The Systems Biology Workbench Concept Demonstrator: Design and Implementation

Herbert Sauro, Andrew Finney, Michael Hucka, Hamid Bolouri
{hsauro,afinney,mhucka,hbolouri}@cds.caltech.edu
Systems Biology Workbench Development Group
ERATO Kitano Systems Biology Project
Control and Dynamical Systems, MC 107-81
California Institute of Technology, Pasadena, CA 91125, USA
http://www.cds.caltech.edu/erato

Principal Investigators: John Doyle and Hiroaki Kitano

May 9, 2001

Contents

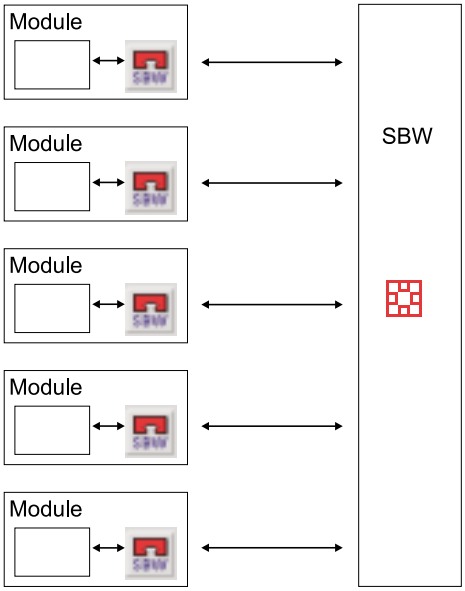

Figure 1: *Diagram of overall architecture of SBW.*

1 Introduction

The Systems Biology Workbench (SBW) is an architecture which allows computational resources to be shared. The modules are the applications which provide the sharable computational resources and the broker is a mediator that allows modules to communicate with each other. A much fuller discussion of this and other related issues can be found in Hucka et al. (2001).

This document describes an experimental implementation of the broker and a set of test modules that have been developed using using Delphi 5.0. This implementation is fully functional within the limits described in this document, but is a subset of the SBW as described by Hucka et al. (2001) and Finney et al. (2001). The current implementation consists of four components:

Broker The current implementation is only 1450 lines long. It was written in such a way that it resides in the background and is visible only in the Windows Tray (see Figure 2). Right clicking over the tray icon generates a simple pop-up menu. This version of the broker is an experimental application and represents a subset of the proposed broker (Hucka et al., 2001; Finney et al., 2001). The final and full implementation of the broker will be done in Java.

Trig This is a non-visible service module which supplies trigonometric functions. This module was written purely for testing purposes and is not meant to represent a realistic module.

Graph2D This is a Delphi form module which provides simple 2-D graphing capability. This module was written purely for testing purposes and is not meant to represent a realistic module.

Inspector This is a form module which can be used to 'inspect' other modules that are attached to the broker. In addition, Inspector allows a user to test the Trig and Graph2D modules (see Figure 3).

The client-side code amounts to about 2000 lines of source code. This again is not an excessive amount of coding. In all, the code required to implement the broker and client side logic is quite modest.

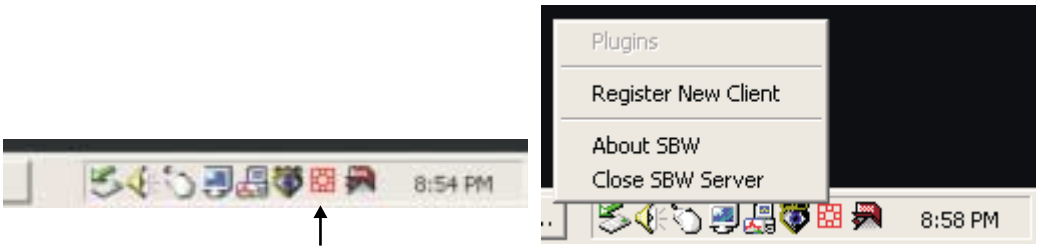

Figure 2: *Broker Icon in Windows Tray.*

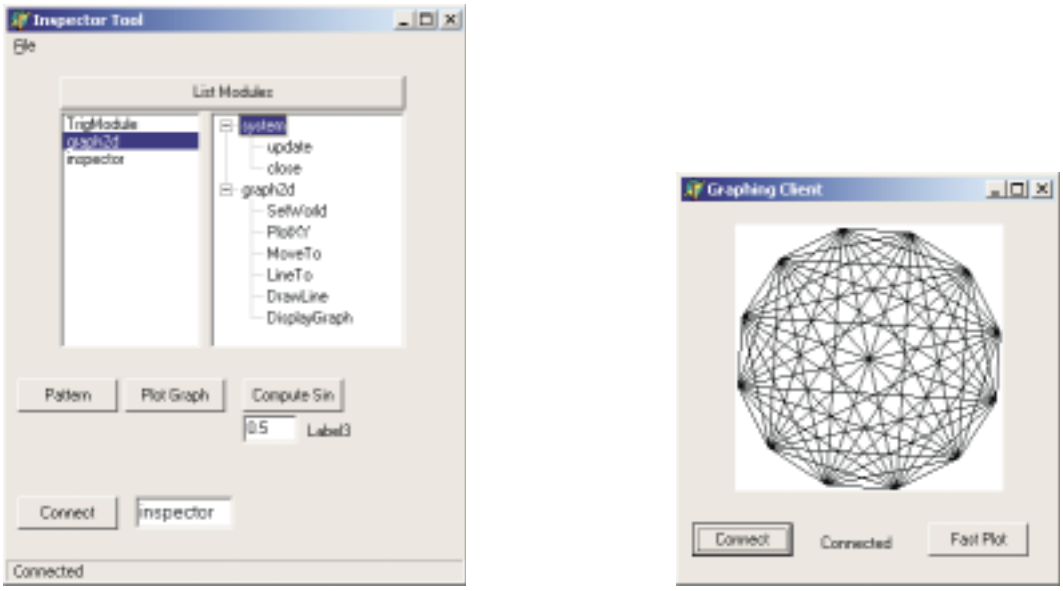

Figure 3: *Inspector and 2D Graph Modules.*

2 Messaging System

2.1 Underlying Transport Medium

The underlying transport medium in this architectural option is simple TCP/IP sockets.

2.2 Module Handles

All modules are assigned a numeric identification handle; thus, references to DestId and SrcId in this document refer to this handle. The handle is generated when a module makes its initial connection with the SBW broker or when SBW starts a module and makes a connection. The broker itself has its own publically reserved handle which allows modules to make requests to services provided by the broker. When a module wishes to communicate to another module, it does so by sending a message through the broker. The message will contain the destination handle which the broker will use to route the message onto the appropriate module.

2.3 Messaging Format

Communication between modules and the broker is via so-called messages. These are simple sequences of binary data. Since the most common microprocessors in use today are the Intel and Intel-compatible

microprocessors we decided to store the data in *Intel-ordered* format. This ordering is sometimes called Little-Endian after the Swift's Gulliver's book Travels, where citizens of Lilliput would break their hard-boiled eggs only at the little ends. Little-Endian specifies that the least significant byte is stored in the lowest-memory address, which is the address of the data.

There are four basic message types: messages which represent blocking calls to methods in other modules or to the broker itself, messages which represent non-blocking calls to methods in other modules or the broker itself, messages which represent replies to earlier messages, and messages which represent error conditions as a result of poorly formatted messages or exceptions which occur in modules.

2.3.1 Call and Send Messages

These messages come in two varieties, `send` (non-blocking) and `call` (blocking). Both types of message have the same internal structure. What distinguishes the two is the value of the message type byte; see below.

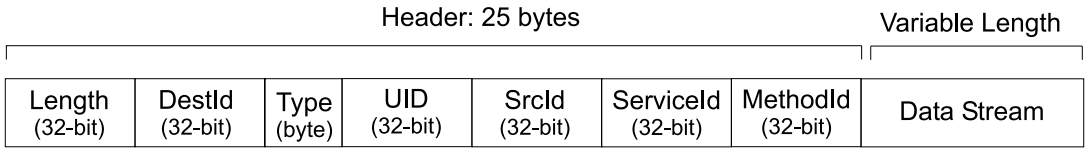

Figure 4: *Structure of the Send/Call Message.*

The fields in a call/send message have the following meanings:

Length 4 bytes: Length of the message in bytes, including the length integer itself.

DestId 4 bytes: An identifier which indicates the destination module for this message.

Type Indicates whether the message is a call, send, reply or an error condition

Reply Message	0
Send Message	1
Call Message	2
Error Message	3

UID A unique identifier associated with this message. A corresponding reply will have the same UID (Unique identifier) and can be used to match a reply to the original sender.

SrcId: 4 bytes: An identifier which indicates the source module for this message.

ServiceId Indicates the required service.

MethodId Indicates the particular method in the service.

Data Stream A data stream containing the arguments required by the method.

2.3.2 Reply Messages

A reply messages is sent in response to a call message. Its sole purpose is to deliver raw data to the recipient.

The format of the first thirteen bytes of a reply message is identical to a calling message. All remaining data in the reply message is composed of a data stream.

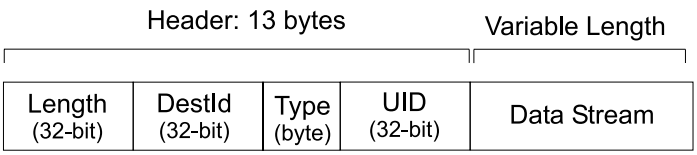

Figure 5: *Structure of the Reply Message.*

2.3.3 Error Messages

Error messages are sent in response to an error condition originating either as a result of a badly formatted message or as a result of an exception in the method which was meant to service the message.

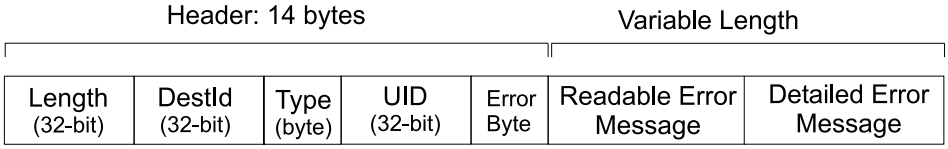

Figure 6: *Structure of the Reply Message*

2.3.4 Optimisation Issues

In the current proposal, many of the fields in the message header are 32-bit integers. In some instances this is necessary, for example the length field. However, the service and method ids may not need to be 32-bit and an opportunity for optimisation may be to reduce these fields to 16-bit. For messages with large payloads, this optimization is probably not important; however, for short messages it might make a significant impact.

2.4 Data Streams

`Data streams` are made from a sequence of `data items`. The end of a data stream is always terminated with a `0xFF` character. Data streams may be empty, in which case they only contain the terminator character. The following section details the different types of allowable data items that data streams can be composed of.

2.5 Data Item Formats

This section describes the data format of the data items that can be carried by data streams.

Each data item is proceeded by a data type byte. The current version supports a number of basic data types, which include

Data Type	Name of Data Type Byte	Value of Data Type Byte
`Byte`	`dtByte`	0
`Integer`	`dtInteger`	1
`Double`	`dtDouble`	2
`Boolean`	`dtBoolean`	3
`String`	`dtString`	4
`Array`	`dtArray`	5
`List`	`dtList`	6

Table 1: *Data Types.*

2.5.1 Byte

Bytes start with a byte code (`dtByte`) indicating a byte type. This is then followed by a 8 bit byte value.

2.5.2 Integers

Integers start with a byte code (`dtInteger`) indicating an integer type. This is then followed by a signed 32 bit integer value in Intel-byte order which has the range -2147483648 to 2147483647.

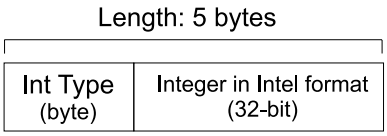

Figure 7: *Integer Data Type*

2.5.3 Double

Double values start with a byte code (`dtDouble`) indicating a double type. This is then followed by a floating-point value stored in standard **IEEE standard 754 double 64-bit** format, that is 1-bit sign, 11-bit base 2 exponent and 52-bit fraction in Intel-byte order.

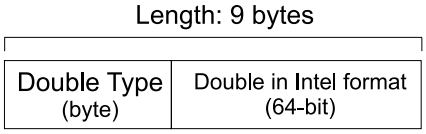

Figure 8: *Double Data Type*

2.5.4 Boolean

Boolean values start with a byte code (`dtBoolean`) indicating a boolean type. This is then followed by a further byte indicate the value of the boolean. A byte value of zero indicates `False` and a value of one indicates `True`.

2.5.5 String

String values start with a byte code (`dtString`) indicating a string type. This is then followed by an unsigned integer denoting the number of bytes in the string. The remainder of the data consists of the sequence of characters that make up the string. Note that the string is *not* null terminated.

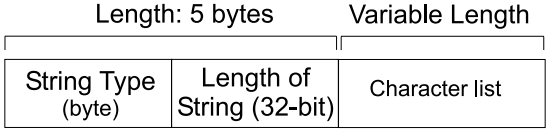

Figure 9: *String Data Type*

2.5.6 Arrays

Arrays are multi-dimensional arrays of arbitrary size containing homogeneous data. Arrays start with a header made up of one byte indicating the data type stored in the array, and an integer indicating the number of dimensions, followed by a sequence of integers, one for each dimension, denoting the number of elements in each dimension. The header is therefore $(2 + 4 + 4d)$ bytes long, where d equals the number of dimensions of the array.

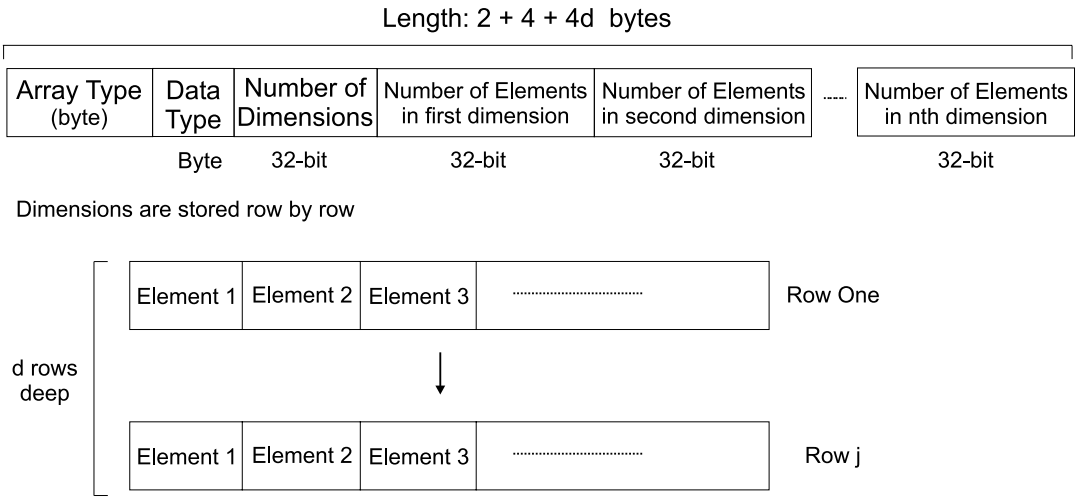

Figure 10: *Array Data Type.*

2.5.7 Lists

Lists are recursively defined structures for storing heterogeneous data. This means that lists can be used to store other lists which allows relatively complex relationships to be represented.

A list is a much simpler structure that an array. A list starts with a list type byte, followed by a 32-bit integer indicating the number of items in the list. Each items in the list can be any of the data types previously described, including a list.

For example, the following are legal list structures:

```
[ 1, 2, 3, 4 ]
[ 1, "ATP", 3.1415, {1, 2, 3} ]
[ [ "S1", "S2", "S3", [ 4, 5, 6 ] ], "k1", "k2" ]
[ ["J1", [ ["X0"], ["S1"] ], "k1S1" ], ["J2", [ ["S1", "S2" ], [ "S3" ] ], "k2S2" ], .... ]
```

Note the nested lists in the third and fourth examples.

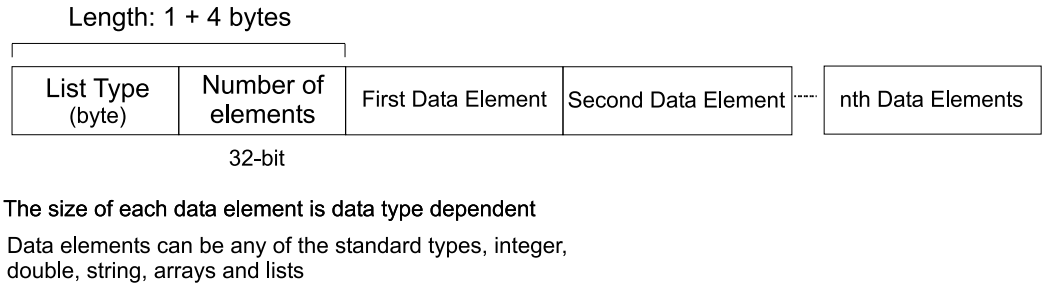

Figure 11: *List Data Type.*

2.6 Summary of Message Format

```
           Message ::= Call/Send Message | Reply Message | Error Message
Initiating Message ::= IHeader DataStream
           IHeader ::= PreHeader SrcId ServiceID MethodId
```

```
Reply Message ::= PreHeader DataStream
Error Message ::= PreHeader String
    PreHeader ::= Length DestId MsgType:Byte UID

       Length ::= Unsigned 32 bit integer
       DestId ::= Unsigned 32 bit integer
      MsgType ::= mtReply | mtSend | mtCall | mtError
          UID ::= Unsigned 32 bit integer
        SrcId ::= Unsigned 32 bit integer
  ServiceIDId ::= Unsigned 32 bit integer
     MethodId ::= Unsigned 32 bit integer

   DataStream ::= 0xFF | ( DataType:Byte DataElement )* 0xFF
     DataType ::= dtInteger | dtDouble | dtString | dtArray |
                  dtList

  DataElement ::=  Integer | Double | String | Array | List
      Integer ::= Signed 32-bit integer
       Double ::= IEEE 64-bit double
       String ::= Length ( character )*
        Array ::= AHeader DimensionList RowList
      AHeader ::= DataType NumberOfDimensions ( NumberOfElements )*
      RowList ::= ( Row )*
          Row ::= ( DataElement )*
        Datum ::= Integer
         List ::= NumberOfElement ( ListOfElements )*
```

3 Broker Issues

There are two ways a connection can be made between the broker and a module.

- A module can initiate a connection with the SBW broker
- The broker can initiate a connection with a module in response to a request from another module

3.1 Module-Initiated Connections

When SBW starts, it first creates a listener thread that is separate from the main thread of the program. The listener thread listens for *all* incoming module connection requests; this is irrespective of whether the connection was initiated by a module or requested by SBW itself. For each module that SBW answers, it spawns a new thread to service that module. Further communication with the module is then carried out within the context of that thread.

3.1.1 Module List

When a new Module connects, one of the first things that SBW does is create a Module object. This object is stored in a globally accessible module list and is used to hold three pieces of information:

```
typedef struct ModuleObject {
   char    *ModuleName
   int     ModuleId
   object  *ThreadRef
} MODULEOBJECT;
```

The **ModuleName** is the name passed from the module to the server at connect time. This allows a mapping to be made between a human readable identifier and the corresponding module handle. The module name

should be unique to avoid conflict with other modules. It has been proposed that for those modules which only use services from other modules and who do not contribute services themselves, be allowed to use a empty name; that is, they are anonymous modules.

The **ModuleId** acts as a unique identification handle for the module. During the life-time of the module, the handle does *not* change. Internally the module Id is an index to the module list, this allows the thread that is managing the particular module to be located very rapidly.

ThreadRef is a reference to the thread which is servicing this particular module.

3.1.2 Connecting Protocol

When the broker detects a connection request from a new module, a module object is created and added to the module list. The module object stores important information related to the module as described previously. When the module object is added to the module list, the position of the module object in the module list is also stored in the module object. This position information is what becomes the module identification handle and to complete the transaction the module index is returned to the module in the form of a single Intel-ordered integer. The module will use this handle in all subsequent calls to the broker. Since the module handle is simply an index to the module list, locating and using the module object field is a fast and simple operation.

In summary, the following sequence of operations occurs during a module initiated connection:

- Module attempts to open a socket to the broker server socket, if successful it passes the name of itself to the broker. This name is in the form of a string of characters, with the first four bytes indicating the length of the string.
- On the broker side, the listener thread picks up the request and reads the name of the module
- The listener thread creates a module thread which will be responsible for all transactions between the module and the broker. The listener thread then creates a module object and enters the necessary information into the module object fields, this includes the module name, and a reference to the module thread which will service this module from now until the module is disconnected from the broker.
- The module object is added to the module list held by the broker and the index at which the module object was stored in the list is returned to the module as a four-byte Intel-ordered integer. This index is also stored with the module object. The index will serve as the module handle during the life-time of the module.

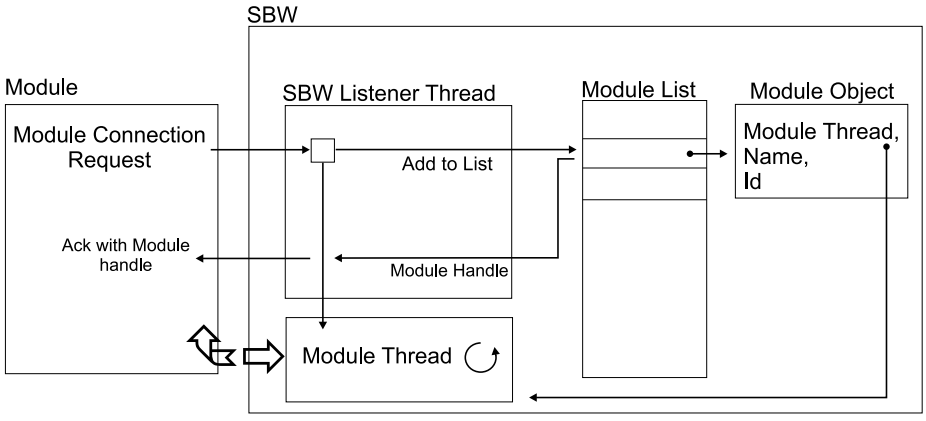

Figure 12: *Connection Interaction for a Module Initiated Connection.*

3.2 Broker-Initiated Connections

If a module requests the services of a module which is not yet running then it is the responsibility of the broker to start that module. During a broker-initiated connection, a very similar process to a module-initiated connection occurs. The only difference is that the broker passes a special command-line flag (`module -register`) to the module which informs the module that it must initiate a connection. Once the connection is made, the broker then proceeds to pass the message it is currently holding to the newly started module.

3.3 Message Protocols and Internal Broker Operations

3.3.1 Broker Module Thread

The broker receives messages from modules through the allocated module threads. The operation of these module threads should be as simple and as efficient as possible since the routing of client messages could be a potential bottle-neck in the design.

A module thread only has two functions. First, the thread is able to determine whether the message is for another module or whether the message is destined for broker itself, the so-called broker service messages. The information required to make this decision is stored in the second entry in the message stream. If the message is for another module, the thread simply reads the remainder of the message and passes the message on to the destination module. The thread does not carry out any other inspection nor does it attempt to alter the message in any way. The only thing that the thread must actively do is use the destination handle as an index to the module thread list. From this list, it obtains the destination thread and hence the destination socket connection.

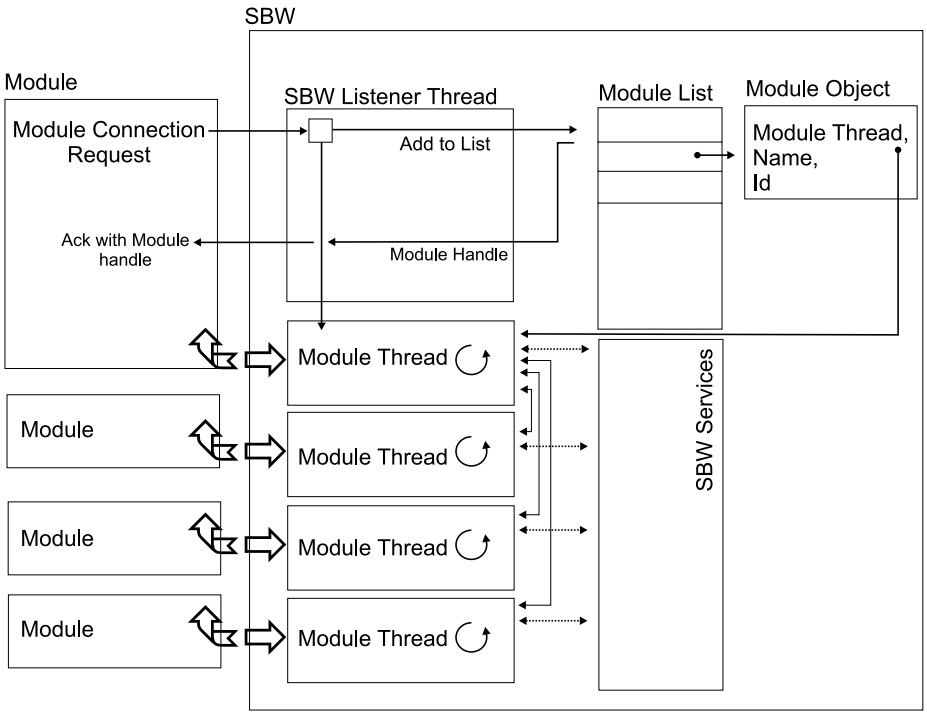

Figure 13: *Server Operation.*

3.3.2 Broker Services

As mentioned in the previous section, messages originating from modules may be sent for the specific attention of the broker itself. In order to make this work, the Broker has been assigned a module handle of '-1'. Thus, to send a message specifically to the broker, simply use a destination handle of '-1' (or the named constant

`BrokerId`). The reason why the handle has a value of '-1' rather than say zero, is because the module thread list starts at index zero and thus using zero to indicate the broker services would conflict with this use.

BrokerId = -1 Service Handle (not a method)

There are a number of methods supported by the broker, these include:

GetNumberOfModules Returns the number of currently connected Modules.

```
n = sbw.GetNumberOfModules()
```

Args

- None

Result

- NumberOfModules `integer`

GetModuleName Returns the name of the module with given Id.

```
name = sbw.GetModuleName(3)
```

Args

- ModuleId `integer`

Result

- NameOfModule `string`

---

4 Module Messaging

Messaging is a two stage process. In the first stage, a module will send a message to the broker, indicating in the message the destination module. In the second stage, the message is passed from the broker to the destination module. If the message is a call, then two other stages begin which involve constructing a reply and sending it to the broker and back to the module which originated the call.

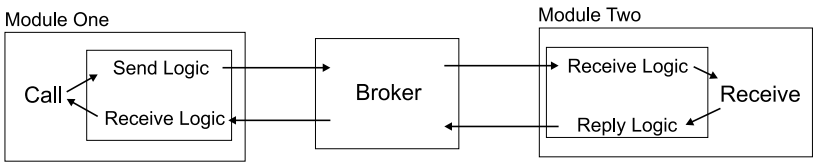

Figure 14: *Broker/Module Message Interaction.*

4.1 Basic Module Architecture

For a given application that wishes to be part of the SBW network of sharable resources, the application must implement client side logic to enable the client to communicate with the broker. Under Delphi, this logic comes in the form of a non-visual component, called the `TModuleComponent`.

The module component consists of four main classes:

ModuleComponent This is the main entry point for applications wishing to use SBW facilities. Under Delphi the module is written in the form of a non-visual component; therefore, to make a Delphi application SBW-aware one simply has to drop a Module component onto one of the forms in the Delphi application. A similar approach could be adopted under Java in the form of a JavaBean.

RecvThread The RecvThread object is responsible for intercepting *all* communications from the broker. RecvThread runs continuously in a thread so that listening for broker messages does not interfere with the normal processing of the application. When the RecvThread receives a message from the broker it spawns a special MessageThread object which has the responsibility for dealing with the particulars of the message. Once the RecvThread has spawned a message thread, it returns to its normal mode of listening for SBW messages. Thus, each time a message is received by RecvThread, a message thread is created. However, to eliminate the performance hit which is suffered each time a thread is spawned, in the actual implementation, a thread pool is maintained and spawning a message thread instead involves requesting a existing suspended thread from a thread pool. When message threads are no longer required they are returned to the thread pool for future use. Tests have shown that this simple use of a thread pool improves message transport roughly six-fold compared to no thread pooling.

MessageThread The message thread is responsible for decoding messages which it receives from RecvThread. The message thread will respond differently depending on whether the message is a call/send, reply or error condition. Once the message thread has completed its task it is returned to the message thread pool.

ServiceList A list of services and methods provided by this module.

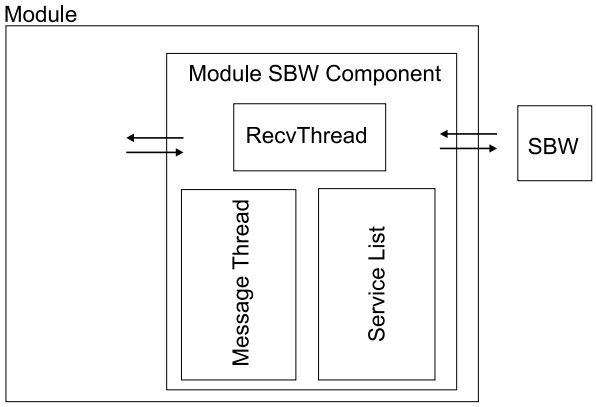

Figure 15: *Module Structure.*

4.2 Sending Calls from a module to the broker

The first stage to sending a call message is to send the message to the broker. The diagram below describes this operation.

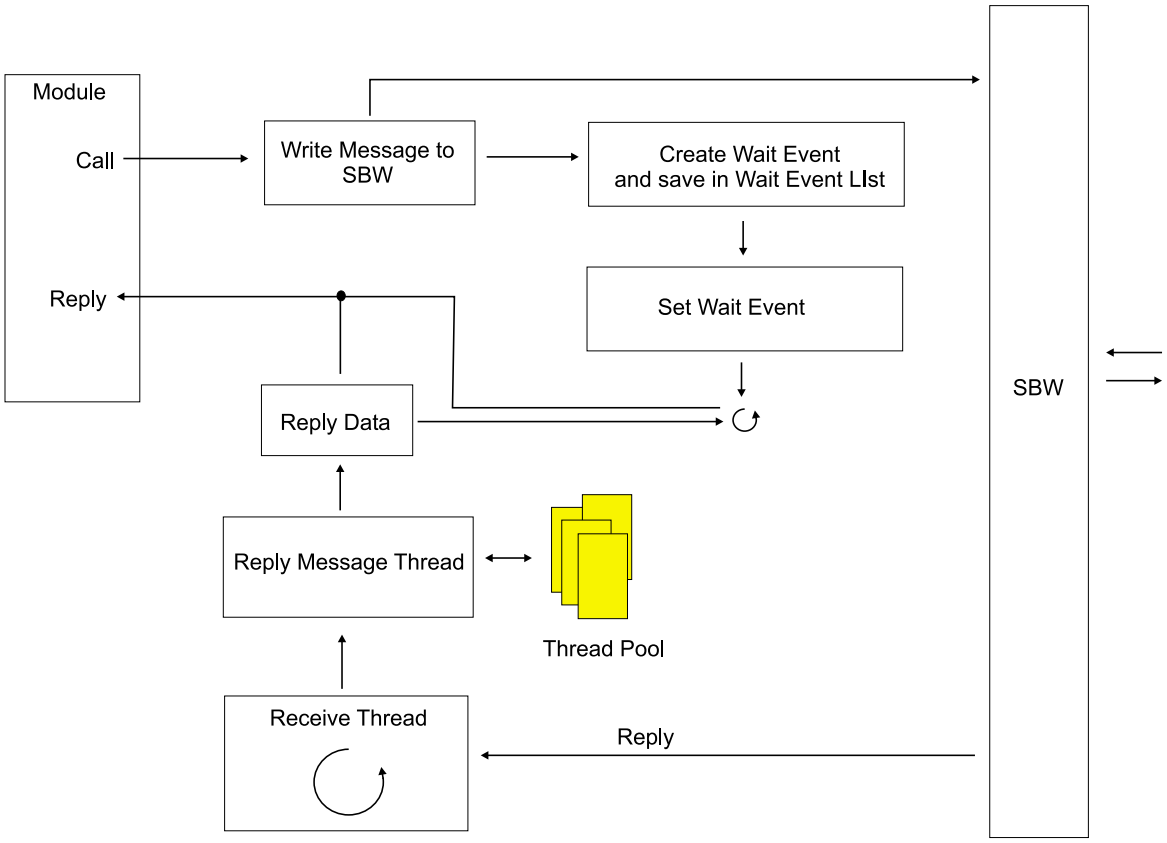

Figure 16: *Dealing with a Call Message to the Broker.*

The sequence of events is as follows;

- The module makes a call using the `sbw.call()` method
- The message is sent to the broker and is received by the main listening thread in the broker.
- The module creates a Wait Event and adds it to the list of wait events stored in the RecvThread object.
- The module waits using the Wait Event.
- Some time later, the broker returns a reply to the module.
- The RecvThread receives a message and recruits a message thread from the thread pool and passes the complete message to the message thread. RecvThread then resumes its normal mode of listening for messages from the broker.
- The message thread interprets the message as a reply, but at this stage does not know who the reply is for. The message thread searches the EventList for the corresponding call which initiated this reply. When it locates the event (an exception results if it cannot find the event), it saves the message and performs a SetEvent call on the appropriate waiting event. This causes the wait event to resume execution where is retrieves the stored message and returns it to the original caller for processing.
- Once the reply has been finally dispatched, the message thread is returned to the thread pool.

To the developer, a call message looks just like a normal function call, e.g. `x = sbw.call (...)`, and the interaction between the broker and module are hidden.

4.3 Receiving Calls from the Broker

When a module sends a call message to the broker, the broker will route the message to the appropriate destination module. The diagram below illustrates the sequence of events which occur when the broker passes a call message to a module. The sequence of events described below occurs within the module.

- The RecvThread intercepts the message from the broker, recruits a message thread, passes the message to the message thread and returns to its normal mode of listening for messages from the broker.
- The message thread examines the message to determine if the message is a call, a send, a reply or an error condition. If the message is a call, the thread routes the message to the call procedure. In the call procedure one of two things can happen: 1) the call method was previously registered, in which case, the message thread automatically calls the appropriate method in the module; 2) the method was not registered, in this case the message thread fires a general purpose CallEvent procedure which is intercepted by the module. Within the CallEvent procedure the module will call the appropriate method according to the supplied method id. If the message is a reply, then the message thread calls the reply procedure.

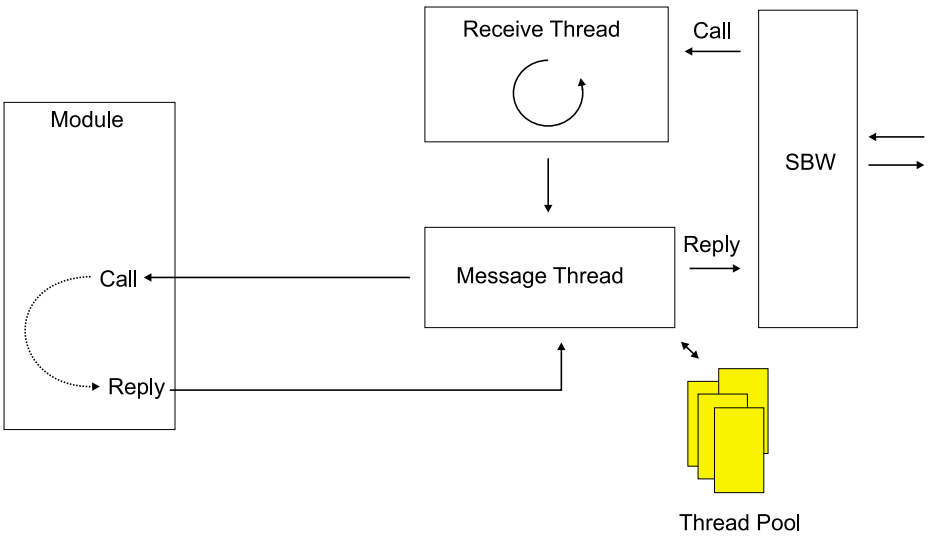

Figure 17: *Dealing with a Call Message from the Broker.*

4.4 Basic API for the Module

Finney et al., (2001) have described a high level API for SBW. This section describes the implementation of a experimental low level API and in no way should it be considered final. All interactions with the SBW server use via a module side SBW object. Under Delphi, you have two options to instantiate this object:

1. Drag the SBW component from the component palette.
2. Instantiate the object at runtime:

```
sbw := TSBW.Create (nil);
```

When created, the SBW object does not have an active connection to the server. To initiate a connection to the server, one must call the `OpenConnection` method, e.g.,

```
sbw.OpenConnection ('MyPlugin', ErrMsg);
```

The following methods are available via the SBW object:

OpenConnection (Name, ErrMsg) Attempt to make a connection with a local SBW. The function returns true if successful, false otherwise. If the call fails to make a connection, an error message can be found in the argument ErrMsg. The first argument, called Name, is the name of the module requesting a connection.

CloseConnection() Close an existing connection. If there is no existing connection, this function does nothing.

Connected Returns true if connected to SBW, else returns false.

Call Calls a method in a remote module and returns the result to the caller. Note that this is a blocking call.

```
res = sbw.Call(Math, Trig, sin, [30.4])
```

Args

- ModuleId `integer`
- ServiceId `integer`
- MethodId `integer`
- Argument List `Delphi Open Array`

Result

- Returned Data `TComType`

RegisterService Register a service in the current module and returns a service Id.

```
Id = sbw.RegisterService('bifurcation')
```

Args

- Name of Service `string`

Result

- ServiceId `integer`

RegisterMethod Add a method name to a particular service in the current module.

```
Id = sbw.RegisterMethod('trig', 'sin', MethodPtr)
```

Args

- Name of Service `string`
- Name of Method `string`
- MethodPtr `MethodPtr Type`

Result

- Method Id `integer`

GetServiceId Returns the Service Id for a given Module Id and a Service Name.

```
Id = sbw.GetServiceId(3, 'trig')
```

Args

- Module Id `integer`

- Service Name `string`

Result

- ServiceId `integer`

GetModuleId Returns the module handle for the specified module name.

```
Id = sbw.GetModuleId('MyModule')
```

Args

- Module Name `string`

Result

- ModuleId `integer`

GetServices Returns a list of supported services for a given module.

```
list = sbw.GetServices('MyModule')
```

Args

- Name of Module `string`

Result

- Number of Services `integer>`
- Name of first Service `string`
- Name of second Service `string`
- Name of ith Service `string`
- ...

GetMethods Returns a list of available method names for a particular module and service.

```
list = sbw.GetMethods('MyModule', 'trig')
```

Args

- Name of Module `string`
- Name of Service `string`

Result

- Number of Methods `integer`
- Name of first Method `string`
- Name of second Method `string`
- Name of ith Method `string`
- up to number of method ...

The SBW method, `RegisterMethod` takes a `MethodPtr` in its third argument. The type for the `MethodPtr` is defined as:

```
TMethodPtr = function (x : PChar) : TComType of object;
```

The argument `x : PChar` is a pointer to the data stream that is being passed to the method. Use the utility routines to extract the relevant data items (see examples).

`TComType` is a variant type which can be used to return a variety of data types to the caller. It is defined as:

```
TComType = record
           ComType : integer;
           str : string;  // Strings not allowed inside a case union
           case integer of
                  dtInteger   : (i : integer);
                  dtDouble    : (d : Double);
                  dtBoolean   : (Bool : Boolean);
                  dtArray     : (Ar : TSBWArray);
           end;
```

This definition is not complete and further types will be added at a future date. When returning data to the caller, it can be convenient to use the `TComType` packaging routines (see example).

Example:

```
function TMyStuff.MySin (x : PChar) : TComType;
begin
  result := ReturnDouble (sin (ReadDouble (x));
end;
```

5 Example Code

The following examples were taken from actual modules.

5.1 Simple Module Providing Computational Services

The following example shows how simple it is to build a module which provides some form of computational service, in this case a simple set of trigonometric functions. The module is built from only three procedures, two procedures which define the trig functions and a start-up procedure which attempts to make a connection with the SBW broker. The startup procedure also then registers the trig functions, supplying the service name, method name and a pointer to the method which will carry out the operation.

`sbw` is the name of the client module object.

```
const MODULE_NAME = 'TrigModule';

function TForm1.MySin (x : PChar) : TComType;
begin
  result := ReturnDouble (sin (ReadDouble (x));
end;

function TForm1.MyCos (x : PChar) : TComType;
begin
  result := ReturnDouble (cos (ReadDouble (x));
end;

procedure TForm1.FormCreate(Sender: TObject);
var ErrMsg : string;
begin
  if sbw.OpenConnection (MODULE_NAME, ErrMsg) then
  else raise Exception.Create ('Failed: ' + ErrMsg);

  sbw.RegisterService ('trig');
  sbw.RegisterMethod ('trig', 'sin(x)', MySin);
  sbw.RegisterMethod ('trig', 'cos(x)', MyCos);
end;
```

Example of a module which makes use of the Trig module.

The first operation is to open a connection to `sbw`. The second stage is to obtain handles to the required services and methods within the Trig module. This mode of accessing services is the lowest form of access, Finney et. al., (2001) have detailed a higher level API which takes a different approach.

```
procedure TForm1.FormCreate(Sender: TObject);
var ErrMsg : string;
begin
  if sbw.OpenConnection ('TestTrig', ErrMsg) then
  else raise Exception.Create ('Failed: ' + ErrMsg);

  try
    TrigModuleId := sbw.GetModuleId ('TrigModule');
    TrigId := sbw.GetServiceId ('TrigModule', 'trig');
    SinId := sbw.GetMethodId ('TrigModule', TrigId, 'sin');
    CosId := sbw.GetMethodId ('TrigModule', TrigId, 'cos');
 except
    on E: ESBWError do
       showmessage ('SBW Exception: Failed to load TrigModule: ' + E.Message);
  end;
end;
```

Once the module has been registered and the remote services handles obtained, we can make a call to the desired service, in the following case, to compute the sine for a particular radian angle.

```
  try
    caption := floattostr (sbw.GetDouble (sbw.Call (TrigModuleId, TrigId, sinId, [5.4])));
  except
    on E: ESBWError do
       showmessage ('Compute Sin: ' + E.message);
  end;
```

Note the use of the utility function, `GetDouble` to extract the result from the data stream returned by the call.

`ESBWError` is a standard SBW Exception. Its message part will contain additional information regarding the nature of the exception. Exception handling, in particular the variety of exceptions that can be handled, will be increased in future releases.

5.2 Utility Routines

Certain utility routines are provided to aid in the extraction of data items from data streams passed to callable module methods and to aid in the packaging of return data from the callable modules.

`ReadInteger (DataPtr), ReadDouble (DataPtr), ReadString (DatePtr)`

These routines will extract a particular data type value from the data stream pointed to by `DataPtr`. After the routine returns with the value, subsequent calls to these routines will retrieve the next data item and so on. An exception is raised if the data in the data stream is not of the appropriate type. Similarly an exception is also raised if an attempt is made to read a data item when no data items remain to be read.

`GetInteger (TComType), GetDouble (TComType), GetString (TComType)`

`sbw.Call()` returns a `TComType`. There are some utility routines which can be used to extract a particular type from `TComTypes`. Thus, `GetDouble` will attempt to extract a double type from a variable of type `TComType`. At exception is raised if the routine fails to locate the expected data type.

`ReturnDouble (double)`

ReturnDouble performs the opposite to `GetDouble`. That is given a double value, `ReturnDouble` will package the double value into a `TComType` ready to returning from a module method call.

References

Finney, A., Sauro, H., Hucka, N., and Bolouri, H. (2001). Programmer's manual for the systems biology workbench (SBW). Available via the World Wide Web at `http://www.cds.caltech.edu/erato/sbw/docs/api`.

Hucka, M., Finney, A., Sauro, H., and Bolouri, H. (2001). Introduction to the Systems Biology WorkBench. Available via the World Wide Web at `http://www.cds.caltech.edu/erato/sbw/docs/api`.