JAUS .NET: A Reflective Implementation
Jean-Francois Kamath, Carl Crane
University of Florida
P.O. Box 116300
Gainesville, FL 32611
(352) 392-9461
jfkamath@ufl.edu ccrane@ufl.edu

ABSTRACT
The JAUS reference architecture provides definitions for message formatting and communications in military robotics. While this helps to standardize cross-system communications, one of the long-standing challenges in developing JAUS compliant applications has been the lack of standardized tools. A new implementation of JAUS has been developed using the .NET framework in an attempt to provide an easily used toolset for rapid component development.

JAUS .NET has several key features:
- Dynamic reconfiguration
- Easily extended message set
- Simple, rapid component development
- Cross-platform compatible

Keywords
JAUS, .NET, reflection

1. INTRODUCTION
The Joint Architecture for Unmanned Systems (JAUS) [6] is a standard for communications in US military robots, defining communications protocols, an extensible message set, and the exact format and structure of the serialized messages. By implementing this architecture, a JAUS component can communicate with any other component without additional configuration.

The motivation for the JAUS .NET reference implementation began after the DARPA Urban Challenge in 2007. The development team at the Center for Intelligent Machines and Robotics (CIMAR) recognized the need for more accurate simulation and modeling tools with which to test their systems. Microsoft Robotics Studio (MSRS) [7][8] was selected as the simulation environment for a number of reasons including speed of development and readily available physics and sensor modeling. However, one of the requirements was that the simulations be fully compliant JAUS components. Unfortunately, the existing implementation of JAUS, written in C/C++, was not easily adapted to function with the .NET-based MSRS. As a result, a full implementation of JAUS was created using the .NET framework.

JAUS .NET is actually built using the Reflective Communications Framework (RCF), a toolset designed for rapid deployment of communications architectures and distributed systems. The RCF defines many of the core features available in JAUS .NET, including generalized message structure and dynamic reconfiguration. A full discussion of the RCF is beyond the scope of this paper, but the relevant parts that are directly visible to JAUS developers will be investigated where relevant.

Since JAUS is a distributed system architecture, it became necessary to design a toolset that would enable rapid deployment of the components. In 1994, Jeff Kramer performed a very thorough analysis of the state of distributed computing and its development. His review highlighted a number of key requirements for successful deployment of distributed systems and discussed the approaches that had been developed to address the challenges involved [9]. Kramer’s review highlights the two key features common to all distributed systems. The first is that all software components need to be highly modular so that they can be deployed in the environment with minimal effort. Each component must completely encapsulate its functionality. Second and most important, all components need to implement some form of standardized communications interface to provide access to the rest of the system. The choice of communications primitives and messaging architecture is paramount to the design and implementation of the systems.

JAUS .NET meets both requirements for distributed system development, in part since it utilizes the JAUS reference architecture. It defines a toolset that centralizes communications in derived applications, enables rapid reconfiguration of the application modules, and allows a developer to easily extend the message set. The centralized communications are fully compatible with other JAUS implementations, but have the advantage of automating data routing and message conversion. It will be shown in later sections that a highly flexible architecture is possible that implements an extremely rigid system definition.

2. REFLECTION
At the core of the framework is the reflection oriented model used to describe and indentify all of the dynamic components. These include the message set definitions, processing functions, and the structuring of the messages themselves. All of this information is extracted at runtime to determine the program behavior.

The Microsoft .NET environment used to develop the framework defines several constructs that enable reflective programming. Assemblies are the core of all .NET applications as they contain all information regarding a compiled code library [3][4]. These include executables and libraries which encapsulate all functionality for a program. These assemblies can contain object definitions, methods, references to other assemblies, and most
importantly specific information about each of these in the form of ‘attributes’ [2]. These descriptors can describe a wide range of factors such as publicly visible names and interpretation methodology.

A .NET attribute is an object definition that stores information about the piece of code to which it has been attached. They can store virtually any type of information that can be processed at compile time, making them very useful for storing generally accessible data about application modules. This data is stored in internal fields and can be accessed by a number of means, the most common of which is ‘properties’. Properties are a .NET construct for code readability that encapsulates method access to fields while providing an interface that still looks like direct field access.

Using a combination of built-in and custom created attributes, assemblies can be created that are fully self-descriptive to applications that know how to interpret them. This ability is leveraged by the underlying framework used by JAUS .NET to dynamically gather information needed for program execution. Developers can create message and processing libraries that contain all information needed for full operation with minimal addition to the code base by added short tags to the code that do not interfere with the structuring or readability. This allows the developer to create an application with virtually no restrictions and to even adapt existing applications to the new framework with minimal effort.

Three classes of custom attributes were specified to aid framework development. The first group identifies methods that are used to process messages that have arrived. The second group acts as an identifier for the individual messages in a message set. This is used to dynamically determine which objects need to be allocated at runtime during interpretation of data. The final attribute group describes the actual message structuring. This is used by the generalized serialization scheme to control which components of a message are serialized, how they are interpreted, and the exact processing order. The implementation of the serializer determines the final representation of messages being transported.

Attributes are easy to utilize in .NET assemblies. They are added to code in a manner similar to comments. The code layout remains unchanged save for an extra line that is processed by the compiler when generating the assembly. The proper form of attribute usage is as follows:

**BNF:**

```plaintext
<attribute> ::= "[" <name> "(" <argList> ")""]" <targetCode>
<argList> ::= λ | <value> | <value> "," <argList>
<argList> ::= <propList> ::= <property> | <property> "," <propList>
<property> ::= <name> "=" <value>
```

**C#:**

```csharp
[@<name>({<argList>})]
<targetCode>
```

One will notice that the argument list provided to an attribute constructor can contain multiple property assignments of the form `name = value`. This is a .NET specific feature that allows a developer to assign additional values to an object at creation time without having to write extra lines of code. For example, the RCF defines an attribute for specifying fields to process name `PackableMember`. This attribute stores several pieces of data regarding the order and nature of processing. A pseudo-C# form of its usage might be:

```csharp
[PackableMember(0, true, Optional=true)]
int number;
```

The above example shows all of the elements of the BNF description. Two values are passed in as part of the argument list, followed by a property list containing one element. This associates three pieces of data with the immediately following `number` field. Note that the single line of attribute code does not affect the functionality of the object itself, instead attaching data to it that can be processed separately. As such, an application can be rapidly converted to compatibility with the Reflective Communications Framework. More detailed examples of attribute usage are presented in the section on message structuring.

### 3. COMPONENT DESIGN

![General component structure](image)

A component developed with JAUS .NET requires two key elements: a persistent instance of a `JausComponent` object and one or more methods that have been marked as capable of processing messages. A developer may also wish to implement methods for handling state machine transitions, connection notifications, and service connection health monitoring. Each of these parts is discussed in detail in the following sections. Figure 1 shows the core pieces of a JAUS component and the primary data flow during operation. Notice that the `JausComponent` object is responsible for all IO with the JAUS network. Newly received messages are routed to the message processing methods. Changes to the state machine are signaled to the state handling method. Updates to the node manager connection state and any issues with service connections are routed to the connection status processing methods. Service connections are automatically managed, created, and monitored by JAUS .NET, only requiring a developer to start the connection process for inbound services.

#### 3.1 Node Manager Interface

As with any communications architecture, JAUS .NET has several distinct pieces which must work closely together. Since every JAUS component will use the same parts for communications, it was decided to encapsulate the core functionality in an overarching management object called a Node Manager Interface (NMI). It serves two purposes: to provide easy, consolidated access to all parts of the communications system and to automate many of the processes, such as signing into the network. Both
the message handlers, their associated communications interfaces, and the service connection manager are monitored and managed by the NMI. It also provides a full state-machine implementation and hooks for an application to attach itself. In fact, the NMI itself uses the state machine for connection management.

The NMI defines two message handlers, two UDP communications interfaces, a service connection manager, and a number of methods for monitoring various processes in the architecture. Figure 2 illustrates the interactions between the various components, many of which are automated. The most important function the NMI performs is setup and initialization of these component parts. It is responsible for making the various calls to attach message processors to the message handlers, connecting the communications interfaces to the message handlers, and providing clean operations such as startup and shutdown procedures.

Note that the NMI both directly and indirectly interacts with the two message handlers. They automatically route messages to the appropriate parts of the interface, which then processes the data to determine appropriate actions. The NMI performs all of its networked communications through these handlers as well as starting, stopping, and reinitializing them as needed.

Lists of objects are maintained that can be used to rebuild the hashables at any time. Hashable elements are lists that contain the actual information of either message types or methods to invoke. This approach provides the flexibility for the message handler to dynamically update its knowledge base and perform error checking, adding an element of safety to the configuration. Note that processing methods can be added and tracked either as part of an object or individually via encapsulation.

3.2 Message Handler

While the communications interface is responsible for getting data to and from application modules, the message handler performs several critical functions related to processing of that data. First and foremost, it acts as the primary interface to the communications interface in an application. It performs the appropriate final operations needed for proper interpretation of raw data coming off the pipe. Derived handlers can also control aspects of sending of data on these interfaces if deemed appropriate. Second, it has knowledge of the currently supported messages that it uses in the interpretation of the data. This knowledge store can be dynamically changed. Third, the handler acts as the final message router for the application. Portions of the application can be dynamically linked to the message handler along with information about the types of data that they wish to process. This approach allows the developer to dynamically change the message set and processing capabilities of the application with relatively little effort. The usage is the same regardless of structuring.

Figure 3 shows the data structure for the base message handler while Figure 4 illustrates the relationship between the message handler and both the message libraries and processing methods. Message types and processors are tracked in much the same way.

In addition to the tracking capabilities, the base message handler definition is built as a multi-threaded module to allow asynchronous, non-blocking processing of messages and data. Data will typically be received by a method that is attached to the communications interface being used by the handler while processing is performed in a specific method run by the stand-alone thread. Since the threading setup is already provided, it is only necessary for a developer to override the threaded method to gain its benefits. The chapter describing the JAUS .NET reference implementation will show in greater detail how these pieces interact.

3.2.1 Message routing

The final service that the abstract message handler provides is maintenance of a table of methods that are meant to be used as message processors. As with the table of message types, this information is updated dynamically, allowing addition and removal of message processors on the fly. This is accomplished by two means, both involving the use of reflection. The first automatically gathers the information from an object that may contain processing methods. The second allows more explicit addition of methods.
For the automated approach, an object is added to the list of message processors. Attributes are then extracted in much the same way as with message types, allowing the handler to update the table. The attribute used to identify processing methods is aptly named `MessageProcessor`. It contains the GUID of the message type that the method can process, which is meant to be used in conjunction with the message types table. Multiple `MessageProcessor` attribute tags can be applied to a single method if desired to allow a single method to process a variety of data types. Once a method has been identified as a message processor, it is added to the appropriate list(s) in the hashtable for later retrieval during message routing. Proper syntax for specifying a processor is:

```csharp
[MessageProcessor(<GUID>)]
void <methodName>(<argList>) { ... }
```

The argument list is determined by the architecture and the exact form of a processing method needs to be explicitly defined for the message handler. This allows the handler to ensure that a method marked with a `MessageProcessor` attribute is structured properly, thereby avoiding invocation errors. The form is expressed using a delegate, a .NET construct that acts as a handle to a method. For a message handlers, the proper structure for a delegate is:

```csharp
delegate void <delegateName>(<argList>);
```

A message handler initialized with a delegate type will only process methods that exactly match the format given and have been marked as message processors. Any other methods are ignored.

The second approach for added message processors allows a developer to explicitly specify a method to act as a processor and the associated GUID of the message type that it should be passed. While the automated approach will be the most widely used, this alternate method can be useful in situations where the exact data type that needs to be handled may not be known until runtime. Certain service oriented architectures may use make use of this feature as they tend to be highly dynamic in nature with a continuously changing structure. Dynamically added methods still have to follow the same structure as the delegate used by the message handler. Adding them follows the form:

```csharp
<handlerName>.AddMessageProcessor(
    new <delegateName>(<instance>,<methodName>),
    <GUID>);
```

Notice that the message GUID will be associated with the instance method, guaranteeing that only messages matching that ID will be routed to the method. If one method in an object is supposed to process several messages with different GUID’s it will have to be added multiple times as only one ID can be associated at a time.

Message routing is accomplished by retrieving the list of processing methods from the table associated with the GUID of the message that has been received. These methods can then be invoked as deemed appropriate by the interpreter without prior knowledge of exactly which parts of the application have requested the information. Properly implemented, this allows flexible asynchronous processing of the data.

### 3.2.2 Dynamic message set

Much of the information regarding known message types and processors is gathered using the reflective capabilities of the .NET environment. A message handler is always created with knowledge of the specific tag used to identify message types for the communications framework being used. When an assembly is attached to a message handler, the handler filters the assembly information for all data types that have been marked with the matching attribute. This is used to build a hashtable of known message types. For example, the JAUS .NET framework uses a `JausContents` attribute to mark data types as usable for messaging. This is used in conjunction with another attribute that identifies the message as being processable by the serialization library. The code would look similar to this:

```csharp
[Packable, JausContents(<GUID>)]
<messageDefinition>
```

The table of known types is organized by globally unique identifiers (GUIDs) that are specific to each message type that serves two purposes. First, data types can be quickly pulled using the known GUID. Since a GUID is a unique identifier, this can be used to rapidly filter the known data types based on information passed during messaging. Secondly, error checking can be performed to look for duplicate or conflicting type definitions. If one or more assemblies assigns the same GUID to multiple data types, the ability to differentiate and process the messages properly will be lost. Figure 5 shows the process for populating the table of known data types.

At any time during operation, an assembly can be attached or unlinked from a message handler, triggering a refactoring of the data type hashtable with full error checking. The entire process is automated in the base class only requiring knowledge of the attribute type to extract from the assembly. An important note is that the hashtable is meant to be used in conjunction with the interpretation aspect of the message handler, providing it with information about the types of data that can be processed. It is left up to the developer how this is used, but a suggested structure is presented in the next section.

![Figure 5. Data type hashtable update.](image)

The creation of the hashtable is a two-step process involving collecting and sorting of the attributes and then checking for duplicate definitions. Attribute information is sorted into lists of common GUIDs. As previously discussed, each GUID should be unique, so lists containing more than one entry indicate a conflict in definitions. An exception is thrown to signal the problem, though this does not have to break the program execution. It is possible for the developer to account for situations that break the design intent with proper exception handling. This will often be performed in the interpretation portion of the message handler.
4. MESSAGE FORMAT

When attempting to address the issue of custom data types in a generalized messaging framework, it becomes necessary to identify the common traits amongst the myriad representations available. Any data type used for messaging is likely to store the information in internal fields. The order in which these fields are processed during serialization determines the final message structuring. However, simply converting the individual fields in the correct order will be insufficient to handle more complex messages. There may be messages with mutable structures that depend on various factors at runtime, requiring intermediate processing to determine which fields need to be handled.

Taking these factors into account, a set of common features can be compiled that together should address the needs of a large number of messaging architectures:

- Specify which fields should be processed during serialization/deserialization. Note that the term ‘field’ is used to represent any construct that can be used to both set and retrieve a piece of data.
- Indicate optional fields and their dependencies. There may exist situations where a field should only be processed if one or more conditions are met. This will allow the developer to easily create a changing message format.
- Specify methods for custom handling of data at different stages of processing. Some messages may require highly specialized routines to appropriately handle conversion to and from the transportable format.

The attributes that have been developed to address these needs fall into two categories: those attached to fields that represent the data and those that are attached to methods used for specialized processing. Following is a complete pseudo-code example of a message definition to illustrate the primary features and usage. The details of each part will be discussed in detail in later sections.

BNF:
\[
<messageDef> ::= \[Packable\] <messageCode>
<messageCode> ::= <modifiers> <name> "\{" \\
<bodyCode> "\}"
<bodyCode> ::= ( λ | <memberAttribute> \\
| <methodAttribute> \\
<property> | <methodCode> ) \\
\) <bodyCode>

<memberAttribute> ::= \[PackableMember(" <argList> ")\]
<methodAttribute> ::= \[DependencySpecifier(" <argList> ")\]
<specialName>

C#:

```csharp
[PackableMember(1, Optional=true, DependencyIndex=1)]
public double Size { get; set; }

[DependencySpecifier]
bool DepSpec(int index) { ... }

[OnPacking]
object PrePack() { ... }
```

As can be seen from the above example, a message definition consists of a standard object definition that has been augmented with attributes. Some of them are applied to the fields and attributes to specify processing details while others are applied to various methods that modify the serialization behavior of the object. Simply by changing some of the attribute values, it becomes possible to completely redefine the serialized message structure without changing the object’s behavior at all.

4.1 Field attribute

The field attribute, called PackableMember, serves several purposes including marking a field for processing, specifying the processing order, and indicating functional dependencies. The processing order is determined by the index assigned to the field. The member is used as follows:

BNF:
\[
<argList> ::= <index> $preAllocate$ $properties$
<properties> ::= <optional> <dependencyIndex> $dependentOn$
\]

C#:

```csharp
[PackableMember(<argList>)]
type <fieldname>;
```

For example, a message might be setup as follows:

```csharp
[Packable]
public class MessageObject
{
    [PackableMember(0)]
    int field1;

    [PackableMember(1)]
    int field2;
}
```

When processed, field1 would be handled first followed by field2 because its index value of 0 is less than that of field2. However, if the developer needed to reverse the order of handling, then the only change would be to swap the index numbers of the fields. This effectively separates data type design from the representation in the messaging architecture, requiring minimal modifications by the developer to completely change the behavior. The only restriction on use of the PackableMember attribute is that it cannot be applied multiple times to a single field. As mentioned previously, the field attribute can also be used to indicate dependencies. It can contain information about whether or not a field is ‘optional’ as well as specifying dependency on a method that will determine if the field should be processed. The usage of said method will be discussed in more detail later in the proposal.
One final feature of the `PackableMember` attribute is the ability to specify pre-allocated fields. While this might seem like an unusual or restrictive flag, it is necessary for proper processing of more complex data types. This results from the way in which the Packer class leverages the reflective allocation of data types. It is possible to allocate an object of a type that is only known at runtime, allowing a program to dynamically create objects as needed. The RCF attempts to generalize the process, which inherently limits the allocation to being type-based only. Unfortunately, specifying the data type alone is not always sufficient to properly allocate the object. As a result, it becomes necessary for the targeted type to be processed as a pre-allocated object that itself contains the information needed for proper processing.

Looking at arrays as an example again, one will realize that for an array to be deserialized properly its size must be known ahead of time. A pre-allocated array can provide its size information to the environment, allowing a generalized approach to be taken in deserialization. However, if the array has to be dynamically allocated by the deserializer, the general nature of the process would be broken as specific constructs would be required to pass the size information through the various levels of processing. Strings and lists face similar problems in that they too contain multiple elements with the added complexity of variable length. As a result, the deserialization process uses the pre-allocated sizes of the specialized data types to iterate over their elements and populate them. The JAUS .NET chapter illustrates the proper use of pre-allocated example messages that require the added complexity. However, do note that beyond these three special cases, it is highly unlikely that a developer will need to pre-allocate other elements.

### 4.2 Processing method attributes

The collection of attributes used to specify specialized processing methods contains a number of members, none of which are particularly complex in design. They address the issues of dependencies and intermediate processing that might become necessary in some message implementations. These attributes have features similar to the field attribute that allows the developer to specify processing order and identification information.

As previously mentioned, optional fields indicate a method that will determine whether or not the field should be processed. This method is identified by a GUID and is passed the ID of the field that should be analyzed. The GUID is attached to the method via attribute named `DependencySpecifier` that is used to identify and invoke the method at runtime. Since each dependency method is assigned a unique ID, it is possible to have an unlimited number of such processors if needed. The proper usage of the Dependency specifier is:

```csharp
[DependencySpecifier(<methodIndex>)]
bool <methodName>(int index) { ... }
```

Note that the form of a dependency specifier method is very rigidly defined with the only developer defined elements being the index identifying the method and the method name. The return Boolean value indicates whether or not to process the field whose index has been passed to the method.

There may be situations where using dependency methods are not enough to provide the flexibility needed for proper message serialization. These can be split into three general categories: pre-, mid-, and post-processing methods. Pre-processing methods are those that need to be executed before the individual fields can be handled. This could include preparation of data, finalizing of certain communications tasks, or a myriad of other cases. Mid-processing methods are those that need to be invoked during the serialization process. Some message structures may benefit from performing certain tasks between processing of fields. Post-processing methods are those that need to be run after the main body of serialization or deserialization has been completed. This is very similar to usage of pre-processing methods.

Both pre- and post-processing methods can be assigned index values that determine the order of invocation. The order is handled in exactly the same way as the indexes for fields. Mid-processing methods are handled somewhat differently. Since they are meant to be invoked between processing of individual fields, the index value assigned to a method indicates the index number of the field after which the method should be invoked. This allows a developer to interleave field processing and specialized methods in a logical and easy to read fashion.

The specialized processing methods differ in structure depending on whether they are used during serialization or in deserialization.

Methods invoked during serialization have the ability to return data to be processed by the serializer whereas those run during deserialization are given access to the raw data. To illustrate the difference:

```csharp
[<packingAttribute>($index$)]
object <packMethodName>() { ... }

[<unpackingAttribute>($index$)]
void <unpackMethodName>(Packer packer, object rawData)
{ ... }
```

Notice that “packing methods” are designed to return data for processing by the packer whereas “unpacking methods” are given access to the packer and raw data for specialized processing. The structure was designed to maximize the flexibility of message implementation by providing developers more direct access to the underlying message conversion should they need it. During serialization, this effectively allows them to dynamically add data that cannot be effectively stored in message fields and in deserialization provides tools to directly work with the compact representation of the message. While this might seem like a very complicated set of tools, one must realize that the vast majority of message architectures can be implemented without many of them. They are provided to improve the flexibility of the framework to handle some of the more obscure or difficult designs that may be encountered.

### 4.3 Inheritance

One last feature built into the messaging definitions is the ability to carry structure through the chain of inheritance. This allows a core object to be defined that contains all of the fields and structure common to a wide range of messages. Each of these messages can then build on the single base object, reducing code complexity and volume while centralizing the design for common objects. Inheritance is automatically handled by the system, so no special considerations are needed when defining derived
messages. The only stipulation is that the indexing of the new packable members and processing methods in the derived classes not conflict with those of the base class. For example:

```csharp
[Packable]
public class BaseMessage
{
    [PackableMember(0)]
    public int _field0;
}

[Packable]
public class PostMessage : BaseMessage
{
    [PackableMember(1)]
    public int _field1;
}

[Packable]
public class PreMessage : BaseMessage
{
    [PackableMember(-1)]
    public int _field_1;
}
```

In the above example, `BaseMessage` specifies a single field to be serialized. The two derived classes `PostMessage` and `PreMessage` add fields that would be processed after and before the '_field0' respectively. Figure 6 shows the results of serialization. Addition of processing methods is just as simple using the corresponding attributes.

![Figure 6. Memory layout of inherited messages.](image)

5. PERFORMANCE

Not only is the RCF extremely compact, but a great emphasis was also placed on the system performance. This includes both the serialization processes as well as the multi-threaded aspects. A number of pieces of the RCF are multi-threaded or asynchronous in nature, often to avoid potential dead-locks and other slowdowns. This is necessary for responsive behavior as any distributed system will need to deal with multiple streams of data simultaneously. Since the asynchronous and event-driven model is built into the RCF, a developer can instead focus on getting the best performance out of each individual section, rather than trying to streamline a single process. The event-driven model also simplifies application development as they can be written as highly modular components with extremely flexible structure. This is directly related to the centralized data routing and interpretation discussed earlier.

The serialization library is the component that received the greatest attention to optimization as data conversion is generally the slowest and most costly operation in any communications system. The resulting implementation is highly streamlined and extremely efficient, imposing very little overhead on application performance. Both serialization and deserialization were tested thoroughly to determine and remove the bottlenecks. Since JAUS .NET acted as a demonstration of the RCF’s capabilities, one of the most common messages was used for testing.

The `ReportGlobalPose` message was selected for two reasons. First, it is a common message used in JAUS systems and contains potentially time-sensitive data. Second and more importantly, it contains a large number of optional fields as previously discussed. As such, it makes an ideal candidate for testing mid-process method invocation as well as numerous small conversions. Three separate classes of tests were performed as illustrated in Figures 6-1 through 6-3 at the end of the chapter. These included simplifying packing to verify proper message conversion, more complete packing operations as would be performed by the message handler, and finally unpacking operations. The last two tests closely approximate the speed at which the `JausMessageHandler` would operate without heavy system load and show the best-case performance. All tests were performed on an Intel Core 2 Duo 2.13 GHz machine with 4 GB of RAM.

When the RCF was first implemented, it used a number of inefficient approaches that were discussed in detail. During the early stages this was of little concern since the focus was on implementing the serialization algorithm properly. However, after the first speed tests were performed, further development shifted to optimizing the performance. Initially serialization of a global pose message required approximately 2.5 ms to complete. Addition of the internal hashtable for known data types and the introduction of dynamic methods reduced this drastically to an average of about 40 µs. The final step of implementing a high efficiency data structure to hold the raw data further reduced this time to only 22 µs, yielding a net performance increase of 110 fold. Under optimal conditions this number has dropped as low as 19 µs, though this is highly dependent on current system load.

Deserialization with the RCF is also extremely fast, though it is somewhat slower than the serialization. This is likely due to the higher cost of bit conversions back into the original data types. Even so, deserialization of a global pose message still averages only 52 µs.

The test shown in Error! Reference source not found. was introduced to help quantify the additional overhead of the intermediate packing operations required for JAUS messages. Notice that it eliminates the intermediate step of packing just the message contents and creating a raw message. This reduces the processing time to just that required for serialization of the header and contents without the overhead of copying the raw contents into the final buffer. Average serialization time for this slightly reduced test averaged 20-21 µs, a miniscule difference when compared to the more complete approach. The time difference is essentially a measure of how long it takes for the custom high efficiency data buffer to copy the intermediate contents into the final buffer. The data buffer is extremely fast and introduces very little overhead to standard conversion operations.

One of the concerns expressed by many developers is the performance impact of a managed environment such as .NET. To investigate this issue, serialization of several messages was tested using both JAUS .NET and an older, native implementation that has been in use for several years. Table 1 shows the average time for processing of ten thousand of each message.
Table 1. Serialization performance of native vs. managed environments.

<table>
<thead>
<tr>
<th>Message</th>
<th>JAUS .NET (μs)</th>
<th>Native (μs)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReportGlobalPose</td>
<td>21.64</td>
<td>3.69</td>
<td>5.86</td>
</tr>
<tr>
<td>ReportWrenchEffort</td>
<td>21.44</td>
<td>4.44</td>
<td>4.83</td>
</tr>
<tr>
<td>ReportCameraCapabilities</td>
<td>24.17</td>
<td>4.39</td>
<td>5.51</td>
</tr>
<tr>
<td>ReportCameraPose</td>
<td>22.40</td>
<td>4.17</td>
<td>5.37</td>
</tr>
</tbody>
</table>

It comes as no surprise that the processing time in the managed .NET environment is consistently slower than its native counterpart. However, closer inspection of the data reveals that on average, the managed implementation is only 5-6 times slower than the native libraries. Considering that the times are relatively low in both environments, the performance impact of serialization is nearly negligible in many systems. When weighed against the ease of development using the Reflective Communications Framework and the greatly improved flexibility of design, the slight overhead required to run in a managed environment becomes a non-issue for most systems.

6. ACKNOWLEDGMENTS
Our thanks to ACM SIGCHI for allowing us to modify templates they had developed.

7. REFERENCES