The Ouroboros Pattern

Ada Lovelace & Iang
(southern) Winter 2012

Abstract. We present a protocol testing tool that we call the **Ouroboros Pattern**. We believe this to be novel, and worth documenting with all its background logic. Ouroboros builds upon the benefits of object-oriented network programming and introduces positive fuzz loopback testing within the network object. When combined, the result is a faster and more reliable protocol development scenario than would otherwise be possible, because low-level errors are knocked out quickly and painlessly.

About the Authors
Ada Lovelace *is the psuedonym of MLN. At the time of writing & coding, she had just completed her (northern) 2012 summer on internship with CAcert for their (southern) winter BirdShack project. Thereafter she entered Junior year at UNC-Chapel Hill in the Computer Science stream, and is now in Senior year.*
Iang *managed the WebFunds team through over a decade of hard cryptographically-secure programming. During this time, the team developed the ability to rapid-prototype protocols and know they were secure.*
The WebFunds Team invented and evolved that which this paper christens the Ouroboros pattern, during the period 1995-2002. We were, at various times: Gary Howland, Mike Wynn, Erwin van der Koogh, Jeroen van Gelderen, Edwin Woudt and Iang.

(1) **Introduction**

As an internship project, I (Ada Lovelace) was tasked to develop a protocol for BirdShack - a classical 3-tier architecture of web servers for user access, middleware server for business logic and backends for database and certificate signing [BirdShack]. An early decision was made to use CRUD and Rest for the communications between web servers and middleware server.

In short, CAcert required a protocol framework for the middleware that is reliable, secure and easy to export to different web server languages. This context formed the ideal environment for exploring and documenting the Ouroboros Pattern - a tiny framework developed by the WebFunds team over many years in developing secure payment systems [WebFunds].

The Ouroboros Pattern permits a much faster and more reliable protocol development scenario than would otherwise be possible, because low-level errors are knocked out quickly and painlessly by a form of object-level loopback testing. We believe the technique to be novel, and worth documenting with all its background logic. Hence this paper is a work-product of the Ada's internship to finally place Ouroboros in the public view for critique and use.

Explanatory Notes

The structure of this paper follows that of my (Ada's) gradual understanding of the subject matter. As a novice, this will be a bottom-up construction from data streams to network programming.

This paper is in two parts - the effect of Object Oriented Programming (OOP) on protocol work, and the Ouroboros pattern as an extension of those techniques. People deeply familiar with OOP in the networking context may wish to skip directly to the third section. The approach described herein can be contrasted with many other approaches but detailed comparisons are beyond scope [Wikipedia].

We assume cooperating agents are sending data as packets back and forth. These concepts are equally applicable to client-server, master-slave and peer-2-peer and data persistence. This paper uses Java, but the concepts have been proven in PHP and Perl, and even in C.

(2) The OO protocol revolution

Object oriented programming (OOP) has had a **dramatic** effect on the art of protocol programming. Where previously code was written in very large and complex conglomerations of functions, OOP provided a way to break down the complexity into neat packet-oriented parcels. This section describes why OOP achieves such a dramatic difference, in part because many programmers outside of OO do not know what they are missing, and in part because it sets the stage for the Ouroboros pattern.

Streams

OOP provides structure for the placing of each natural element of the protocol traffic into a class of its own. In each class, Input/Output (IO) methods to read and write data find themselves naturally alongside each other, facilitating both easy maintenance and easy alignment of their mutual contract to read/write the object data.

This ordering along the lines of data element composition however led to a strong need for more powerful IO handles, a gap which Streams filled [Streams]. The complexity of connecting IO with many subsidiary elements was simplified by passing an IO handle called a Stream along to each object that could concentrate on its sole task of handling its own data.

At the lowest level of composition, bytes need to be output, then input again by a matching process. In OO programming, this is organized naturally by Streams which both carry the handle for lower-level IO, and allow increasing sophistication in the nature of the data worked with, by means of extending, compatible versions.

For example, consider Java's *DataOutputStream*. It has methods that will write out the primitive types: bytes, floats, longs, and Strings. Yet, even these methods write out data in a format chosen for Java. As protocols typically cannot limit themselves to one implementation or language choice, we need our own output stream, which can provide a uniform and canonical layout for a small subset of agreed primitives. It does not matter how this differs from Java's formats, what matters is that it belongs to us - we can export this canonical layer across all our implementations.

Our canonical formats layer is created in classes *WireOutputStream* and *WireInputStream*, names that reflects our focus on transmitting objects over the wire. *WireOutputStream* extends on Java's *OutputStream* to provide the agreed set of canonical primitives. In this article we will comment only on two, mostly because these fill 99% of needs:

> ✓ CompactInt is a series of bytes, each holding 7 bits of the unsigned number. In each byte, the high bit signals whether this is the last byte. See Endnotes for an example.

✓ ByteArray is composed of two concatonated elements: a CompactInt, and a series of bytes, the length of which is described by the former CompactInt.

Using Wire Streams

Consider a class to implement a CAcert assured name. It would handle primitives like this in psuedocode:

```
class Name {
 int points;       // how assured this name is
 String name;

 public void wireEncode(WireOutputStream wos)     {
      wos.writeCompactInt( points );
      wos.writeByteArray( name.getBytes() );
 }
 public void wireDecode(WireInputStream wis)   {
      points = wis.readCompactInt();
      name = new String( wis.readByteArray() );
 }
}
```

Each class within any OOP network system will typically include encoding and decoding methods. For example, our classes extend from *WireObject*, which requires methods such as *wireEncode()* and *wireDecode()* above.

Controlling Semantics

In security coding, it is essential to check all inputs. When applied here, each of our classes must then check the ranges of each element written. For example, imagine that our class includes a set of Assurance points limited from 0 to 100:

```
public void wireDecode(WireInputStream wis)
{
  points = wis.readCompactInt();
  if ( (points < 0) || (points > 100) )
            throw new Exception("points out of range");
  …
}
```

As a result of our security mandate, each class *has to control local semantics*, and is thus assumed as a responsibility of the wireDecode.

Once the semantics are absorbed into the class, it turns out that the two primitives for *CompactInt* and *ByteArray* provide for most data needs; within OOP each class can simply apply additional semantics to model up any variation to the primitives needed.

This reduces complexity. In contrast, Java's *DataOutputStream* offers integers of 1, 2, 4, and 8 bytes, OpenPGP has 8 different ways of encoding an integer [OpenPGP], and other formats are no less overwhelmingly helpful. Formats derived from hardware register sizes have little utility in typical network programming because packets are pure application data. Instead, one single expansible form of number accompanied by range-checking replaces all the other forms, which reduces programmer costs and increases reliability.

In the above psuedocode, we can also see a further reason why OOP helps protocol programmers. Input of data is heavily influenced by error conditions, and *exceptions* provide a convenient way to deal with deep errors without causing overwhelming demands on the logical thread.

Composition

OOP also gives us the ability to compose our packets like lego blocks. For example, consider a Member:

```
class Member {
 int unique;       // identifier within system
 Name name;
 String email;

 wireEncode(WireInputStream wis) {
      unique = wis.readCompactInt();
      name = new Name();           // Name knows what to do!
      name.wireDecode(wis);
      email = new String( wis.readByteArray() );
 }
 wireDecode(WireOutputStream wos) {
      wos.writeCompactInt(unique);
      name.wireEncode(wos);        // Name knows what to do!
      wos.writeByteArray( email.getBytes() );
 }
}
```

In the above, the subsidiary object *name* is as easy to handle as the primitives.

Summary of OOP benefits

The above describes what is now well known -- that writing network code the OOP way is much easier. Let's summarize. Firstly, streams give powerful handles that allow us to build a customized layer of primitives, as well as handling the IO. Secondly, classes can share and control the definition of the data much more readily, by dint of the tightness of the OO concept and thus assist our security goals. Thirdly, classes are readily composable from others following the same rules. Fourthly, exceptions make error handling simple.

These benefits make writing network code far more tractable, they translate it from a black art to an engineering science. In this concept, each class's exclusive purpose is to transport its own component of data--all methods are auxiliary to this purpose. By constraining them to this end, in network programming we can construct classes as if they were lego blocks, and build the protocol packets upwards and outwards. Indeed, they allow the programmer to focus on the meaning of the protocol, not the chain of functions and logic through which bytes travel.

Alternative Approaches

It is worth mentioning some alternate approaches to network programming that are popular.

The custom language school, or IDLs for Interface Definition Language. This method is typified by google's ProtoBufs or Apache Thrift, which take a description of the packet data and produce classes and code to handle IO of those packets.

The XML language school. In this method, a textual layout is specified for all primitives, and classes are build up using XML's textual language. This approach requires a generator to produce output, and parsers to read in the input. The latter is particularly fraught, as XML is both complicated and broad.

The pure binary school, ASN.1 with its very complicated binary formats. JSON typifies the tag=value approach in a simple, easy to read textual format. Thrift comes closer to our form with a stream-oriented TLV binary approach.

All of these methods suffer disadvantages: they require programmers to learn another language, and to incorporate parsers or libraries. They hide the real byte-level data from the programmer through layers and semantically powerful tools, and they limit the semantic control to that which the language handles. They also tend to suck in middleware frameworks which might or might not assist. Although these tools deliver handsome compatibility and productivity benefits, they come at the cost of what is in effect outsourcing security to a hopefully benign and careful third party.

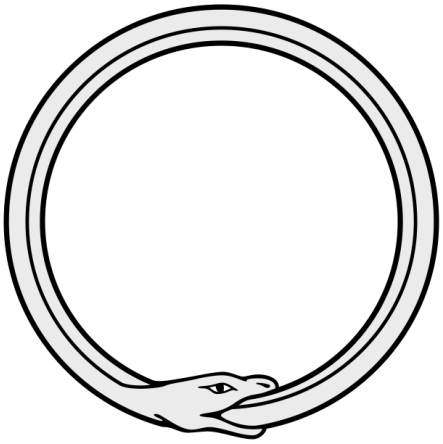
The bottom line is likely that there is no one way to handle serialization of data, and the choice is complicated by many considerations. This approach presented here is just one among many.

(3) The Ouroboros Pattern

Before the advent of these ideas, much protocol code and enough of the application code had to be written before testing could typically begin. Worse, some testing frameworks worked from grammars and protocol specifications, again adding to the workload and "completion trap." Waiting until these components--client, server and protocol--are built before testing for functioning is neither temporally or cost-wise beneficial; simple bugs in lower code mean long debugging cycles which result in lost time better spent on improving the protocol.

To address this we introduce a simple 'unit test' that isolates the errors in each class independently before we begin using it in concert with others in a system. In general terms, this test is a type of loopback test, a concept more typically found in communications hardware and the TCP/IP stack loopback device [Loopback]. It utilises an idea similar to Fuzzing in that the input to the loopback is a randomly formed object, albeit good by construction [Fuzzing].

We call this testing technique the Ouroboros Pattern, after the snake that consumes itself. Our goal is to test the encode/decode path of each class by generating random data to run through the encode/decode methods and then consuming it in a test of equality. To accomplish this, we need two new methods, *example()* and *equals()* in each class.



The ***example()*** method generates a legal object of its class type. It instantiates an object filled with fuzzy or random but valid data; ints are in range, strings follow the proper format, etc.

```
public static Object example()     {
       int points = Ex.exampleInt(0, 100);
       int len = Ex.exampleInt(1, 10);
       String name = Ex.exampleString(len);

       ID me = ID.example();  // ID class also has example()
       return new ThisClass(me, name, points);
}
```

For primitives, we create example values using a source of random numbers and some glue code (above, a class *Ex*). For subsidiary objects, each has its own example(), hence we can obtain a deep example object with no extra work.

Then, ***equals()*** compares objects. Equality is defined in our case by (i) classes being identical, (ii) immutable data being the same, and therefore (iii) the object successfully

transfers to another time/space. In effect we have created an endogenous contract of equality that is driven by the goal of transferring the object - an object is correctly transferred if equals returns true.

```
public boolean equals(java.lang.Object obj)
{
   if ( ! (obj instanceof ThisClass))
       return false;
   ThisClass x = (ThisClass) obj;

   if ( points != x.points)
       return false;
   if ( ! name.equals(x.name))
       return false;
   if ( ! me.equals(x.me))
       return false;
   return true;
}
```

With these two methods in addition to encoding and decoding, we can now describe the full pattern. It is composed of 4 phases: (1) calling the *example()* method to get a random object, (2) calling its *encode()* method to generate a network packet, and then (3) feeding that network packet back into *decode()* to recover the object. Finally, (4) comparing the decoded result to the original object with *equals()*.

```
public static void ouroborosTest()
{
  // 1. Generate the random object:
  ThisClass original = ThisClass.example();

  // 2. Encode the object onto the wire:
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  WireOutputStream wos = new WireOutputStream(baos);
  original.wireEncode(wos);

  // 2.b “send” and “receive” the packet:
  byte[] p = baos.toByteArray();

  // 3. Decode the object from the wire:
  ByteArrayInputStream bais = new ByteArrayInputStream(p);
```

```
    WireInputStream wis = new WireInputStream( bais);
    ThisClass recovered = new ThisClass();
    recovered.wireDecode(wis);

    // 4. Check equality between original and recovered:
    assert (original.equals(recovered));
  }
```

When the wire/en/de coding is working correctly, the object will pass over our virtual wire and fulfill our definition of equality. However, utility is not realized without repetition - a test harness runs each cycle many times so as to hit the extremes in the random data:

```
for (int i = 0; i< 1000; i++) {
     ThisClass.test();
}
// We further abstracted this into a general
// testing harness employing Java's reflection
// to access example() on each class.
```

This pattern is inductive in its behaviour: the contract of equality is ensured through repetition rather than deductive reasoning--a test rather than a proof.

Equality. Let us diverge into the slightly conceptual quandary of what constitutes equality in the domain of objects, noting that languages such as Java leave the definition of *equals()* to the programmer. If an object is a collection of data and methods, then it can be argued that one object is equal to another when they both have the same immutable data and answer to the same calls, as seen by the caller. This definition satisfies our needs within the context of network programming: it tests for the correct transport of data.

Our endogenous contract of equality turns on several points: (i) the obligation of *wireEncode()* to encode its data set, (ii) the matching obligation of *wireDecode()* to recover that data, and (iii) the *equals()* method's ability to check exactly that which is pertinent.

Testing Efficacy. This cycle allows us to check the input and output methods, how the exceptions are being handled, and ensure that the wire class is working well overall. As the major purpose of the class is to transport its data, this test cycle covers all or most of the testing needs of such classes.

Building on the basis of transporting data gives us a strong bottom-up design which is mirrored into our testing strategy. Every time we run ouroboros on a high level class, every component below is retested. This prevents coming back to your early code 18

months later, only to discover bugs that you never squashed; Ouroboros ensures deep-testing while programmers think high level.

Implementations Interoperability Testing. We can expand the pattern to the context of testing compatible implementations or servers. Between a pair of cooperating agents: one end will use the *wireEncode()* to ‘send’ a piece of data across the wire. The receiving end uses *wireDecode()* to collect and then reconstruct the datum at the other end. It immediately encodes it back onto the wire and sends it back to the sender which decodes and checks for equality.

(4) **Conclusion**

The goal of network programming is to reliably transport information. The OOP perspective is to reliably transmit an object. Techniques of class-level encoding and decoding have developed over time to enable an object to in effect help itself be transported.

This strategy of endogenous containment --a class laying out its data to a stream and also recovering that data stream--should be seen as a dramatic contrast to pre-OOP methods. Vertical stacks of functions struggling to make sense of the flow of data, with no real attention to natural composition found within, are replaced with an elegantly self-contained and recursive tree of classes that nicely track and reinforce the data's natural structure. We use the composition, streaming and other benefits of OOP to make the class sovereign in its territory.

Our novel feature is to expand on OOP's bounty to add convenient, appropriate and manageable testing. By adding an example() and equals() method to all classes, and wrapping these methods into a loopback cycle of example -> encode -> decode -> equals, we close the loop on testing: creating, transmitting, recovering and testing-for-equality over thousands or millions of random samples gives great comfort that the basics are working.

As this exactly mirrors the primary goal of the network object, our testing job is done.

Better, the method is self-enforcing through unexpected side-effects: programmers find it easier to build the extra two methods at development time than deal with bugs later on. The test is as efficacious for the individual class author as it is for the project builder. Indeed, because of composition, an Ouroboros test of a high level, large object will automatically test all of its component objects, down to the smallest ones. In this way, Ouroboros scales and aligns with maintenance cycles and team boundaries - we naturally get more and deeper testing the more sophisticated the data layout becomes. It easily leaps across boundaries such as languages and implementations. The supporting code is quite minimal, making porting predictable and mechanical.

All these benefits lead to comfort on the part of the programmer. Within the framework, it is possible to rapidly prototype new protocol conversations, and get them right in the first instance. This frees the coder from the mundane bit-bashing, and leaves him and her thinking of the conversation. What is the right thing to say, when saying anything is easy?

Future work. Ouroboros saves time, but how much? Compared to which other technique? This could be quantified by some form of controlled comparison. Ouroboros also contrasts with techniques like Fuzzing, and an easy extension is to use the latter's malicious injection technique. Is it worthwhile to add a method called *mutate()* that fuzzes with *example()* and inject that many times? A last benefit is that

because it is so tight in its cycle, it is even helpful if one is not incredibly proficient at programming. Can Ouroboros find a place in a pedagogical environment?

REFERENCES

Ref	Citation	Comment
BirdShack	CAcert Community, *BirdShack project* http://svn.cacert.org/CAcert/Software/BirdShack/index_dev.html	classical 3-tier server architecture for a Certification Authority
CRUD	Wikipedia, "Create, read, update and delete," http://en.wikipedia.org/wiki/Create,_read,_update_and_delete	framework for storing and accessing elements of data over the net in client-server architecture
Fuzzing	Takanen, Demott, Miller, *Fuzzing for Software Security Testing and Quality Assurance*, ISBN 13: 978-1-59693-214-2 http://rogunix.com/docs/Reversing&Exploiting/Fuzzing.pdf	black-box testing to inject bad input with little knowledge of the internals (paraphrased, pp26)
Loopback	Wikipedia, "Loopback" http://en.wikipedia.org/wiki/Loopback	routing data from origin directly back to source for comparison testing
OpenPGP	Jon Callas, et al, "OpenPGP Message Format," RFC4880, http://tools.ietf.org/html/rfc4880	describes packets to carry keys and related data for privacy purposes
REST	Roy Fielding, "Representational State Transfer (REST)," *doctoral dissertation - chapter 5*, U.California, Irvine, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm	an abstraction of the architectural elements within a distributed hypermedia system
Streams	David J. Eck, "Streams, Readers, and Writers," Introduction to Programming Using Java, Sixth Edition, V6.0, June 2011, http://math.hws.edu/javanotes/c11/s1.html	abstractions for handling I/O, either a source of input or a destination to which output can be sent.
WebFunds	Systemics Inc, *WebFunds project*, http://www.webfunds.org/	a secure Internet payment system
Wikipedia	Wikipedia, "Comparison of data serialization formats," http://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats	describes, compares and references 20 or so popular frameworks for data layout

ENDNOTES

CompactInt. A compact integer is formed as a sequence of bytes that expand as the value contained grows. In the example below, 143 is encoded:

1	0	0	0	0	0	0	1
0	0	0	0	1	1	1	1

In the above, the leading byte has the high bit set which indicates this is not the last byte. The next byte is high-byte-zero and is this the last byte. The number consists of the right-most 7 bits, shifted / concatonated: (01 << 7) | (0F). This technique can be extended indefinitely, but in practice is employed only up to longs of 64 bits fitting in 9 bytes.

This technique is byte-wise efficient because most numbers, even if they occasionally demand high values, are typically concentrated in the low values. Further, although the processing is a little tricky, the amount of conversion is typically dominated by application code.

Layouts. Other data layouts suffer from the same inappropriate focus as Java - several formats where only one would serve the application programmer. For specific example, only one natural number format is generally required (we do not typically need negatives or floats). This is primarily a historical artifact; other layouts prefer time-honoured abstractions from hardware over newer information-friendly abstractions possible in OOP.

Types. As well as a set of canonical primitives, the WireOutputStream adds methods to write out *typed* versions of these primitives, which are simply the primitive prefixed by a fixed number (as a CompactInt of course). For example, a typed ByteArray is a series of three elements: type, length, bytes. This is similar to the tag-length-value tuple (TLV) found in other protocols.

```
wireEncode(WireOutputStream wos) {
     ...
     wos.writeTypedObject(name);
     ...
}

wireDecode(WireInputStream wis) {
     ...
     name = (Name) wis.readTypedObject( Name.WIRE_TYPE );
```

```
        ...
    }
```

In our Wire layout, the type is optional. More specifically, the use of a type is under the control of each class, which allows it some flexibility in how it lays out its constituent members. For example, in a factory approach, readers can handle a wide range of inputs.

Fuzzing. *Example()* generates an object similar to generation-based fuzzing; however, their approaches are fundamentally different. Fuzzing aims to send bad data into the system, in order to crash it -- and thus harden its response against illegal input [Fuzzing]. Fuzzing as a technique is typically external to the system being tested, even ignorant of it. Ouroboros uses good data to prove it can handle legal cases, and its methods are integral, down to the smallest class level, and is fully aware of the protocol. While both *example()* and generation-based fuzzing use random data to test a system, Ouroboros shows that good data is positively handled, while fuzzing proves that bad data is rejected. Fuzzing intends to crash a system through stress and injected errors, while *example()* aims to show reliability under non-byzantine conditions.

Ouroboros could easily be extended to do fuzzing: a method *fuzz()* or perhaps *mutate()* could take an object (from *example()*) and adjust its data out of legal bounds. This could then be cycled through the three later phases to prove the code's efficacy at rejection. We have not done this, although slavish attention to our security context would suggest we should.

Cost. Ouroboros requires the addition of 2 methods to every network packet class. This is about as much work as the encode and decode; therefore the additional work at the class level is minimal and bounded. The supporting code includes: Wire streaming, a test harness, random primitives and deep equals methods, amounting to around 2000 lines of code.

Frameworks. It has often been the goal to simplify the black arts into objective, defined and hopefully simple frameworks, and network programming rises to that challenge. Typically the approach is to create strong and comprehensive definitions at various levels, as typified by ASN.1 (binary layouts), XML (human readable layouts), and Google's Protocol Buffers (layouts in code), and to build big framework tools to implement these definitions.

However, each of these popular approaches has resulted in large and overwhelming systems, which in the end undermine the original goal. The success of an abstraction is often determined by whether the result is simpler than the original, and few of these frameworks meet the incisive test of Einstein's razor: *everything should be as simple as possible, but no simpler*. E.g., they typically impose the programmer burden of, in

effect, learning another language, and include large packages of non-trivial, specialized code. Instead, direct networking with OOP as described here places the line of abstraction inside the existing OO classes; we get more value from less abstraction.

Goals; Similarities with Java.

Although the Ouroboros Pattern gives a language independent platform, Java shaped the character of our project. This section deals with some of the Java specific effects on our pattern, as well as language based choices we made along the way.

Java's toString() method. Another serendipitous or secondary addition from Java is toString() method. This is also a required and default method, but it comes to shine in debugging of packets.

Java's *serialization* suffers from several shortcomings: language-dependence, version dependence, and no control over the data at the byte level. We choose to use our own encode/decode methods and over others such as Java serialization primarily because it provides greater security.

The Ouroboros cycle was developed during the coding of an extensive, secure payment system for which Java formed the client-side and Perl formed the server-side. This was during the 1990's and early 2000s, in which times Java was less developed. Combined with a sense of critical analysis, this led to a ban on serialization: the obvious Java alternative to the ouroboros cycle. While taking advantage of a language's particular mode of communicating objects across networks seems logical on the surface, there are several inherent drawbacks to serialization for our project context.

Firstly, serialization is language dependent, it destroys our ability to have a server and client written in different languages, and therefore inextricably locates us into Java. It doesn't stop there. Serialization is version dependent, so we could be looking at potential incompatibility nightmares when e.g., someone slips in a new version of Java. (Although not developed in this article, the techniques described herein are also used without change for object database storage.)

For projects that deal with sensitive information, security is a top concern. We want absolute control of what *exactly* was sent across the wire. This omnipotence in information control is better achieved through Ouroboros, in which we are required to stipulate exactly what the immutable data is, and write it out to be sent. The fact that we're building the code from the bottom up, instead of borrowing someone else's (and assuming it is secure) gives us an added layer of security where we need it: at the very bottom.

References based on locations in memory are lost from machine to machine (whether virtual or physical), and the ensuing confusion is not advantageous for network programming, in which our very goal is to transfer unchanging references. The direct OOP techniques do not solve this, rather they force the programmer to deal directly with the issue by not hiding it -- some method of transportable references is required.

Finally, serialization has inherent difficulty dealing with deep copies, and it is better to force the programmer to deal directly with that choice.

Java's hashCode() method. One of our most elusive bugs stemmed from the interrelation of contracts between *hashCode()* and *equals()*. In short, objects that are equal should return hashcodes that are equal, and if this is not followed, certain of the Collections classes will behave erratically. For this reason, (in java) when we override Object's equals() we are required to also override hashcode().

Luckily we can code this relatively easily for the general case, and tuck it into an abstract class:

```
public int hashCode()
{
   ByteArrayOutputStream baos = new ByteArrayOutputStream();
   WireOutputStream wos = new WireOutputStream(baos);
   try {
        this.wireEncode(wos);
   } catch (IOException iox) {
        throw new Error("IOEx in wireEncode???: " + iox);
   }
   byte[] p = baos.toByteArray();
   ByteArrayInputStream bais= new ByteArrayInputStream(p);
   CRC32 crc32obj = new CRC32();
   return (int) crc32obj.getValue();
}
```

RULES, and similar

1. The primary goal of a network object is to transport itself.
2. Each class describes a component.
3. Each object is immutable.
4. Each class has a method *encode()* to stream its object out, and a method *decode()* to stream itself in. (The name is not important, these methods are variously called marshalling, decoding, streaming, outputting in other approaches.)
5. Rose's network security principle applies: we are precise about what we send out and we are precise about what we accept in. (With a nod to Rose [RFC3117], we note that this is in contrast to Postel's robustness principle [RFC1122].)
6. Therefore, *decode()* must check entirely and fully for presence, legality and sanity of input. Exceptions are thrown, object is not returned.

7. Likewise, original constructors should blocks illegal objects from existence, and/or encode should decline to output a bad object.
8. Equality of the object means that the object & data that should be transmitted has been transmitted. Each class has an *equals()* method that implements this definition.
9. Each class has an *example()* that returns an object stuffed with random but legal and sane data.
10. Each class has a *toString()* method that presents the object's debugging summary.