A Better Middleware

Eugene Nelson

Copyright 2017 by Eugene Nelson

All rights reserved, including the right to reproduce this book or portions thereof in any form whatsoever. For information regarding permissions, send email to SoftEcoSDK@gmail.com.

Preface

Existing software research and products are mostly about how software works on a single computer. The Internet and Web enable two computers to communicate messages and access content. The IaaS, PaaS and SaaS models and protocols map the concept of an application running on a single computer onto a cloud of resources that are leased as needed. Middleware such as CORBA and ODBC are alternative methods for how a program accesses functionality on another computer. The Service Availability Forum describes real time message distribution within an embedded system of multiple computers. Very little research or available solutions are about how programs on multiple computers work together as one system.

The potential advantage of a system composed of multiple computers and programs working together can be understood via an analogy. A business starts out as one person doing work for one customer. As a business grows, multiple people perform the same work. The work for one task can even be handed off to another person as needed. A larger business can implement internal security measures. Intermediate work output can be stored at the end of a work day within a locked desk, locked office, secure document room or safety deposit box within a vault. Procedures can be implemented so that multiple people can work on the same task to achieve both reliability and security.

The current boundary between an application and operating system is devised to maximize performance at minimal cost for a single computer. Reliability and security are provided by an operating system with minimal help from an application. The knowledge needed to recover in progress work after a computer fails can be known by an application but is not shared with an operating system. The knowledge needed to know the identity and authority of remote programs can be known by system software but is not shared with a program performing work for a remote program.

Cybersecurity efforts mainly focus on how to improve the situation with existing software. Much better system security is possible if programs and system software share knowledge of in progress work and authority of remote programs. The goal is not optimal performance of a single computer. Extra work is performed by each computer. A system can reliably and securely perform work that is then scaled across many computers.

A new middleware must be devised such that a middleware helps a program to do its part for system reliability and security. The purpose of this book is to describe the what, why and how for one such middleware. An exploratory project has completed a basic design, implementation and test of a possible middleware. The design supports multiple computers within a system. The implementation maps the design onto one computer for easy testing. The implementation is some 300 classes and more than 85,000 lines of source written in Java. A multiple computer implementation is close to completion but requires interested parties to help to complete the implementation and testing. Much work remains to make this an off the shelf product with standardized program interfaces. Note that the design allows an easy mapping of the implementation into other software languages such as C++ and C#.

The software community needs to realize that more can and should be done to address the cybersecurity crisis. A new middleware must be part of this effort. New client and server programs need to do their part in improving reliability and security. Please ask yourself what you can do to join and advance this effort.

Existing programs and middleware will not go away. They will keep having the same purpose and value that they have today. New programs will be written to use a new middleware to achieve better system reliability and security.

Please send questions and suggestions for improvement to “SoftEcoSDK@gmail.com”.

Table of Contents

1 INTRODUCTION 6

1.1 Data Format Conversion Layer 6
1.2 Message Communication Layer 7
1.3 Client to Server Communication Layer 7
1.4 Infrastructure Checklist 8

2 SYSTEM CHARACTERISTICS 10

3 DATA FORMAT CONVERSION LAYER 11

3.1 Color as a Class 12
3.2 Color as a Convertible Data Class 13

4 MESSAGE COMMUNICATION LAYER 17

4.1 Message Distribution Algorithm 17
4.2 Backup Main Hub Instance 18
4.3 Program Definition 18
4.4 Program Copy 18
4.5 Embedded SCADA 18
4.6 System Configuration 19
4.6.1 eventConfig sample 19
4.7 Sample Message Publisher and Subscriber 20

5 CLIENT TO SERVER COMMUNICATION LAYER 26

5.1 Sample Service Client Session 27
5.2 Sample Server Session 32

6 CHARACTERISTICS OF A BETTER MIDDLEWARE 38

7 ANALYSIS OF A BETTER MIDDLEWARE 39

8 EMBEDDED ENVIRONMENT SUPPORT 40

1 Introduction

A better middleware is needed to survive the current and evolving cybersecurity situation. Better security goes hand in hand with better reliability. Therefore, new programs need to achieve both better reliability and security through the help of a new middleware. This book describes one possibility for a better middleware.

New programs, using a better middleware, can be added to an existing system. This allows migration of system functions to achieve improved characteristics as needed.

The presented middleware is composed of three layers. A data format conversion layer, message communication layer, and a client to server communication layer work together to pass data between programs. The design of these layers was performed in a bottom up manner. This ensures that a layer offers only functionality that can be reasonably accomplished. The following describes the essential usage, workings and advantages of each of the three layers.

The purpose of this book is to attract interest in a better middleware. The basic concepts are explained and some of a possible interface is demonstrated. A programmer and administrator guide book provides a more complete documentation of this possible interface.

1.1 Data Format Conversion Layer

A data conversion layer is needed for reasons such as allowing a system to be composed of different computer types. It helps to eliminate most of the programmer effort needed to convert data between formats. A programmer defines a data class in a new manner to create a convertible data class. Each data field has programmer initialized meta data. A program calls the layer to convert the content of a convertible data class instance into another format. Currently a serial format and a human readable text format (subset of XML) are supported. A program can call the layer to reverse a conversion by supplying an empty instance of the original class and the serial or text result of a conversion from a class.

One advantage of a convertible data class is that it can help to exchange a message between programs. A sending program creates a message by inserting field values into a convertible data class. A message is then converted to serial. A serial message is communicated to a receiving program where it is converted back into a convertible data class. A receiving program accesses field values of a message from a convertible data class. This is like passing a class instance between programs except that a programmer has complete control over which field values are passed. The sending and receiving convertible data classes must have the same meta data for passed field values but can differ for a field that is not passed. A notable benefit is that a sending and receiving convertible data class can be written in different software languages.

Another advantage of a convertible data class is that it can help a program to obtain its configuration information. The current approach, where configuration is contained in a text file, requires a fair amount of programming effort. The data format conversion layer can be used to eliminate most of this programming effort. A programmer starts by defining a convertible data class or classes to hold configuration information. A program calls the layer to read a binary file containing a serial format of configuration information. The serial is read from a file and converted into a convertible data class. A program accesses field values of configuration from a convertible data class.

A file containing a serial format of configuration information can be created in different ways. There is an administrator utility to convert a file containing a text format into a file containing serial format. This is a general-purpose utility as it uses a class name of a convertible data class to instantiate a class that performs a conversion between a class instance, serial and text formats. An administrator creates configuration in an appropriate text format file and uses the utility to convert the text file into a serial file. Of course, a configuration specific utility can be designed and implemented to internally use a convertible data class that reads and writes a configuration file in serial format.

Yet another advantage of a convertible data class is that it can help to convert a message into text that is appended to a program created trace file. A program can use an environment that includes a trace capability. A program calls trace to determine whether a specific kind of trace, as identified by a program specific keyword, is enabled by an administrator. If a specific trace is enabled, then a program calls trace with a convertible data class containing a message. Trace converts the content of the convertible data class into text and appends the text to the trace file.

1.2 Message Communication Layer

A message communication layer appears to a programmer like a typical message oriented middleware. A dedicated program to program message channel can be used to pass a message in either direction. A publisher program can generate a message associated with a unique event identifier that is distributed to listener programs interested in a type of event.

A message communication layer includes a security feature to only allow messages to pass between programs as configured by an administrator. There is a reliability feature to survive the failure of a communication path by sending each message via two paths. The layer is composed of several components. A main hub, local hub and message bus components are directly involved in passing a message and embedding security and reliability features.

One local hub instance resides on each computer. A local hub initializes by connecting to two different main hubs. A message bus component resides with a program and makes a connection to a local hub. A message is sent from a program via a message bus to a local hub. A local hub sends the message to each main hub. A main hub distributes a message to a local hub destination(s). A local hub ignores any duplicate message and distributes a message to a message bus destination(s). A message bus gives a message to a receiving program.

A main hub resides on a dedicated computer. It only allows messages to pass between computers (and ultimately programs) as configured by an administrator. A main hub program resides on a neutral computer to minimize interaction with other computers within a system. Message content is less likely to be compromised with this form of isolation.

Each original and received message is held within a convertible data class. Message data content is converted to serial, passes to a destination, and is converted back into a convertible data class at a destination.

Advantages of this mechanism include improved reliability and security while maintaining good performance. A published message consisting of a floating-point or integer value with type of event, unique identity, time of collection, etc. can be serialized within only 25 bytes or 200 bits. A message is packed into a super message during transit. A main hub on a gigabit network can distribute up to a million small messages per second depending on system message traffic characteristics. Latency of a published message is two network hops of a few milliseconds.

1.3 Client to Server Communication Layer

A client to server communication layer allows a client to access functions from a server, a server to limit client access to only permitted functions, and recovery of in progress work after the failure of a server or client. The two latter features require a new type of client to server mechanism. A client and server program must be written in a new manner.

The client to server communication layer is composed of two components. One component, called an executive, resides with a client or server. The executive provides an environment for a client to access server functions and for the server to communicate with the client. The other component is a layer manager that resides on an isolated computer. An executive instance uses the message communication layer to create a message channel to an active layer manager instance.

A programmer defines a session to hold the state of in progress work. The session also holds minimal information needed to recover a session in the event of a program failure.

A service is written by a server programmer to reside with a client program. A service provides an API to access server functions and to hide client to server communication details. Session recovery information and any message is passed to a layer manager where it is saved and a message is forwarded to its destination. Full details are described in a following section.

An advantage of this client to server mechanism is that work of either a client or server can be recovered after a program failure. Any external user, or the other program in the client to server association, is not affected except for observing a slower response when a failure and recovery scenario occurs.

Another advantage is that a server session contains what work a client is permitted to request per administrator configuration. A server can validate a request against permissions before performing work. A server reports any illegal request.

An initial disadvantage is that it takes some time and effort before a programmer understands and can write a client or server program in an appropriate pattern. Over time this becomes an advantage as writing a client or server program is less effort when following the same design pattern for every client, service and server.

1.4 Infrastructure Checklist

Literally thousands of ideas are woven together like threads within the 300+ classes that operate within some dozen programs. Together the programs create one version of a system communication infrastructure. System specific programs perform work and communicate with each other as a system via an infrastructure version. Some of the more important ideas are:

Message Communication Layer

1. Reliably, Securely & Efficiently distribute messages between programs on multiple computers
2. Do not talk to strangers
3. Message travels two paths
4. Point to point message conversation between two instances of same program
5. Point to point message conversation between parent and child e.g. client to server mechanism
6. Publish and subscribe event message distribution per type of event message
7. Multiple instances of publisher while only one active
8. TCP/IP connection per admin, event & data message stream
9. Pack & unpack message within fixed sized super message
10. Disassemble & assemble data message larger than super message
11. Maximum size data message defined by system
12. Admin or event message must fit within one super message
13. Embed format conversion between class and serial for 4 specific types of event format
14. Convert between event class content, method parameters and serial residing within super message for maximum performance i.e. communication media and event size in serial are limits on events per second
15. Multiplex message onto appropriate super message stream
16. Flow control on super message
17. Keep one partial super message in transmission to minimize latency
18. Only transmit extra super message when full
19. Embed Automated Control System i.e. monitor communication state & assign program instance as active
20. Record & replay last event message for a starting subscriber
21. Failure of both paths forces complete communication shutdown
22. Automatic startup of paths and communicating program instances per configuration
23. Configuration of production version frozen to be the same on all computers
24. Each version has unique TCP/IP port assignments
25. Several versions can execute on same set of computers with minimal interference
26. Each local hub computer has unique encryption key used with communication to main hub
27. Multiple main hub instances per A & B path communicate and decide which becomes active

Client to Server Layer

- Reliably, Securely & Efficiently distribute work between programs on multiple computers
- Reliability
  1. In progress work restart information check pointed with each communication and upon demand
  2. Client determines its part of restart information
  3. Server determines its part of restart information
  4. Client to server mechanism adds messages being exchanged to restart information
  5. Client and server recovers in progress work from restart information
  6. Either client or server part of communication can be recovered with other participant only detecting an extra delay in communication time
  7. Checkpoint restart information to isolated infrastructure computer that saves information and forwards message between client and server
  8. Backup instance of infrastructure computer holds restart information
  9. Multiple instances of one type of server exchange information used to perform work
- Security
  1. Identity of Client and Server securely known
  2. Client to server mechanism limits client to allowed server
  3. Server is informed of what work client can request

4. Server restricts client to allowed work

- Efficiency
  1. Use message communication layer
  2. Multiple work sessions per client and server program instance
  3. Multiple instances of server can perform work for clients
  4. Server can be client of different server
  5. Service API translates between client method call and server message

Data Format Conversion Layer

1. Reliably & Efficiently convert data content between two formats
2. Infrastructure supports definition of convertible data class
3. System specific source defines convertible data class
4. Support class, serial and human readable text formats
5. Convertible data class has embedded meta data per attribute to direct conversion
6. Meta data consumes about 10 times extra memory space
7. Statements to compiler could support explicit meta data with minimal extra memory space
8. Convertible data class has source to explicitly initialize meta data
9. Most attribute data types supported except for hash table and class pointer
10. Convertible data class can hold list of convertible data class instances
11. Convertible data class can hold hierarchy of convertible data class instances
12. Convertible data class can hold non-convertible attributes such as hash table and class pointer
13. Serial and text formats contain identity of originating convertible data class
14. Serial and text formats omit attributes with a value equal to a default value
15. Serial format identifies data type for error detection
16. Infrastructure admin program converts file containing text to or from file containing serial
17. Trace method within Infrastructure converts class to text and appends to trace file
18. Message layer and Client to Server mechanism use data format conversion for configuration
19. Message layer and Client to Server mechanism use data format conversion to convert between local data class instance and data or message exchanged between programs
20. Message layer and Client to Server mechanism trace message flow per configuration

2 System Characteristics

A computer system is physically composed of computers and networks. The purpose of a system is to perform work for an external user and to optionally monitor and control an external process. Work can include data collection, persistence and modification as well as computation and action. System characteristics include performance, reliability and security.

An external user can observe and judge a system based on some characteristics. A user views performance as to how quickly a system shows requested data or performs requested work. A user views reliability as the combination of how often a system is available to perform work, if data is persisted as needed and expected, and if a system performs work in a consistent and needed manner. A user views security as the combination of only permitting data modification or actions by an authorized user and keeping sensitive information private.

Measured availability of a system can be improved by:

- Use hardware components with a higher MTBF (mean time between failure) rate.
- Supplement commercial power with backup batteries and a power generator.
- Shift work to an alternate computer during a computer failure scenario.
- Shift communication to an alternate network component during a network component failure.

Security of a system includes:

- An externally accessible program identifies a user via supplied credentials. A user can only perform permitted work as configured by an administrator.
- A network firewall only allows an external entity to communicate to an internal program as configured by an administrator.
- Sensitive data is encrypted during transit through a public network.
- A scan of persisted data or in memory programs can detect known virus instruction patterns.
- A scan of historical events and actions can detect abnormal usage patterns.

The above availability and security features are available on any system regardless of the middleware that is used. A better middleware adds many more features described and summarized by the following sections.

3 Data Format Conversion Layer

A convertible data class is defined by extending a convenience class called gcData. This base class manages a list of fields defined within a convertible data class. There is a convenience class for each kind of field such as an integer which is called gcIntegerValue. For example, Figure 1 - Field with meta data, shows an integer field called age with a current age value of 42 and initialized meta data.

Value	42
Name	Age
Numeric Id	3
Data Type	4
Default Value	18

Figure 1 - Field with meta data

This age field, contained in a convertible data class, can be converted to the serial format shown in Figure 2 - Serial format of Age Field by calling the data conversion layer. The serial format contains the numeric id, data type, size in bytes of a value, and a value of 42 which is hexadecimal 2A. The conversion algorithm of a field within a class into serial is clear.

0403012A

Figure 2 - Serial format of Age Field

The original convertible data class content can be restored by combining the serial format with an instance of the convertible data class. The conversion algorithm uses the numeric id of 3 from the serial format to locate the field with meta data. The data type 4, meaning integer, is compared between the serial format and meta data to detect an unintentional error. The size in bytes of a value shown as 1, is used to know how many serial bytes is used to specify a value. The byte of value 2A is converted into a proper sized integer of decimal 42.

This age field, contained in a convertible data class, can be converted to the text format shown in Figure 3 - Text format of Age Field by calling the data conversion layer. The tag is taken from the meta data for name. The conversion algorithm converts to a subset of XML.

<Age>42</Age>

Figure 3 - Text format of Age Field

The original convertible data class content can be restored by combining the text format with an instance of the convertible data class. The conversion algorithm uses the tag of Age from the text format to locate the field with meta data. The data type of integer from the meta data is used to pick a conversion of text into an internal integer. The text 42 is converted to an internal integer of decimal 42.

A complete listing of convenience classes is specified by a programmer reference guide. There are seven for the usual types of primitive data and two more for an enumeration and time stamp. The enumeration is quite useful to equate an internal integer value to an external text name. For example, a value of 0 is represented as “false” and a value of 1 as “true” within a text format. There are nine more convenience classes for arrays of the primitive data types.

There is a convenience class to hold the serial output of a data conversion. This is quite useful to wrap a message around another message already converted to serial. Finally, there is a convenience class for an array of gcData class instances.

A convertible data class can contain another convertible data class or a list of convertible data classes. The conversion algorithm embeds the name of each convertible data class into the serial or text format. The conversion algorithm uses the name embedded in the serial or text format to locate a class used to convert data content back into a hierarchy of convertible data class instances.

Complex types such as a class pointer or a hash map are not supported. This kind of information can be built from more primitive types of data after a serial or text format is converted into convertible class instances.

Note that the interfaces in the implementation are not final. Names such as gcData need to be reviewed through a standard making process.

3.1 Color as a Class

The following is an example class called uiColorSpec. The class holds the specification of a color as the amount of the combined primary colors red, green, and blue along with a factor to specify transparency. This class will be re-specified later as a convertible data class that includes meta data to control data format transformation.

```
001 package nelson.eugene.serviceFrame.services.ui;
```

Line 1 specifies a package so that this class can be used by another class outside of the package. Line 2 shows how a class begins. Lines 4 thru 7 show attributes held as an integer.

```
002 public class uiColorSpec
003 {
004   private int m_r;
005   private int m_g;
006   private int m_b;
007   private int m_a;
008
009
010 public uiColorSpec()
011 {
012 }
013
014
```

The initDefault method initializes the data class to default values.

```
015 public void initDefault()
016 {
017   return init(0, 0, 0, 255);
018 }
019
020
```

The init method sets all attribute values at one time.

```
021 public void init(int red, int green, int blue, int alpha)
022 {
023   m_r = red;
024   m_g = green;
025   m_b = blue;
026   m_a = alpha;
027 }
028
029
030 public int getRed()
031 {
032   return m_r;
033 }
034
```

```
035
036 public int getGreen()
037 {
038   return m_g;
039 }
040
041
042 public int getBlue()
043 {
044   return m_b;
045 }
046
047
048 public int getAlpha()
049 {
050   return m_a;
051 }
052
053
054 public void ensureValid()
055 {
056   if ((m_r < 0)
057       || (m_r > 255)
058       || (m_g < 0)
059       || (m_g > 255)
060       || (m_b < 0)
061       || (m_b > 255)
062       || (m_a < 0)
063       || (m_a > 255) ) initDefault();
064 }
065
066
067
068 } //uiColorSpec
```

3.2 Color as a Convertible Data Class

The class uiColorSpec is specified as a convertible data class with the following source:

```
001 //Copyright Eugene Nelson Software 2012 All Rights Reserved
002 //Reproduction or Other Use Only By Permission of Authors
003
004 // uiColorSpec
005
006 package nelson.eugene.serviceFrame.services.ui;
007
008 import nelson.eugene.serviceFrame.gc.*;
```

Line 8 refers to all classes in a package so that they can be used by this class.

```
009
010
```

The next line shows class uiColorSpec as an extension of the convenience class gcData. Lines 13 thru 16 show attributes held within a gcIntegerValue convenience class.

```
011 public class uiColorSpec extends gcData
012 {
```

```
013   private gcIntegerValue m_r;
014   private gcIntegerValue m_g;
015   private gcIntegerValue m_b;
016   private gcIntegerValue m_a;
017
018
019 public uiColorSpec()
020 {
021 }
022
023
```

The initDefault method initializes the data class to default values.

```
024 public gcErrorResult initDefault()
025 {
026   return init((uiColorSpec)null);
027 }
028
029
```

The init method sets up programmer defined meta data that is later used to convert between data formats. It optionally obtains meta data from a previously defined class instance. If an error occurs then the error is returned in class gcErrorResult.

```
030 public gcErrorResult init(uiColorSpec baseData)
031 {
032   gcErrorResult errorInfo = null;
```

The next line calls initSchema with a schemaName of sfUiColorSpec and an external name of uiColorSpec. If a convertible data class is embedded within another convertible data class then the external name is typically overridden e.g. foreground replaces uiColorSpec as the external name. The schemaName is used by the serial encoder and decoder algorithm to identify the proper class needed for decoding.

```
033   initSchema("uiColorSpec", "sfUiColorSpec");
034
```

The next line calls method createIntegerValue with an external name of red, a default value of 0, and a unique attribute id of 1. If the id is not unique then the container is not created and an error is returned. A gcIntegerValue convenience class is created, the Meta data is saved, the value is set to the default, and the class is added to the list held within the gcData class.

```
035   m_r = createIntegerValue("red", 0, 1);
036   if (m_r == null) return idErr("uiColorSpec", 1);
037
038   m_g = createIntegerValue("green", 0, 2);
039   if (m_g == null) return idErr("uiColorSpec", 2);
040
041   m_b = createIntegerValue("blue", 0, 3);
042   if (m_b == null) return idErr("uiColorSpec", 3);
043
044   m_a = createIntegerValue("alpha", 255, 4);
045   if (m_a == null) return idErr("uiColorSpec", 4);
046
047
048 return null;
049 }
050
```

The init method sets all values at one time. The setValue method of a value convenience class is called to store the new value.

```
051 public void init(int red, int green, int blue, int alpha)
052 {
053   m_r.setValue(red);
054   m_g.setValue(green);
```

```
055   m_b.setValue(blue);
056   m_a.setValue(alpha);
057 }
058
059
```

The getValue method of the value convenience class is called to get a stored value.

```
060 public int getRed()
061 {
062   return m_r.getValue();
063 }
064
065
066 public int getGreen()
067 {
068   return m_g.getValue();
069 }
070
071
072 public int getBlue()
073 {
074   return m_b.getValue();
075 }
076
077
078 public int getAlpha()
079 {
080   return m_a.getValue();
081 }
082
083
084 public void ensureValid()
085 {
086   if ((m_r.getValue() < 0)
087       || (m_r.getValue() > 255)
088       || (m_g.getValue() < 0)
089       || (m_g.getValue() > 255)
090       || (m_b.getValue() < 0)
091       || (m_b.getValue() > 255)
092       || (m_a.getValue() < 0)
093       || (m_a.getValue() > 255) ) reset();
094 }
095
096
```

The newInstance method is mandated by class gcData. It creates a new data class instance with meta data based on this class instance.

```
097 public gcData newInstance()
098 {
099   uiColorSpec newInst = new uiColorSpec();
100   newInst.init(this);
101 //Ignore duplicate id error as initial creator will report
102   return newInst;
103 }
104
105 } //uiColorSpec
```

The human readable text for a uiColorSpec embedded within another class looks like:

```
<foreground>
<dataSchemaName>sfUiColorSpec</dataSchemaName>
<red>0</red>
<green>0</green>
<blue>0</blue>
<alpha>255</alpha>
</foreground>
```

Note that the name foreground appears rather than the name uiColorSpec. Also, lines 33 thru 44 of the data class specify the external names that appear within the text. The serial is a sequence of binary bytes that is quite unreadable so no example is shown. The only notable feature is that the string "sfUiColorSpec" appears encoded in UTF8 format.

The programmer and administrator reference guide contains much more information about how a convertible data class can be used as shown by examples of middleware usage.

4 Message Communication Layer

The message communication layer uses a TCP/IP connection to communicate between a message bus and local hub as well as a local hub and main hub. The Figure 4 - Message Passing shows how these components logically communicate via two independent networks.

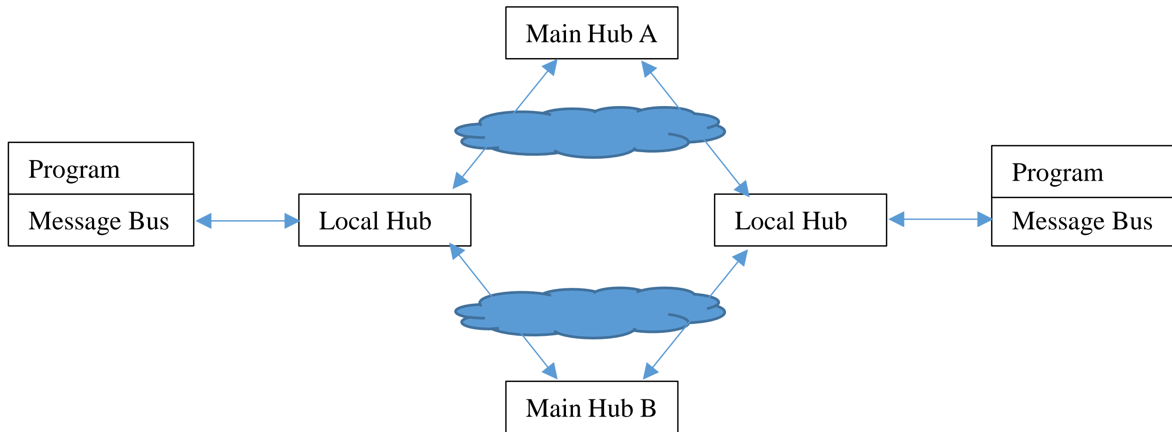

Figure 4 - Message Passing

The message communication layer works like most software that multiplexes a message from a virtual connection over a TCP/IP connection. A message is packed into and unpacked out of a fixed size super message that passes between a message bus and local hub. Similarly, a message is packed into and unpacked out of a fixed size super message that passes between a local hub and main hub. If a message is larger than the super message size then it is disassembled and reassembled at a destination. The layer sends a super message when it becomes full and a flow control window is open. If a flow control window becomes closed and there is no more space in the pre-allocated super message buffers, then a program can choose to be either blocked or be informed that a message is not queued and must be sent in the future. The layer immediately sends a super message containing only one message if there is no other message currently in transit. This mechanism minimizes latency while automatically adjusting to minimize the resources used for a changing message work load.

4.1 Message Distribution Algorithm

A message distribution algorithm must solve several difficult and conflicting issues. It must reliably distribute a broadcast message to all interested subscribers. It must efficiently identify a computer, program, type of event, etc. when a message is transmitted between computers. It must allow changes to configuration such as new programs and computers. It must track changes in the state of communication to a computer and communication to a program instance on a computer. It must help to determine the state of a program and choose which instance of a program is active.

Several different message distribution algorithms have been tried before finding one that appears to work in all scenarios. The current algorithm uses the concept of a system version to reliably achieve expected results.

An administrator creates the configuration for a system version as a small set of configuration files. The configuration has a state that an administrator migrates with the values of CONFIGRATION, TEST, MAIN, and DEATH. The configuration content can only be changed during CONFIGURATION state. Migration is allowed between TEST and CONFIGURATION. Once the state migrates to MAIN, which means it is the default operational version on all computers within a system, then it can only migrate to DEATH.

A transition from CONFIGRATION is significant in that internally unique integer keys are generated to identify a computer, program, event type, etc. The keys are assigned as monotonically increasing integer values based on the position of a definition in a configuration file. Configuration files can then be distributed to all computers in a system. Each computer has the identical configuration files for a version. This allows main hub, local hub and message bus to efficiently know

program, computer, etc. identity when a message is sent between computers. Main hub and local hub can calculate message distribution routing only once from a version configuration.

A new system version is created when a change to configuration is needed. Each version must use a unique network address for a local hub and main hub. This means that multiple versions can run at the same time on the same computers. A new version can be tested at any time while the current version remains the default operational version. An older version can be accessed to perform a function that is unavailable or does not work properly on the current default version.

A program instance and message remains inside the boundary of a version. The file system of an operating system can be used to access state information from a prior version upon a first time start of a new version.

A careful examination of the configuration file convertible data class definitions will reveal field values that do not exist in the text and serial formats. For example, there are class pointers that become initialized as the message distribution algorithm calculates message destination. The same convertible data class holds both configuration values and values only needed by the main hub and local hub programs. The main hub and local hub setup algorithms are greatly simplified but a programmer examining the source must be aware of this feature.

4.2 Backup Main Hub Instance

Multiple instances of a main hub can be configured on different computers as the center of a message distribution path. Each instance connects to the other instances to pick an active instance. The active instance allows a connection from a local hub. This feature improves main hub availability.

4.3 Program Definition

A program and associated characteristics must be configured by an administrator. A program must also be written to perform configured characteristics. It might appear to eliminate mistakes and make administration easier by having a program announce its properties but this approach allows a virus to take over a system.

A program is defined with a name and a Java class reference. The current implementation only supports a Java class as a program but it is anticipated that other software language classes will become supported.

4.4 Program Copy

A program is defined to execute and be administered as a program copy. A program copy is a configuration and program instance based on a configuration executing on one or more computers. A message channel can be created by an instance of a program copy to an active instance of the same program copy. Alternatively, a parent program is specified to identify the destination of a created message channel. Only connecting to a peer of yourself or an active parent program is a security feature that ensures a message is only exchanged with a trusted program instance.

4.5 Embedded SCADA

The purpose of a program copy is to allow instances of the identical program and configuration to be available on multiple computers and to use the message communication layer in an appropriate manner. A program copy instance is called a bus program instance. The message communication layer has an embedded SCADA feature to track, manage and report the state of significant system components such as a bus program instance. The communication state of an instance can be DOWN, CONNECTED, STANDBY, or ACTIVE.

A bus program instance starts in a down state. A program initializes its configuration and starts access to the message communication layer. A message bus connection to a local hub changes the state to connected. A program initializes access to event messages and receives the last message distributed for a type of event. A layer program called recorder remembers the last message of each event type and sends a copy to a starting subscriber. Upon finishing subscriber setup, the state transitions to standby. A layer program called assigner tracks state and chooses an active instance among instances of a program copy. If there is no active instance, then an assigner chooses a standby instance to become active. The instance chosen

to be active is informed of this designation. If the instance is a publisher then it is now allowed to publish a message for a type of event allowed by administrator configuration.

4.6 System Configuration

An administrator must configure all elements and associated properties within a system version configuration file such as computers dedicated to a main hub program, computers with a local hub program, where to find executables, programs, and program copies. There are properties of the message communication layer such as size and number of super message buffers.

There is a startup program that always runs on every computer that is part of a system. It looks for new or removed system version configuration files. Elements on a computer are started as programs or stopped based on the configuration file contents.

The following is a fragment of a system configuration file taken from an example shown in the administration guide book. Each possible program that can execute is specified by a programDefinition entry which follows the format:

```
<programDefinition>
<dataSchemaName>sfPrgDef</dataSchemaName>
<name></name>
<programType>UNKNOWN</programType>
<javaClassReference></javaClassReference>
<executableReference></executableReference>
</programDefinition>
```

Each program copy that executes is specified by a programCopyDefinition entry such as:

```
<programCopyDefinition>
<dataSchemaName>sfPrgCpyDef</dataSchemaName>
<programName>sfEventTest</programName>
<programCopyId>A</programCopyId>
<localHubNames>
<value>localHub1</value>
</localHubNames>
</programCopyDefinition>

<programCopyDefinition>
<dataSchemaName>sfPrgCpyDef</dataSchemaName>
<programName>sfEventTest</programName>
<programCopyId>B</programCopyId>
<localHubNames>
<value>localHub2</value>
</localHubNames>
</programCopyDefinition>
```

The dataSchemaName is the name of a convertible data class used to define this format of data. The programName must match a name for a programDefinition entry. The programCopyId identifies how a program can be executed for more than one purpose. For example, A executes on one computer while B executes on a different computer. The list of localHub names identifies each computer that executes an instance of a program copy. At least one instance becomes active and more than one instance can become active depending on the situation. Note that programDefinition and programCopyDefinition are names of two convertible data classes that are used to define and easily access configuration information.

4.6.1 eventConfig sample

The programCopy sfEventTest copy A publishes an event that is allowed by the following:

```
<eventDefinition>
<dataSchemaName>sfEvtDef</dataSchemaName>
<eventId>/eventTest/mf1</eventId>
```

```
<eventType>FLOAT</eventType>
<eventSchemaName>sfFltValSt</eventSchemaName>
<saveType>LAST</saveType>
</eventDefinition>
```

The eventId can be any text string that is unique. The eventId /eventTest/mf1 identifies an event state that contains a floating point number. The programCopy sfEventTest copy B subscribes to an event that is allowed by the following:

```
<subscribeEventIds>
<value>/eventTest/mf1</value>
</subscribeEventIds>
```

which is the event being published by copy A.

4.7 Sample Message Publisher and Subscriber

The following source is a bus program called sfEventTest along with a class called comm. A comm class instance uses the message bus component to either publish or subscribe to an event message. This example uses convertible data classes and interfaces that are documented in the programmer guide book. Hopefully, this example is useful without knowing the exact details of these classes and interfaces.

```
001 //Copyright Eugene Nelson Software 2011 All Rights Reserved
002 //Reproduction or Other Use Only By Permission of Authors
003
004 // sfEventTest
005
006 package nelson.eugene.serviceFrame.fred.samples;
007
008 import nelson.eugene.serviceFrame.net.io.netMain;
009
010 import nelson.eugene.serviceFrame.bus.*;
011
012
013 /* class sfEventTest
014 *
015 * Test fred event feature
016 *
017 */
```

A bus program, by definition, must extend class netProgramBase. Method startFromCommand() of netProgramBase obtains information passed from the infrastructure sfStart program and sets up the environment for execution of a bus program. Infrastructure class netIo method doWork() contains a select that awaits message I/O or a maximum sleep time.

```
018 public class sfEventTest extends netProgramBase
019 {
020 //attribute
021 private long m_priorNanoTime = 0;
022
023 private comm m_comm = null;
024
025
026 /* doWork - look for work
027 *
028 * @param nanoTime - current nanoTime
```

```
029 *
030 * @return none
031 * @throws none
032 */
033 public void doWork(long nanoTime)
034 {
035   if (m_comm == null)
036   {
037     m_comm = new comm();
038     m_comm.init(this);
039   }
040   m_comm.doWork(nanoTime);
041 }
042
043
044 public String getProgramName()
045 {
046   return new String("sfEventTest");
047 }
048
049
050 /* main - start program from command parameters
051 *
052 * @param args - parameters from command line
053 *
054 * @return none
055 * @throws none
056 */
057 public static void main(String[] args)
058 {
059   sfEventTest tst = new sfEventTest();
060
061   netMain netIo = tst.startFromCommand(args);
062   if (netIo != null)
100
063   {
064     //endless loop to perform work
065     long nanoTime = 0;
066     while (true)
067     {
068       //Call programs to do work
069       nanoTime = System.nanoTime();
070       tst.doWork(nanoTime);
071
072       //Call netIo to process messages
073       netIo.doWork();
074     }
075   }
076
077 } //main
078
079 } //sfEventTest
```

Every bus program must have a class very similar to sfEventTest. Class comm extends commAccessBase and implements valueNotifyInterface. Infrastructure class commAccessBase implements busInterface. It executes within a bus program and communicates with other parts of the infrastructure to exchange messages between bus programs. The infrastructure

valueNotifyInterface defines method onValueChange() that informs when the contents of an event class have been changed. The source for class comm is:

```
001 //Copyright Eugene Nelson Software 2011 All Rights Reserved
002 //Reproduction or Other Use Only By Permission of Authors
003
004 // comm
005
006 package nelson.eugene.serviceFrame.fred.samples;
007
008 import java.util.Date;
009
010
011 import nelson.eugene.serviceFrame.gc.*;
012
013 import nelson.eugene.serviceFrame.core.*;
014
015 import nelson.eugene.serviceFrame.bus.*;
016
017 import nelson.eugene.serviceFrame.fred.access.commAccessBase;
018
019
020 /* class comm
021 *
022 * Communication via fred
023 *
024 */
025 public class comm extends commAccessBase implements valueNotifyInterface
026 {
027 //attribute
028 private netProgramBase m_prg;
029
030 private floatValueState m_pFloat1;
031
032 private floatValueState m_sFloat1;
033
034 private boolean m_update = false;
035 private long m_publishNanoTime = 0;
036 private long m_checkNanoTime = 0;
037 private float m_priorValue = (float) 1.1;
038 private boolean m_isQualifierChange;
039
040
041 /* doWork - look for work
042 *
043 * @param nanoTime - current nanoTime
044 *
045 * @return none
046 * @throws none
047 */
048 public void doWork(long nanoTime)
049 {
050   super.doWork(nanoTime);
051
052   //Determine if time to check trace settings - every 5 minutes
053   long elapsedNanoSeconds = nanoTime - m_checkNanoTime;
054   if (elapsedNanoSeconds > ((long)(5 * 60) * (long)1000 * (long)1000000))
```

```
055   {
056     m_checkNanoTime = nanoTime;
057     m_prg.checkConfigChange();
058     m_logEvent = m_prg.isTrace("event");
059   }
060
061   if (!m_update) return;
062
063   //Determine if time to update - every 5 seconds
064   elapsedNanoSeconds = nanoTime - m_publishNanoTime;
065   if (elapsedNanoSeconds > ((long)(5) * (long)1000 * (long)1000000))
066   {
```

Periodically generate and publish a new value for the float m_pFloat1.

```
067     m_publishNanoTime = nanoTime;
068     m_pFloat1.setState(true, true, System.currentTimeMillis());
069     m_pFloat1.setValue(m_priorValue);
070     m_priorValue = (float) (m_priorValue + 1.0);
071
072     publishEvent(false, m_isQualifierChange, m_pFloat1);
073
074     m_isQualifierChange = false;
075   }
076 } //doWork
077
078
079 /* init - initialize
080 *
081 * @param prg - net program base
082 *
083 * @return none
084 * @throws none
085 */
086 public void init(netProgramBase prg)
087 {
088   m_prg = prg;
089
090   m_logEvent = m_prg.isTrace("event");
091
```

Initialize instances of infrastructure class floatValueState that are used to contain a value being published or subscribed. Note that "/eventTest/mf1" would normally come from configuration but it is hard coded to keep the sample simple.

```
092   m_pFloat1 = new floatValueState();
093   m_pFloat1.initDefault();
094  valueQualifierBase qualifier = (valueQualifierBase) m_pFloat1.getQualifier();
095   qualifier.setAdminSource(sourceType.MEASURED);
096   qualifier.setSource(sourceType.MEASURED);
097   qualifier.setSourceError(sourceErrorType.NONE);
098   qualifier.setSeverity(severityType.INFORMATION);
099
100   m_pFloat1.init(true, eventType.FLOAT, "/eventTest/mf1", null, null, null);
101   m_pFloat1.setValue(m_priorValue);
102
103   m_isQualifierChange = true;
104
105   m_sFloat1 = new floatValueState();
```

```
106   m_sFloat1.initDefault();
107
108   m_sFloat1.init(false, eventType.FLOAT, "/eventTest/mf1", null, null, this);
109
110   m_prg.trace("sfEventTest", "init", null, null, m_pFloat1);
111
112   super.init(m_prg, true);
113
```

If this instance is copy A then act as publisher, else act as subscriber.

```
114   if (m_prg.getProgramCopyId().equals("A"))
115   {
116     registerAsEventPublisher(m_pFloat1);
117   }
118   else
119   {
120     subscribeToEvent(m_sFloat1);
121   }
122 } //init
123
124
125 /* onAccessDown - communication access down
126 *
127 * @param none
128 *
129 * @return none
130 * @throws none
131 */
132 public void onAccessDown()
133 {
134   m_update = false;
135 }
136
137
138 /* onAccessUp - communication access up, start configuration
139 *
140 * @param none
141 *
142 * @return none
143 * @throws none
144 */
145 public void onAccessUp()
146 {
147   requestSync();
148 }
149
150
151 /* onActive - instance is now active
152 *
153 * @param none
154 *
155 * @return none
156 * @throws none
157 */
158 public void onActive()
159 {
```

```
160 if (m_prg.getProgramCopyId().equals("A"))
161 {
162   m_update = true;
163 }
164 }
165
166
167 /* onLicenseRetrieval - license retrieved from mainHub computer
168 *
169 * @param licenseInfo - license validated by mainHub
170 *
171 * @return none
172 * @throws none
173 */
174 public void onLicenseRetrieval(license licenseInfo)
175 {
176 }
177
178
179 /* onSyncComplete - event sync complete
180 *
181 * @param none
182 *
183 * @return none
184 * @throws none
185 */
186 public void onSyncComplete()
187 {
188 }
189
190
191 /* method onValueChange - value changed from event
192 *
193 * @param isRetry - true if retry attempt else first try
194 * @param valueState - value state instance
195 *
196 * @return true if success else retry still needed
197 * @throws none
198 */
199 public boolean onValueChange(boolean isRetry, valueStateBase value)
200 {
```

Generate a trace each time the subscriber sees a change.

```
201   if (m_logEvent)
202   {
203     m_prg.trace("sfEventTest", "update", null, null, value);
204   }
205   return true;
206 }
207
208
209 } //comm
```

5 Client to Server Communication Layer

The client to server communication layer consists of a manager program and an executive component resident with a user written client or server program as shown in Figure 5 - Client to Server Communication Layer. There is one active instance of a manager and standby instances that are informed of saved information such as recovery state and an in-progress message. An executive starts a program class and connects to the active layer manager instance via the message communication layer.

Figure 5 - Client to Server Communication Layer

A client uses a service to access functions of a server as shown in Figure 6 - Client to Server Mechanism. A service uses a client session to exchange messages with a server session in a remote server instance.

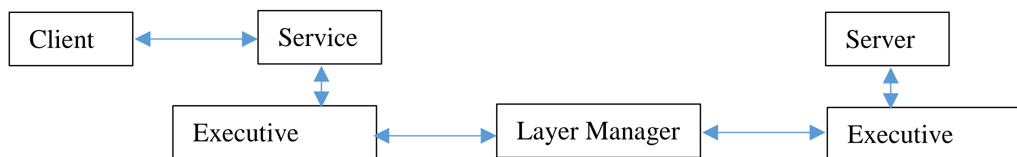

Figure 6 - Client to Server Mechanism

A service is a client resident API written by a server programmer. A service API permits a client to access server functions with a direct call interface.

A client program calls a service to request a connection to a server associated to a work session. A service requests the executive to create a server session in a server and a client session in the client that associates a client program work session to the client session and remote server session. The executive passes the request to the manager layer.

The layer manager, upon receiving a request to create a client to server connection, validates that a client program is permitted to access the requested server. If not, then a violation is reported and the connection to the client program instance is terminated. If yes, then the executive on the server program is directed to create a server session that includes the functions that a client can access. The executive on the client program is directed to create a client session associated to the server session and the client work session.

The executive on the client program directs the service to create a client session. The service creates a client session and informs the client program of the client session associated with the client program work session. The client is informed of a client session becoming available.

A client program calls the API on the client session to obtain server functionality. A service converts a client session API call to a server message sent via a client session to a server session. The executive passes session recovery state and the message to the layer manager.

The layer manager saves session recovery state and any message. A message is forwarded to its destination. If a connection to an executive is lost, then affected sessions are assigned and recovered by an instance of the identical program. Any external user, or the other program in the client to server association, is not affected except for observing a slower response when a failure and recovery scenario occurs.

An executive within a server program receives a client message from a layer manager. The executive calls the server via an API on a server session to process a message from a client. The server checks if a client is permitted to access the requested function. If not, then a violation is reported and the server session is terminated. If yes, then work is performed and the result is formed as a message that is sent to a client by calling an API on a server session.

An executive within a client program receives a server message from a layer manager. The executive calls a service via an API on a client session to process a message from a server. The service processes a message and calls the client program via the client session API. The client program accepts data or work completion from a server, integrates this into the work performed for and saved as state within a work session, and continues the work being directed by the work session.

Both a client program and server program are designed to support multiple sessions. An executive within a client program calls to create a new work session or a recovered work session with an associated client session. An executive within a server program calls to create a new server session or a recovered server session with an associated client session. Note that a server program can also be a client program. Care must be taken to ensure a timely response to client requests and to prevent a circular chain of calls through multiple servers.

An astute reader will wonder how a client gets a work session. There is an initial work session when a program is started. A work session called an application session can also be created to support an external user. The layer supports configuration of a user interface service and a user manager service. An external user connects to an interface service in a manner specific to a server of the interface service. The interface service requests the user manager to identify a new user. A validated user can request connection to a program designated as an application. A user manager requests the layer to create an application session. The layer directs an application to create an application session associated to an external user. The application uses an interface service to open and manage windows containing fields that appear to an external user. This mechanism allows one or more kinds of programs to manage application sessions to perform work. The application interface to a user is defined by a service added to the middleware. This mechanism is intended to support a gateway to web application instances.

A server program should use the message distribution layer to keep multiple instances of the same server program in contact and synchronized. A server requests a virtual connection to an active server instance. The connection provides two-way message delivery or disconnects upon delivery failure. A server can publish a message identified by a unique type identifier. A server can be a subscriber that receives a published message by type. A starting server receives the last published message for each type of subscribed message.

5.1 Sample Service Client Session

Class sFile1ClientSession is a service client session that extends the infrastructure client session and implements a service API. The sample service is a simple request of a file from the server that either returns the file content or an error indication. This example uses convertible data classes and interfaces that are documented in the programmer guide book. Hopefully, this example is useful without knowing the exact details of these classes and interfaces.

```
001 //Copyright Eugene Nelson Software 2012 All Rights Reserved
002 //Reproduction or Other Use Only By Permission of Authors
003
004 // sFile1ClientSession
005
006 package nelson.eugene.serviceFrame.services.sFile1;
007
008
009 import nelson.eugene.serviceFrame.gc.*;
010
011 import nelson.eugene.serviceFrame.bus.*;
012
013 import nelson.eugene.serviceFrame.services.*;
014
015 import nelson.eugene.serviceFrame.exec.*;
016
017
018 /* class sFile1ClientSession
019 *
020 * client session access to remote service
021 *
022 */
023 public class sFile1ClientSession extends clientSession
```

```
024 implements clientSessionInterface
025 {
026 //attribute
```

The service client session class holds references to the environment.

```
027 private netProgramBase m_prg;
028 private serviceExecInterface m_exec;
```

The class holds the instance of the client side manager.

```
029 private sFile1Access m_serviceAccess;
```

The class holds the user reply instance.

```
030 private sFile1ReplyInterface m_reply;
031
```

The class holds the message class exchanged with the server.

```
032 private fileMessage m_fileMsg = null;
```

The class holds the map from schema name to message class.

```
033 private gcSchemaToClassMap m_map;
034
035
036 /* method delete - delete session
037 *
038 * @param none
039 *
040 * @return none
041 * @throws none
042 */
043 public void delete()
044 {
```

Tell the client side manager to forget about this client session.

```
045   m_serviceAccess.deleteAccess(getServSession());
```

Tell the clientSession that the connection is gone.

```
046   super.disconnect();
047 }
048
049
```

This is part of the unique service signature. It allows a user to request the content of a remote file.

```
050 /* getFileContent - get file content
051 *
052 * @param vaultName - name of vault
053 * @param path - path to directory containing file
054 * @param fileName - file name
055 *
056 * @return requestId
057 * @throws none
058 */
059 public int getFileContent(String vault, String path, String fileName)
```

```
060 {
```

Get the next unused requestId and advance the id for the next time.

```
061   int requestId = getRequestId();
062   advanceRequestId();
```

Get a new instance of a fileMessage class. This must be done since the message will be placed in a request queue that is encoded and sent to the server only after this thread returns back to the service technology.

```
063   fileMessage fileMsg = (fileMessage)m_fileMsg.newInstance();
```

Generate a get file content message.

```
064   fileMsg.setGetFileContent(requestId, vault, path, fileName);
```

Queue the request message to be sent to the server.

```
065   sendRequestMessage(fileMsg);
```

Return the requestId for this request in case the user must correlate a response back to a request.

```
066   return requestId;
067 } //getFileContent
068
069
070 /* method getSchemaMap - get schema map of class(es)
071 * placed on client / server session lists
072 *
073 * @param none
074 *
075 * @return schema map of class(es) placed on client / server session lists
076 * @throws none
077 */
078 public gcSchemaToClassMap getSchemaMap()
079 {
```

Return the map of schema name to message class as created by init method.

```
080   return m_map;
081 }
082
083
084 /* method getState - get current state
085 *
086 * @param none
087 *
088 * @return gcData based object containing current state
089 * @throws none
090 */
091 public gcData getState()
092 {
093 //no state beyond that held by sessionBase request, response,
094 //event and ack lists
095   return null;
096 }
097
098
099 /* init - initialize service manager
```

```
100 *
101 * @param prg - net program base
102 * @param exec - serviceExec control interface
103 *
104 * @return none
105 * @throws none
106 */
107 public void init(netProgramBase prg, serviceExecInterface exec)
108 {
```

Save environment context instances.

```
109   m_prg = prg;
110   m_exec = exec;
111
```

Instantiate a fileMessage class instance that is used to exchange messages with the server.

```
112   m_fileMsg = new fileMessage();
113   gcErrorResult errorInfo = m_fileMsg.initDefault();
114   if (errorInfo != null)
115   {
116     m_prg.trace("sFile1", "init", null, errorInfo);
117     return;
118   }
119
```

Create the map of schema name to message class.

```
120   m_map = new gcSchemaToClassMap();
121   m_map.addMapping(m_fileMsg);
122 }
123
124
125 /* initReply - init reply
126 *
127 * @param serviceAccess - service access
128 * @param reply - user reply instance
129 *
130 * @return none
131 * @throws none
132 */
133 public void initReply(sFile1Access serviceAccess,
134 sFile1ReplyInterface reply)
135 {
```

Save client side manager and user reply instances.

```
136   m_reply = reply;
137   m_serviceAccess = serviceAccess;
138 }
139
140
141 /* method onConnect - session connected
142 *
143 * @param none
144 *
145 * @return none
146 * @throws none
```

```
147 */
148 public void onConnect()
149 {
```

Tell user that client session can now be used.

```
150   m_reply.onAccessConnect(this);
151 }
152
153
154 /* method onDisconnect - session disconnected
155 *
156 * @param none
157 *
158 * @return none
159 * @throws none
160 */
161 public void onDisconnect()
162 {
```

Tell user to stop using the client session.

```
163   m_reply.onAccessDisconnect(this);
164 }
165
166
167 /* onDelete - on delete of server session by client side
168 *
169 * @param none
170 *
171 * @return none
172 * @throws none
173 */
174 public void onDelete()
175 {
```

Nothing to do since this is a simple service with no client state or fancy stuff supplied to user.

```
176 }
177
178
179 /* method onEventMessage - process event message from service
180 *
181 * @param event - event message
182 *
183 * @return none
184 * @throws none
185 */
186 public void onEventMessage(serviceMessageBase event)
187 {
```

Nothing to do since server does not send event messages.

```
188 }
189
190
191 /* method onResponseMessage - process response message from service
192 *
193 * @param response - response message
```

```
194 *
195 * @return none
196 * @throws none
197 */
198 public void onResponseMessage(serviceMessageBase res)
199 {
```

Cast the response message to the fileMessage class since this is the only class in the schema map that can be used by the infrastructure.

```
200   fileMessage response = (fileMessage) res;
```

Get any error returned from the server.

```
201   gcErrorResult errorInfo = response.getErrorInfo();
202   if (errorInfo != null)
203   {
```

Tell the user about a server error.

```
204     m_reply.onAccessError(this, response.getRequestId(), errorInfo);
205   }
206   else
207   {
```

Tell the user the serial content of the requested file.

```
208     m_reply.onSerialContent(response.getRequestId(),
209       response.getContent().getSerialPackage());
210   }
211 }
212
213
214 } //sFile1ClientSession
```

A service client session class for any service will look somewhat like this sample as it must extend clientSession and must implement clientSessionInterface. Of course, the service API is unique and the message convertible data class is tailored to the functionality being accessed.

5.2 Sample Server Session

This is an example of a server session. The prior section shows the client session that matches this server session to provide a function to a client. A service server session called sSession extends serverSession and implements serverSessionInterface.

```
001 //Copyright Eugene Nelson Software 2012 All Rights Reserved
002 //Reproduction or Other Use Only By Permission of Authors
003
004 // sSession
005
006 package nelson.eugene.serviceFrame.servers.sFile1;
007
008 import java.util.ArrayList;
```

Import the java classes needed to perform a file open and read.

```
009 import java.io.File;
010 import java.io.RandomAccessFile;
011 import java.nio.channels.FileChannel;
```

```
012 import java.nio.ByteBuffer;
013
014 import nelson.eugene.serviceFrame.gc.*;
015
016 import nelson.eugene.serviceFrame.bus.*;
017
018 import nelson.eugene.serviceFrame.services.*;
019
020 import nelson.eugene.serviceFrame.exec.*;
021
```

Import the message class from the service.

```
022 import nelson.eugene.serviceFrame.services.sFile1.*;
023
024
025 /* class sSession
026 *
027 * server session access to remote service
028 *
029 */
030 public class sSession extends serverSession
031 implements serverSessionInterface
032 {
033 //attribute
```

The class holds references to the environment.

```
034 private netProgramBase m_prg;
035 private serviceExecInterface m_exec;
```

The class holds reference to the server manager.

```
036 private sFile1Server m_service;
037
038 private state m_state;
039
040 private String m_versionDirectory;
041 private gcErrorResult m_errorInfo;
042
```

The class holds the message class exchanged with the server.

```
043 private fileMessage m_fileMsg = null;
```

The class holds the map from schema name to message class.

```
044 private gcSchemaToClassMap m_map;
045
046
047 /* doWork - look for work
048 *
049 * @param currentTimeMillis - current time in milliseconds
050 *
051 * @return none
052 * @throws none
053 */
054 public void doWork(long currentTimeMillis)
055 {
```

Call from the service infrastructure in case some work needed by server session (as requested by server manager).

```
056 }
057
058
059 /* method getSchemaMap - get schema map of class(es)
060 * placed on client / server session lists
061 *
062 * @param none
063 *
064 * @return schema map of class(es) placed on client / server session lists
065 * @throws none
066 */
067 public gcSchemaToClassMap getSchemaMap()
068 {
```

Return the map of schema name to message class as created by init method.

```
069   return m_map;
070 }
071
072
073 /* method getState - get current state
074 *
075 * @param none
076 *
077 * @return gcData based object containing current state
078 * @throws none
079 */
080 public gcData getState()
081 {
082   //no state beyond that held by sessionBase request, response,
083   //event and ack lists
084   return null;
085
086   //return m_state; State must be saved so that it can be restored.
087 }
088
089
090 /* init - initialize service manager
091 *
092 * @param prg - net program base
093 * @param exec - serviceExec control interface
094 * @param service - main service
095 * @param priorState - state content if prior state to be recovered
096 *
097 * @return none
098 * @throws none
099 */
100 public void init(netProgramBase prg, serviceExecInterface exec,
101                  sFile1Server service,
102                  gcData priorState)
103 {
```

Save environment context instances.

```
102   m_prg = prg;
```

```
103   m_exec = exec;
104   m_service = service;
105
106   //Restore state from prior session instance
107   //This example does not use state, however, follow this pattern when state
108   //must be saved and recovered.
109   m_state = (state)priorState;
110
```

Get the version directory for this server instance.

```
111   //Setup to access versionDirectory and exchange service messages.
112   m_versionDirectory = m_prg.accessFile().getVersionDirectory();
113
```

Instantiate a fileMessage class instance that is used to exchange messages with the client.

```
114   m_fileMsg = new fileMessage();
115   gcErrorResult errorInfo = m_fileMsg.initDefault();
116   if (errorInfo != null)
117   {
118     m_prg.trace("file1Test", "init", null, errorInfo);
119     return;
120   }
121
```

Create the map of schema name to message class.

```
122   m_map = new gcSchemaToClassMap();
123   m_map.addMapping(m_fileMsg);
124 }
125
126
127 /* method getFileContent - get file content
128 *
129 * @param path - path to directory containing file
130 * @param fileName - file name
131 *
132 * @return serial if no error else null on error
133 * @throws none
134 */
135 public gcSerialPackage getFileContent(String path, String fileName)
136 {
```

Perform the work needed to read the content from a file. A gcSerialPackage is how the gcData technology represents the serial in a sequence of ByteBuffer class instances. Most of this method is dictated based on how Java file I/O works.

```
137   gcSerialPackage serial = new gcSerialPackage();
138   serial.init(1);
139
140   String fullDirPathName = new String (m_versionDirectory + "\\" + path);
141   File dir = new File(fullDirPathName);
142
143   File file = new File(fullDirPathName + "\\" + fileName);
144   try
145   {
146     FileChannel fc = new RandomAccessFile(file, "r").getChannel();
147
148     int len = (int)fc.size();
```

```
149     ByteBuffer buf = ByteBuffer.allocate(len);
150     fc.read(buf);
151     fc.close();
152     buf.flip();
153     serial.addPart(buf);
154   }
155   catch (Exception e)
156   {
157     m_errorInfo = new gcErrorResult();
158     m_errorInfo.init();
159     ArrayList<String> params = new ArrayList<String>();
160     params.add(file.getPath());
161     params.add(e.toString());
162     m_errorInfo.setErrorInfo("sf", "file_error", params);
163     return null;
164   }
165   return serial;
166 } //getFileContent
167
168
169 /* method onConnect - session connected
170 *
171 * @param none
172 *
173 * @return none
174 * @throws none
175 */
176 public void onConnect()
177 {
```

Ignore as nothing to do beyond what is done by init method.

```
178 }
179
180
181 /* onDisconnect - on disconnect by client side
182 *
183 * @param none
184 *
185 * @return none
186 * @throws none
187 */
188 public void onDisconnect()
189 {
```

Ignore as nothing to do beyond class going away on disconnect.

```
190 }
191
192
193 /* onRequestMessage - on request message
194 *
195 * @param request - request message
196 *
197 * @return none
198 * @throws none
199 */
200 public void onRequestMessage(serviceMessageBase req)
```

```
201 {
```

Cast the request message to the fileMessage class since this is the only class in the schema map that can be used by the infrastructure.

```
202   fileMessage request = (fileMessage)req;
203   if (request.getFileMsgType() == fileMsgType.GET_FILE_CONTENT)
204   {
205     gcSerialPackage serial;
```

Get file content via the local class method.

```
206     serial = getFileContent(request.getPath(), request.getFileName());
207     if (serial == null)
208     {
```

If nothing returned then send error back in new file message.

```
209       fileMessage fileMsg = (fileMessage)m_fileMsg.newInstance();
210       fileMsg.setGetFileContentReply(request.getRequestId(), m_errorInfo);
211       sendResponseMessage(fileMsg);
212     }
213     else
214     {
```

Return serial file content back in new file message.

```
215       fileMessage fileMsg = (fileMessage)m_fileMsg.newInstance();
216       m_fileMsg.setGetFileContentReply(request.getRequestId(), null);
217       m_fileMsg.getContent().setSerialPackage(serial);
218       sendResponseMessage(m_fileMsg);
219     }
220   }
221   else
222   {
223     m_prg.trace("file1Test", "badRequest", null, null, request);
224     disconnect();
225   }
226 }
227
228
229 } //sSession
```

This sample server session has a structure that is similar for all servers. There is flexibility that allows for how functionality occurs. Perhaps the most work, not shown, is in how information is shared with either standby or other active instances of a server. The serviceExecInterface environment method getBusInterface() returns access to the message bus. The bus supports both point to point message connections and publisher to subscriber near real time event messages. These message distribution mechanisms provide a reasonable way to share information with different pros and cons.

6 Characteristics of a Better Middleware

Reliability is improved by:

- Each message travels two paths. Failure of one path does not affect communication between programs.
- A starting subscriber receives the last message published for each subscribed event message type.
- In progress work and communication between a client and server are represented as data in a session. Session contents are persisted periodically or upon significant change. A session can be reassigned and recovered after a program failure. An external user can only observe slower performance when a failure scenario occurs.
- A server supports multiple instances where either one is active and the others are standby or all are active. Changes in server state are passed between instances such that a client receives an identical function from any active server.

Security is improved by:

- Two programs can only communicate as configured by an administrator.
- Published event messages are only distributed to subscribers as configured by an administrator.
- A server knows the functions that a client is permitted to request as configured by an administrator.

Other characteristics can be improved such as:

- Less programmer effort is needed to convert data between formats.
- Less administrator effort is needed to operate a 24/7 system.
- Multiple versions of a system can operate at the same time on the same hardware. This supports testing of a new version and access to an older version.
- A client program can communicate with a server program running on a different kind of computer under a different operating system and written in a different software language.
- Work organized as a recoverable session can be scaled over multiple program instances on different computers.
- Survive expected disaster scenarios by placing computers within a system at geographically dispersed sites.

7 Analysis of a Better Middleware

The characteristics of the data format conversion layer are quite interesting. Roughly ten times as much local memory is needed to hold a convertible data class instance versus a more traditional data class representation.

A programmer must spend a week or so to learn how to define a convertible data class. A programmer only needs a few minutes to program the initialization of meta data which is in addition to the time it takes to define a more traditional data class representation.

The ease of converting data formats leads to some significant advantages. An administrator prepares configuration in a human readable text format. Configuration is converted, persisted and distributed in a serial format. Little programming effort or execution time is needed to obtain configuration stored in serial and convert to an in-memory convertible class format.

Each message is held in a convertible data class instance. This allows a message to be converted to and from serial format for transmission between programs. A message is easily converted to human readable text format for inclusion in a log file.

The characteristics of the message communication layer are what might be expected. Extra processor time and memory are used to convert a message between a convertible data class instance and a serial format.

Messages are aggregated into a super message such that needed resources are reduced as work load increases. Extra resources are used to replicate a message, transmit via two paths, and ignore a duplicate message.

The use of extra resources can be mitigated by spreading work load across extra computers.

Note that bulk data can be securely transmitted directly between a client and server upon a connection requested by a server to a client.

The characteristics of the client to server communication layer are quite interesting. Extra resources are used to hold and convert a message between formats.

The work sessions of a failed client or server program can be assigned and recovered by an identical instance of the same client or server program.

A client or server program instance can support multiple sessions. This makes it more likely that aggregation of messages into a super message will help to reduce overall system work load.

A client and service can access a server running on a different type of computer under a different operating system written in a different software language. This means a type of server can be designed and implemented once and reused by any type of client. This approach offers a reasonable bridge between different legacy environments such as Java and .Net C++.

8 Embedded Environment Support

The Java implementation is designed and implemented for both an operating system and embedded environment. The software is divided into 300+ classes with a file per class. Dependence on environment specific features are isolated into just a few of the classes.

A class detects, reports and handles unusual situations as needed. A report captures interesting details into an error message class that is passed to a trace class. The trace class implementation within an operating system environment translates the error message class into text and appends the text to a trace file. If the error occurs within the context of a connection then the connection is terminated. The caller is always informed whenever an error occurs so that proper handling of the situation can be performed by the caller. An embedded environment needs to supply an appropriate trace class implementation. Note that there are no print statements.

The work of many classes is based on receiving or sending data within a memory buffer. The allocation of memory is common to most all environments.

There is tracing of events and messaging within key classes that can be enabled via configuration.

The design is suitable for porting to another language such as C++. The design can even be ported to assembly language when there is an appropriate substitute for a class instance.

A special test command simulates the infrastructure running on many computers. A class instance is created per computer that instantiates all the class instances needed to run the infrastructure on a computer. A few low-level classes simulate a TCP/IP connection passing messages on a connection. Each simulated computer has a dedicated incoming buffer that is filled with a message from a sending simulated computer. Most of the class instances work as if they are running on different computers even though everything runs under the test command. This command is useful to test most functionality.