

Protocol design: you get what you put

A penny's worth of advice and a Java generic Protocol class

When designing anything from embedded systems to client/server based systems or GUI APIs you have to know the protocols involved. Programs communicating with each other must do this on a mutually agreed basis. This article tries to sum up experience from protocol design in typical embedded environments. You may never be able to repair a failed design after it has been shipped, so you'd better do it right from the start.

FOREWORD

On June 4, 1996, the maiden flight of the Ariane 5 launcher ended in self-destruction. The active inertial reference system had correctly transmitted error information to the launcher's main computer, where it was interpreted as angular data. The nozzles of the solid boosters were then deflected to such an extreme that the launcher was lost. The computers were later found, and log data was read from EEPROM. A "wrong" piece of software had been reused, the protocols did not match.

When designing anything from embedded systems to client/server based systems or GUI APIs you have to know the protocols involved. Programs communicating with each other must do this on a mutually agreed basis. This article tries to sum up experience from protocol design in typical embedded environments. You may never be able to repair a failed design after it has been shipped, so you'd better do it right from the start.

A PROTOCOL IS AN INTERFACE

When a C function calls another C function your compiler may complain that it sets up a call with "no prototype in scope" if you have forgotten to make a header file. The compiler will also complain if the prototype's definition is not equal to your usage. Basically a protocol verification is done by the compiler, function parameters define the protocol involved. Distribute the functions in a multiprocessor system, and the compiler most probably is of little help. The parameters will be packed into a byte stream, and an integer may not even be of the same size at each side.

ONE PLACE TO LOOK

All participants that need to understand a protocol should find the documentation at a central repository. Any deviation should also be available at the same place. Pull them out of the projects' directories into a protocol directory. If you use a struct to define the protocol, put it into a header file and use `#ifdef's` (if you believe in `ifdef's`) to differentiate between different uses

which need slightly different definitions.

INCLUDE RATIONALE

In the last projects I've participated we have made the protocol description as a web-based system of html files. Descriptions, figures, diagrams, rationale and source code reside in a set of html files. The program sources have been extracted by a simple "literal programming" system. The top level html file is the input to the extractor program, only external hyperlinks decorated with `#clip` are included. The result is browsable documentation and a group of write-protected generated source files. The `.h.inc` or `.java` files (or anything you define) are then used during compilation.

Be sure not to let everyone edit the documentation, the fewer the better. Html based editors often don't protect an open document, adding a top line like "now open by" is better than no such info, even if it has to be kept manually.

Still, the protocol is something that everyone has a meaning about, it should be teamwork. You should act on behalf of that team.

LAYER

When you have the infrastructure in place, you should decide which layer your protocol should cover. Read about the different ISO OSI layers first, but understand them literally. Should your protocol handle:

- collisions on the physical layer,
- sequence control or flow control,
- CRC and checksumming,
- addressing or routing information,
- packeting information, merging and splitting,
- byte order (endianess) reordering,
- primitive data type conversion,
- complex data type conversion,
- or should it contain application layer information only?

It can be fatal to mix the layers: it is better to define one

protocol for each layer than mixing layers into one protocol. Mixing makes error handling and recovery more difficult. These days you can reuse industry standard or international standard protocols at the lower layers, but you would most probably have to define your own at the application layer. Stick to it first, and make no assumptions about the lower layers. Lower layer would then be plug and unplug for you.

A CONSISTENT NAMING SCHEME

The name you give a protocol "command" is quite important. You can use some kind of Hungarian naming scheme and add a postfix to describe the semantics. This way you can embed parts of the message sequence diagram in the name. If a command requires an Ack reply the command could be called *foo_Ack*, if it requires an Ack or a Nack you could call it *foo_AckNack*. If you require a specific reply you could call the command *foo_AckPlus* and the reply *foo_End*.

Spontaneous messages you could call *foo_Event*, or if you make a subscribing scheme they could be called *foo_Subscribed*.

A defined protocol may cause applications to deadlock, so you could make a table of all postfix names and their expected semantics. Make a separate column for blocking properties. A command could be considered blocking if the client will wait forever for a reply. (In this article client initiates and server responds, an example would be a host client and an embedded server). This may be acceptable semantics at the application layer, as traffic may be defined as secure. Exceptions like time-outs from broken cables or crashed servers should still be delivered by a lower layer, and the user may be warned so that she can take proper action, like exiting the application that's waiting forever.

APPLICATION LAYER

At the application layer you should assume that transport at lower layer has been assured so that you can think of it as error free. It should ideally contain logical rather than physical addressing, no flow control, no checksum or retransmission.

Try to find generic patterns instead of solving short term needs. If you need to export curves, don't make one particular protocol command for each type of curve. Make a group out of all curve handling, and have each type of curve export its own credentials like name, scaling, curve axis names and unit etc. The client that is going to use your curve data would then be able to handle this set of commands, and you would be able to invent new type of curves after the old client has been shipped.

Have your server export what it has, or what it is. If you have an embedded controller that exports these curves, make a command in the curve group where the controller exports the list of curves it is hard-coded to handle. An easy way to do this is to let the controller export a text string of comma separated names of curves. After this, all naming of the curves could either be by the index in the list, or they could be accessed by the textual name given - this is more general but

requires more of the controller and network bandwidth. It is a good idea to make a command where the server exports which commands it understands and which versions of each it understands. Two arrays containing command numbers and version numbers would suffice. In this case a newer client is able to use a different strategy to get a defined behavior out of a server.

You should decide what your client should do if the server receives a message it is not coded to handle. If a lower protocol layer has sent this command to a wrong unit, or sent it to two, the best strategy could be to be quiet and not report back to the client, provided you have some other means to verify the lost action. You could however, lit a protocol error LED on the server side.

It may be a good idea to define a subset of the protocol that all connected units must respond to, no matter whether they belong to the problem domain you are initially designing the protocol for, or if future reuse demands a quite different problem domain.

One point is that a command label (the starting identifier) of a byte only is perhaps too short for future extensions.

If you have a client/server architecture, you should consider making a log-on scheme available if the server contains shared resources. This could be quite simple: the key could be a byte only - among trusted users. One command could ask who's logged on, this would be a command that did not supply the key. The server could throw a client out after say, 1 hour.

If you think you have a good design for a protocol, you could envelope older protocols into the new protocol for Mark n+1 of your product. You could then get reuse of some of the newer programs in the old problem domain.

IDL (Interface Definition Language) is especially designed to define object interaction. You can have interface code automatically generated for you by IDL compilers. However, we are working with everything from PIC and LON processors to 32 bits DSPs and x86 machines with the lot of runtime systems - so looking into (several meanings of) IDL will be a future task.

EXPORT DATA OF YOUR DATA

If you need to export data, and want to have a meta description of it, here's a scheme for it that we have used with success in embedded applications:

- Describe the server's exported data structure in a header file (assuming C).
- Use this file in your embedded program and during compilation.
- Make a tool that packs the essentials of the header file into a struct.
- Include some id of how your compiler does struct layout, basically it's not portable!
- Link this struct into your embedded code.
- You now have a fingerprint of the used data.
- Have your embedded server send off this structure (as a file?).

- Build a parser on the client side, parse the header file.
- Now you know names, datatypes, offset addresses and other attributes that you may have included as comments in the original h-file.

With this homebrewed "xml" a client always has a 100% correct view of the data as used by the unknown server's compiled code.

LOWER LAYER FUNCTIONS AT APPLICATION LAYER

If you have a multiple window client, and several windows use the protocol, it may be a good idea to include some information about message origin (other than the lower layer sender address), so that its reply could be routed to the correct window. This saves a complicated state machine. The cost is that a byte or two have to travel to the server and back without doing any good at the server.

Sometimes you may need to send large amounts of data. In this case make three values that keep track of application layer segmenting: numBytesSent, numBytesLeft, numBytesNow and then the byte array. The sum of the three would always be the total length of the data, and the segment is explicitly defined. This is just one example of how the protocol can help you make a client not mirror the server's state, a technique called state-less design. The more you have of it, the better.

It is a good idea to include a *Ping.Plus* message, requiring a *Ping.End* reply, also at the application layer. Within it you could include a few elements to verify lower layer behavior (not all is Catch-22). We have included several fields, one has a *float*, the server is then required to add 1.00 before it replies. Try this on a Texas DSP which uses an odd float format, and conversion error is easily seen. A ping also verifies the connection, of course.

LOWER LAYERS

If you design the lower layer protocol you should ensure that it is able to carry all types of behavior demanded by the application layer.

If the application layer is not only client/server with mandatory command/reply pairs you should perhaps not use a lower layer where the outgoing packet includes some information from the last incoming packet. Incoming and outgoing lower layer data streams should be independent.

The behavior of the system if a lower layer maximum packet size is 1500 bytes or 100 bytes could differ - if for example a log-on message is 101 bytes it would fit into one or two packets. If two such log-on messages arrived from two clients you're bound for trouble if you are not able to handle more than one datastream at the lower layer.

PACKING AND UNPACKING

Some languages, like occam and Limbo have protocol primitives defined into the languages. This way both compiler and run-time system will help with correct

usage, and variable-sized arrays will be taken care of. This must be hand-coded into C. But you could design a scheme to automatically handle variable length arrays. With C you could always place an "int32" in front of every variable length array to define its length. Delving into Perl and you could have scripts generate code to pack and unpack the structs.

Packing and unpacking could be function driven with one function per protocol entry. The sender would use that platform's *pack* function and the receiver its platform's *unpack* function. Alternatively generic pack and unpack functions could be made table driven. This would yield only one pack and unpack function at each side. This table could also be generated by a script. We have used both schemes with success.

We have also used a scheme where marshalling (=packing, unpacking etc.) has been done on error-free application level byte-streams. The protocol is in this case defined in a table, and the same thread code may be used for both unpacking (input) and packing (output), only started as being either input or output. This scheme yields a "context free" system, where no lower-layer software is needed to define protocol element boundaries.

VERIFICATION & TESTING

You could verify that a protocol is correctly defined by writing a model of it. *Spin* is a tool that may be used for this, it verifies *Promela* descriptions. The concept is based on process algebras which may be described as machine-analyzable languages to describe interacting concurrent processes. This is, however, far outside the scope of this article.

One of the good things about strict definitions of protocols, is that you can pipe the data between unnamed concurrent processes easier. Whether you talk with a test program or a real application is irrelevant. This does of course imply that there is no public variables to modify. You can use protocol definitions between all concurrent processes - this will take systematic testing further down into your system.

PROTOCOL.JAVA ON JCSP

I have included an experimental Java generic *Protocol* class. It is able to carry any Java object, even the command label is a Java Object, not a globally understood number. In the example the protocol is carried by a *One2OneChannel* between *MyConsumer* and *MyProducer*, which are instances of *CSPProcess*, as defined in the Java Communicating Sequential Processes (JCSP) API by Peter Welch et al. at University of Kent at Canterbury, UK. CSP is a process algebra. The example shows Java multithreading without *wait*, *notify*, *notifyAll* and *synchronized*, they are hidden and forgotten inside JCSP. A nightmare is taken out of threads programming. Listing 1 defines the protocol labels as a type-safe enumeration scheme. The Protocol class has sender-side and receiver-side methods, see listing 2. MyProducer and MyConsumer (listing 3,4) engage in several synchronous rendezvous. MyProducer can reuse his objects immediately after the *transit* call, which includes double buffer

AD IMAGE MEDIA

PROTOCOL

ing. The program that starts the concurrent processes is Test (listing 5) and the Java *main* is in listing 6.

CONCLUSION

Programmers who define protocols try to do it right. The article lists some design suggestions born out of

```
final class Label{
    public final static int MAX_ELEMENTS = 2;
    public final static Label Integer_NoAck = new Label();
    public final static Label String_Integer_NoAck = new Label();
    public final static Label Flush_NoAck = new Label();
    public final static Label String_NoAck = new Label();
    public final static Label OutputStreamWriter_NoAck = new
Label();
    public final static Label Exception_NoAck = new Label();
}
```

Listing 1

```
public class Protocol{
    private int    numFilled = 0;
    private int    numTaken = 0;
    private Object label;
    private Object[] elements;
    private int    elementCapacity;
    private int    noOfBuffers = 2;
    private Protocol[] buffer;
    private int    iOfBuffer = -1;

    public Protocol() {init (2);}
    public Protocol (int elementCapacity){init (elementCapacity);}
    public void attach_Label (Object Label){
        label = Label;
        numFilled = 0;
    }
    public void fill_Element (Object Element){
        elements[numFilled] = Element;
        numFilled = numFilled + 1;
    }
    public void set_NumFilled (int numFilled){this.numFilled =
numFilled;}
    public void set_NumTaken (int numTaken) {this.numTaken =
numTaken;}
    public void transit (ChannelOutput channel){
        if (iOfBuffer == -1){
            buffer = new Protocol [noOfBuffers];
            for (int i=0; i < noOfBuffers; i++){
                buffer[i] = new Protocol(elementCapacity);
            }
            iOfBuffer = 0;
        }
        buffer[iOfBuffer].attach_Label (label);

        for (int i=0; i < numFilled; i++){
            buffer[iOfBuffer].fill_Element (elements[i]);
        }
        channel.write (buffer[iOfBuffer]);
        iOfBuffer = (iOfBuffer + 1) % noOfBuffers;
    }
    public Object read_Label(){
        numTaken = 0;
        return label;
    }
    public Object take_Element(){
        numTaken = numTaken+1;
        return elements [numTaken-1];
    }
    public boolean got_All() {return (numTaken == numFilled);}
    public int get_NumFilled(){return (numFilled);}
    public int get_NumTaken() {return (numTaken);}
    private void init (int elementCapacity){
        label = new Object();
        elements = new Object [elementCapacity];
        this.elementCapacity = elementCapacity;
    }
}
```

Listing 2

several close encounter protocol design experiences. A warning of mixing of protocol layers is issued. A generic Java protocol class to be used on top of JCSP is also shown. ■

Øyvind Teig is Senior Development engineer at Navia Maritime AS, division Autronica (<http://www.autronica.no/>). He has worked with embedded systems more than 20 years, and is especially interested in real-time language issues.

REFERENCES

- Ariane 5: Flight 501 failure, Inquiry board's report, <http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>
- Spin Model Checking - Reliable design of concurrent software, Holzmann, , Dr.Dobb's Journal, October 1997. Spin available at <http://netlib.bell-labs.com/netlib/spin/>
- JCSP library, Free download from <http://www.cs.ukc.ac.uk/projects/ofa/jcsp/>.

```
import java.util.*;
import java.io.*;
import jcsp.pawt.*;
import jcsp.lang.*;

class MyProducer implements CSProcess{
    private final ChannelOutput out;
    private Protocol prot;
    public MyProducer (ChannelOutput out_){
        out = out_;
        prot = new Protocol();
    }
    public void run(){
        try{
            FileOutputStream fileOutputStream =
                new FileOutputStream ("OutputFile.txt", false);
            OutputStreamWriter outputStreamWriter =
                new OutputStreamWriter (fileOutputStream);
            prot.attach_Label (Label.OutputStreamWriter_NoAck);
            prot.fill_Element (outputStreamWriter);
            prot.transit (out);
        }
        catch (FileNotFoundException e){
            prot.attach_Label (Label.Exception_NoAck);
            prot.fill_Element (e);
            prot.transit (out);
        }
        catch (IOException e){
            prot.attach_Label (Label.Exception_NoAck);
            prot.fill_Element (e);
            prot.transit (out);
        }
        for (int i=0; i < 1000000000; i++){
            Integer value = new Integer(i);
            prot.attach_Label (Label.Integer_NoAck);
            prot.fill_Element (value);
            prot.transit (out);
            prot.attach_Label (Label.String_Integer_NoAck);
            prot.fill_Element ("The meaning of life is: ");
            prot.fill_Element (value);
            prot.transit (out);
        }
        prot.attach_Label (Label.Flush_NoAck);
        prot.transit (out);
    }
}
```

Listing 3

```

class MyConsumer implements CSProcess{
    private final ChannelInput in;
    private Protocol prot;
    public MyConsumer (ChannelInput in_){
        in = in_;
        prot = new Protocol();
    }
    private void verify_getAll (String label){
        if (!protgot_All()){
            System.out.print (" Error: not read all with " + label);
            System.out.print (" numFilled: " + prot.get_NumFilled());
            System.out.println (" numTaken: " + prot.get_NumTaken());
        }
    }
    public void run(){
        Label label = new Label();
        boolean running = true;
        while (running){
            prot = (Protocol) in.read();
            label = (Label) prot.read_Label();
            if (label == Label.Integer_NoAck){
                int now = ((Integer) prottake_Element()).intValue();
                verify_getAll ("Label.Integer_NoAck");
            }
            else if (label == Label.String_Integer_NoAck){
                String text = (String) prottake_Element();
                String number = ((Integer) prottake_Element()).toString();
                verify_getAll ("Label.String_Integer_NoAck");
                System.out.println (text + number);
            }
            else if (label == Label.OutputStreamWriter_NoAck){
                OutputStreamWriter outputStreamWriter =
                    (OutputStreamWriter) prottake_Element();
                verify_getAll ("Label.OutputStreamWriter_NoAck");
                try{
                    outputStreamWriter.write ("Hello!");
                    outputStreamWriter.flush();
                    System.out.println ("Wrote 'Hello' to disk");
                }
                catch (IOException e){
                    System.out.println ("Could not write to disk
(IOException)");}
            }
            else if (label == Label.Exception_NoAck){
                Exception e = (Exception) prottake_Element();
                verify_getAll ("Label.Exception_NoAck");
                System.out.println (e.toString());
            }
            else if (label == Label.Flush_NoAck){
                System.out.println ("Label.Flush_NoAck");
                verify_getAll ("Label.Flush_NoAck");
                running = false;
            }
            else{
                System.out.println ("Label not known, stopping.");
                running = false;
            }
        }
    }
}

```

Listing 4

```

class Test implements CSProcess{
    public Test(){
        final One2OneChannel c = new One2OneChannel ();
        MyProducer prod = new MyProducer(c);
        MyConsumer cons = new MyConsumer(c);
        new Parallel (new CSProcess[]{prod,cons}).run();
    }
    public void run(){
    }
}

```

Listing 5

```

public class Protocol_test{
    public static void main (String argv[]){new Test().run();}
}

```

Listing 6