.

Enter SmartVariables. Able to contain and work with any piece of data, including graphics and engineering drawings, and able to intelligently share and save that information, SmartVariables can be “dropped into” legacy systems as a simple way of extracting and/or converting your data. Because only minor modifications are usually involved for a SmartVariables conversion, old legacy systems that might otherwise need rewriting in order to extract database information (or become B2B ready), can now be quickly and easily modified to share their databases with more modern systems. Should data format conversion also be required, that process can easily be offloaded to another system that uses SmartVariables to *subscribe* to original, extracted data. This minimizes any chance of breaking an existing application and eliminates “down” time during development.

Legacy applications can even be modified to provide *Web-Services* for other systems, or to invoke services, as needed. It is possible to do data sharing *between* legacy applications that previously didn't interact, with little effort. In some cases, conversion of legacy software requires only a single day's programming work. The addition of a few well chosen lines of new SmartVariables code to your legacy software is sometimes sufficient to modernize it. For example, if you wanted to publish your database for B2B support, the addition of two lines of new code and one header declaration may be enough. *To illustrate:* at the time of writing a database record, your legacy application could also assign that record into a SmartVariables container object, and then name it with the original database's *key* value. In so doing, you just published your legacy database across the network.

Potential network communications and reliability issues are detected and corrected as they arise. SmartVariables automatically handles many error situations. However, your application still retains final control because objects are duly notified about out-of-service conditions so they can respond appropriately. Because of the simplicity of programming with SmartVariables objects, junior programmers can even carry out much of a legacy conversion task. This saves a lot of money from your IT budget: *Labor, conversion costs and maintenance costs are all lowered, with no immediate need to rewrite applications.*

Instant Messaging, Document Sharing and Spiders

Instant messaging (IM) applications allow people to directly communicate and share files with other IM users in a real-time session similar to a private chat room. Currently, only a few businesses use IM technology as part of their work flow – mostly because they don't want employees wasting time with idle chatter; however, if Instant Messaging were selectively incorporated into your support system, in a tool-to-tool fashion, your business's quality of service would increase. For example, many businesses run *helpdesk software* of one sort or another. It may be desirable to incorporate IM into your helpdesk, so that a subset of employees might more effectively communicate to solve regular work problems. SmartVariables is easily incorporated into existing software and can be used to provide IM capabilities; however, it can do much more than that: To keep productivity high, SmartVariables also provides a simple way for management to *monitor the work activities* of their staff. By modifying your desktop's tools to replicate active work information into SmartVariables objects, management gains the ability to selectively monitor remote activities. Such information could be used to more effectively manage promotions and to identify or assist with training needs.

SmartVariables can be useful in efficiently sharing documents. Imagine two (or more) workgroups in different physical locations. The groups are each assigned different aspects of some project, but they share responsibility for a few of the common documents. Say these documents are stored as PDF⁴⁶ files. Now, if the document *reader* and *creation* software being used were modified to also place all documents inside SmartVariables container objects, then whenever a document changed it would ship itself automatically to everyone in that document's subscription list – basically to everyone in each workgroup. It would not be possible to “forget” to tell someone about changes. Even if one person at one location were actively reading a document while someone at another location was editing it; the person reading the document would be immediately notified when changes were saved at the remote location. This would happen because the document reader software would have been modified to pop-up a window stating that “this document has just changed.” It would then give the reader several options: *view the new document*, *view the changes to the document*, or *ignore the new document*. Such a solution might improve the effectiveness of all the workgroup's communications – thanks to document management with SmartVariables.

Web Spiders are automated programs designed for gathering information on the Internet. Often, this information goes into a search-engine's database. Spiders are totally necessary to maintain cohesiveness of the Internet. Each of us uses Spider results when we access a search-engine. Spider programs are easy to build with SmartVariables, because the container objects can directly load and parse web pages. Either SmartVariables container type can *load pages*; however, when a *named Var* is used, the load operation happens in the background, instead of the foreground. Additionally, *List* objects can be assigned web *link* information: The page gets parsed, extracting that page's HTTP links as element *indexes* and associated page text for element *values*. Here is an example that loads a page, and also shows how to parse out the links and put them into a *List*.

```
Mem webpage;
webpage.Load( "http://www.smartvariables.com", CONTAIN_URL ); // just load the page's HTML source
cout << "webpage HTML source=" << webpage << endl;

VarArr pagelinks;
pagelinks = webpage.Load( "http://www.smartvariables.com", CONTAIN_URL_LINK ); // load and parse the page
cout << "list of HTML links=" << pagelinks << endl;
```

⁴⁶ The Portable Document Format (PDF) is a file format created by Adobe to provide a standard form for published documents.

Conclusion

Most systems today manage data by moving the data in and out of databases - using a "buffer" to transfer data between the database and the program's working variables. SmartVariables removes this "buffer" by attaching the working variables directly to network information storage. When the storage changes, so do the variables... on **all** of the machines with copies of that information. The design of the SmartVariables product is such that distributed databases "push" altered content to machines accessing that - and deliver the data to each process's working variables automatically. By adopting a naming convention and attaching callback methods, it is possible to implement complex Web Services using only simple containers and lists.

SmartVariables technology represents a quantum leap in distributed computing that is anticipated as having major impacts on the design and implementation of networked systems. SmartVariables take the important middle ground in distributed object management, not trying to define your application's framework or its components, yet still providing the high-level building blocks required for constructing fantastically complex distributed object systems. Corporations using our platform will enjoy business-logic simplification and portability/configurability benefits that have not been previously available when building distributed, network-persistent object systems. SmartVariables is the simplest parallel-processing platform and object-oriented database product in the world. By using SmartVariables, the developer does *not need to care* how to do low-level database programming or network programming; it is natural and easy for applications and businesses to efficiently share complex information across a heterogeneous network of any size.

Junior programmers can quickly learn the environment, due to design simplicity and a functional similarity with basic and familiar programming constructs. SmartVariables technology supplements intrinsic data types with type-less, intelligent, self persistent, *generic container and list* objects. The generic container naturally manipulates any piece of data, including binary information. Data-arrays converted to a SmartVariables list transform into self persistent associative arrays of generic, transparently shared objects. Seamless integration with intrinsic variables and arrays make the product immediately familiar to use. Advanced behaviors, like simulating instantiation in persistent network-shared memory and infinite-precision calculations make the product immensely flexible. SmartVariables transparently propagate changed object-values and array elements across the network and into select running processes and databases. The data "*knows*" where copies exist. So new data values go there and appear, like magic, inside running programs and the database. Natural syntax for "event" processing means implementing Directory and Web services is simple.

For the network-programmer or the database-programmer, these utility objects are of amazing convenience. For the business manager, simplification translates into lowered development, maintenance and training costs, reduced time to market with higher quality products, and legacy systems that are modernized without being rewritten. This new technology has much to offer.

Download at www.smartvariables.com