




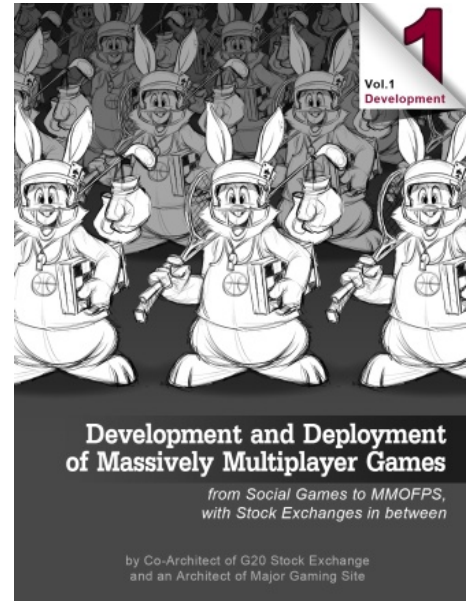
IT Hare on Software

IDL: Encodings, Mappings, and Backward Compatibility

posted February 15, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko 

[[This is Chapter VII(d) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further details see "Book Beta Testing". All the content published during Beta Testing, is subject to change before the book is published.

To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]



As we've discussed those high-level protocols we need, I mentioned Interface Definition Language (IDL) quite a few times. Now it is time to take a closer look at it.

Motivation for having IDL is simple. While manual marshalling is possible, it is a damn error-prone (you need to keep it in sync at least at two different places – to marshal and to unmarshal), not to mention too inconvenient and too limiting for further optimizations. In fact, the benefits of IDL for communication were realized at least 30 years ago, which has lead to development of ASN.1 in 1984 (and in 1993 – to DCE RPC).



These days in game engines, quite often a (kinda) IDL is a part of the language/engine itself; examples include [RPC]/[Command]/[SyncVar] tags in Unity 5, or UFUNCTION(Server)/UFUNCTION(Client) declarations in Unreal Engine 4. However, for most game and game-like communications I still prefer to have my own IDL. The reason for it is two-fold: first, standalone IDL is inherently better suited for cross-language use, and second, none of in-language IDLs I know are flexible enough to provide reasonably efficient compression for games; in particular, per-field Encodings specifications described below are not possible¹

¹ and even if Encodings (along the lines described below) are implemented as a part of your programming language, they would make it way too cumbersome to read and maintain



“ However, for most game and game-like communications I still prefer to have my own IDL.

IDL Development Flow

With a standalone IDL (i.e. IDL which is not a part of your programming language), development flow (almost?) universally goes as follows:

- you write your interface specification in your IDL

- IDL does NOT contain any implementation, just function/structure declarations
- you compile this IDL (using IDL compiler) into stub functions/structures in your programming language (or languages)
- for callee – you implement callee-side stub functions in your programming language
- for caller – you call the caller-side stub functions (again in your programming language). Note that programming language for the caller may differ from the programming language for callee

One important rule to remember when using IDLs is that

Never Ever make manual modifications to the code generated by IDL compiler.

Modifying generated code will prevent you from modifying the IDL itself (ouch), and usually qualifies as a Really Bad Idea. If you feel such a need to modify your generated code, it means one of two things. Either your IDL declarations are not as you want them (then you should modify your IDL and re-compile it), or your IDL compiler doesn't do what you want (then you need to modify your IDL compiler).



“Modifying generated code usually qualifies as a Really Bad Idea

Developing your own IDL compiler

Usually I prefer to develop my own IDL compiler. From my experience, costs of such development (which are of the order of several man-weeks provided that you're not trying to be overly generic) are more than covered with additional flexibility (and ability to change things when you need) it brings to the project.

With your own IDL compiler:

- whenever you feel the need to change marshalling to a more efficient one (without any changes to the caller/callee code) – no problem, you can do it
- whenever you need to introduce an IDL attribute to say that this specific parameter (or struct member) should be compressed in a different manner² (again, without any changes to the code) – no problem, you can add it
- whenever you want to add support for another programming language – no problem, you can do it

- you can easily have ways to specify the technique to extend interfaces (so that extended interfaces stay 100% backwards-compatible with existing calls/callees), and to have you IDL compiler check whether your two versions of the IDL *guarantee* that the extended interface is 100% backwards-compatible. While techniques to keep backward compatibility are known for some of the IDLs out there (in particular, for ASN.1 and for Google Protocol Buffers), the feature of comparing two versions of IDL for compatibility, is missing from all the IDL compilers I know[[**IF YOU KNOW AN IDL COMPILER WHICH HAS AN OPTION TO COMPARE TWO VERSIONS OF IDL FOR BACKWARD COMPATIBILITY – PLEASE LET ME KNOW**]]

Now to the question “how to write your own IDL compiler”. Very briefly, the most obvious and straightforward way is the following:

- write down declarations you need (for example, as a BNF). To start with your IDL, you usually need only two things:
 - declaring structures
 - declaring RPCs
 - in the future, you will probably want more than that (collections being the most obvious example); on the other hand, you’ll easily see it when it comes 😊
- then, you can re-write your BNF into YACC syntax
- then, you should be able to write the code to generate Abstract Syntax Tree (AST) within YACC/Lex (see the discussion on YACC/Lex in Chapter VI).
- As soon as you have your AST, you can easily generate whatever-stubs-you-want.

² see section “Publishable State: Delivery, Updates, Interest Management, and Compression” above for discussion of different compression types

IDL + Encoding + Mapping

Now, let’s take a look at the features which we want our IDL to have. First of all, we want our IDL to specify protocol that goes over the network. Second, we want to have our IDL compiler to generate code in our programming language, so we can use those generated functions and structures in our code, with marshalling for them already generated.

AST
In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language.

— Wikipedia —

When looking at existing IDLs, we'll see that there is usually one single IDL which defines both these things. However, for a complicated distributed system such as an MMO, I suggest to have it separated into three separate files to have a clean separation of concerns, which simplifies things in the long run.

The first file is the IDL itself. This is the only file which is strictly required. Other two files (Encoding and Mapping) should be optional on per-struct-or-function basis, with IDL compiler using reasonable defaults if they're not specified. The idea here is to specify only IDL to start working, but to have an ability to specify better-than-default encodings and mappings when/if they become necessary. We'll see an example of it a bit later.

ASN.1
Abstract Syntax
Notation One
(ASN.1) is a
standard and
notation that
describes rules
and structures
for
representing,
encoding,
transmitting,
and decoding
data in
telecommunications
and computer
networking.

— Wikipedia —

The second file (“Encoding”) is a set of additional declarations for the IDL, which allows to define Encodings (and IDL+Encodings effectively define over-the-wire protocol). In some sense, IDL itself is similar to ASN.1 language as such, and IDL encodings are similar to ASN.1 “Encoding Rules”. IDL defines *what* we're going to communicate, and Encodings define *how* we're going to communicate this data. On the other hand, unlike ASN.1 “Encoding Rules”, our Encodings are more flexible and allow to specify per-field encoding if necessary.

Among other things, having Encoding separate from IDL allows to have different encodings for the same IDL; this may be handy when, for example, the same structure is sent both to the client and between the servers (as optimal encodings may differ for Server-to-Client and Server-to-Server communications; the former is usually all about bandwidth, but for the latter CPU costs may play more significant role, as intra-datacenter bandwidth usually comes for free until you're overloading the Ethernet port, which is not that easy these days).

The third file (“Mapping”) is another set of additional declarations, which define what kind of code we want to generate to use for our programming language. The thing here is that the same IDL data can be “mapped” into different data types; moreover, there is no one single “best mapping”, so it all depends on your needs at the point where you're going to use it (we'll see examples of it below). Changing “Mapping” does NOT change the protocol, so it can be safely changed without affecting anybody else.

In the extreme case, “Mapping” file can be a file in your target programming language.

Example: IDL

While all that theoretical discussion about IDL, Encodings, and Mappings is interesting, let's bring it a bit down to earth.

Let's consider a rather simple IDL example. Note that this is just an example structure in the very example IDL; syntax of your IDL may vary very significantly (and in fact, as argued in "Developing your own IDL compiler" section above, you generally SHOULD develop your own IDL compiler – that is, at least until somebody makes an effort and does a good job in this regard for you):

```
1 PUBLISHABLE_STRUCT Character {
2   UINT16 character_id;
3   NUMERIC[-10000,10000] x;//for our example IDL compiler, notation [a,b] means
4       // "from a to b inclusive"
5       //our Game World has size of 20000x20000m
6   NUMERIC[-10000,10000] y;
7   NUMERIC[-100.,100.] z;//Z coordinate is just +/- 100m
8   NUMERIC[-10.,10.] vx;
9   NUMERIC[-10.,10.] vy;
10  NUMERIC[-10.,10.] vz;
11  NUMERIC[0,360] angle;//where our Character is facing
12      //notation [a,b] means "from a inclusive to b exclusive"
13  enum Animation {Standing=0,Walking=1, Running=2} anim;
14  INT[0,120) animation_frame;//120 is 2 seconds of animation at 60fps
15
16  SEQUENCE<Item> inventory;//Item is another PUBLISHABLE_STRUCT
17      // defined elsewhere
18  };
```

This IDL declares *what* we're going to communicate – a structure with current state of our Character.³

³ yes, I remember that I've advised to separate inventory from frequently-updated data in "Publishable State" section, but for the purposes of this example, let's keep them together

Example: Mapping

Now let's see how we want to map our IDL to our programming language. Let's note that mappings of the same IDL MAY differ for different communication parties (such as Client and Server). For example, mapping for our data above MAY look as follows for the Client:

```

1  MAPPING("CPP","Client") PUBLISHABLE_STRUCT Character {
2      UINT16 character_id;//can be omitted, as default mapping
3          // for UINT16 is UINT16
4      double x;//all 'double' declarations can be omitted too
5      double y;
6      double z;
7      double vx;
8      double vy;
9      double vz;
10     float angle;//this is the only Encoding specification in this fragment
11         // which makes any difference compared to defaults
12         // if we want angle to be double, we can omit it too
13     enum Animation {Standing=0,Walking=1, Running=2} anim;
14         //can be omitted too
15     UINT8 animation_frame;//can be omitted, as
16         // UINT8 is a default mapping for INT[0,120)
17
18     vector<Item> inventory;//can be also omitted,
19         // as default mapping for SEQUENCE<Item>
20         // is vector<Item>
21 };

```

In this case, IDL-generated C++ struct may look as follows:

```

1  struct Character {
2      UINT16 character_id;
3      double x;
4      double y;
5      double z;
6      double vx;
7      double vy;
8      double vz;
9      float angle;
10     enum Animation {Standing=0,Walking=1, Running=2} anim;
11     UINT8 animation_frame;
12     vector<Item> inventory;
13
14     void idl_serialize(int serialization_type,OurOutputStream& os);
15         //implementation is generated separately
16     void idl_deserialize(int serialization_type,OurInputStream& is);
17         //implementation is generated separately
18 };

```

On the other hand, for our Server, we might want to have *inventory* implemented as a special class Inventory, optimized for fast handling of specific server-side use cases. In this case, we MAY want to define our Server Mapping as follows:

```

1  MAPPING("CPP","Server") PUBLISHABLE_STRUCT Character {
2  // here we're omitting all the default mappings
3  float angle;
4  class MyInventory inventory;
5  //class MyInventory will be used as a type for generated
6  // Character.inventory
7  //To enable serialization/deserialization,
8  // MyInventory MUST implement the following member functions:
9  // size_t idl_serialize_collection_get_size(),
10 // const Item& idl_serialize_collection_get_item(size_t idx),
11 // void idl_deserialize_collection_reserve_size(size_t),
12 // void idl_deserialize_collection_add_item(const Item&)
13 };

```

As we see, even when we're using the same programming language for both Client-Side and Server-Side, we MAY need different Mappings for different sides; in case of different programming languages such situations will become more frequent. One classical (though rarely occurring in practice) example is that `SEQUENCE<Item>` can be mapped either to `vector<Item>` or to `list<Item>`, depending on the specifics of your code; as specifics can be different on the different sides of communication – you may need to specify Mapping.

Also, as we can see, there is another case for non-default Mappings, which is related to making IDL-generated code to use custom classes (in our example – `MyInventory`) for generated structs (which generally helps to make our generated struct `Character` more easily usable).

Mapping to Existing Classes

One thing which is commonly missing from existing IDL compilers is an ability to “map” an IDL into existing classes. This can be handled in the following way:

- you do have your IDL and your IDL compiler
- you make your IDL compiler parse your class definition in your target language (this is going to be the most difficult part)
- you do specify a correspondence between IDL fields and class fields
- your IDL generates serialization/deserialization functions for your class
 - generally, such functions won't be class members, but rather will be free-standing serialization functions (within their own class if necessary), taking class as a parameter
 - in languages such as C++, you'll need to specify these serialization/deserialization functions as friends of the class (or to provide equivalent macro)



“I want YOU to
read page 2!

**Continued on Page 2... Further topics
include IDL Encodings (including Delta
Compression, rounding, etc.) and IDL
Backward Compatibility**

Example: Encoding

We’ve already discussed IDL and Mapping (and can now use our generated stubs and specify how we want them to look). Now let’s see what Encoding is about. First, let’s see what will happen if we use “naive” encoding for our C++ struct Character, and will transfer it as a C struct (except for *inventory* field, which we’ll delta-compress to avoid transferring too much of it). In this case, we’ll get about 60bytes/Character/network-tick (with 6 doubles responsible for 48 bytes out of it).

Now let’s consider the following Encoding:

```
1  ENCODING(MYENCODING1) PUBLISHABLE_STRUCT Character {
2  VLQ character_id;
3  DELTA {
4      FIXED_POINT(0.01) x; //for rendering purposes, we need our coordinates
5                          //only with precision of 1cm
6                          //validity range is already defined in IDL
7                          //NB: given the range and precision,
8                          // 'x' has 20'000'000 possible values,
9                          // so it can be encoded with 21 bits
10     FIXED_POINT(0.01) y;
11     FIXED_POINT(0.01) z;
12     FIXED_POINT(0.01) vx;
13     FIXED_POINT(0.01) vy;
14     FIXED_POINT(0.01) vz;
15 }
16 DELTA FIXED_POINT(0.01) angle; //given the range specified in IDL,
17                               // FIXED_POINT(0.01) can be encoded
18                               // with 16 bits
19 DELTA BIT(2) Animation; //can be omitted, as 2-bit is default
20                               // for 3-value enum in MYENCODING1
21 DELTA VLQ animation_frame;
22 DELTA SEQUENCE<Item> inventory;
23 };
```

VLQ
A variable-

Here we’re heavily relying on the properties of MYENCODING1, which is used to marshal our struct. For the

**length quantity
(VLQ) is a
universal code
that uses an
arbitrary
number of
binary octets
(eight-bit bytes)
to represent an
arbitrarily
large integer.**

— *Wikipedia* —

example above, let's assume that MYENCODING1 is a quite simple bit-oriented encoding which supports delta-compression (using 1 bit from bit stream to specify whether the field has been changed), and also supports VLQ-style encoding; also let's assume that it is allowed to use rounding for FIXED_POINT fields.

As soon as we take these assumptions, specification of our example Encoding above should become rather obvious; one thing which needs to be clarified in this regard, is that DELTA {} implies that we're saying that the whole block of data within brackets is likely to change together, so that our encoding will be using only a single bit to indicate that the whole block didn't change.

Now let's compare this encoding (which BTW is not the best possible one) to our original naive encoding. Statistically, even if Character is moving, we're looking at about 20 bytes/Character/network-tick, which is 3x better than naive encoding.

**Even more importantly, this change in encoding can
be done completely separately from all the
application code(!) – merely by changing Encoding
declaration**

It means that we can develop our code without caring about specific encodings, and then, even at closed beta stages, find out an optimal encoding and get that 3x improvement by changing only Encoding declaration.

Such separation between the code and Encodings is in fact very useful; in particular, it allows to use lots of optimizations which are too cumbersome to think of when you're developing application-level code.

To continue our example and as a further optimization, we can add dead reckoning, and it can be as simple as rewriting Encoding above into

```

1  ENCODING(MYENCODING2) PUBLISHABLE_STRUCT Character {
2  VLQ character_id;
3  DELTA {
4      DEAD_RECKONING(0.02) { //0.02 is maximum acceptable coordinate
5                          // deviation due to dead reckoning
6          FIXED_POINT(0.01) x;
7          FIXED_POINT(0.01) vx;
8      }
9      DEAD_RECKONING(0.02) {
10         FIXED_POINT(0.01) y;
11         FIXED_POINT(0.01) vy;
12     }
13     DEAD_RECKONING { //by default, maximum acceptable deviation
14                     // due to dead reckoning
15                     // is the same as for coordinate
16                     // (0.01 in this case)
17         FIXED_POINT(0.01) z;
18         FIXED_POINT(0.01) vz;
19     }
20 } //DELTA
21 DELTA FIXED_POINT(0.01) angle;
22 DELTA BIT(2) Animation;
23 DELTA VLQ animation_frame;
24 DELTA SEQUENCE<Item> inventory;
25 };

```

When manipulating encodings is *this* simple, then experimenting with encodings to find out a reasonably optimal one becomes a breeze. How much can be gained by each of such specialized encodings – still depends on the game, but if you can try-and-test a dozen of different encodings within a few hours – it will usually allow you to learn quite a few things about your traffic (and to optimize things both visually and traffic-wise too).

Backward Compatibility

One very important (and almost-universally-ignored) feature of IDLs is backward compatibility. When our game becomes successful, features are added all the time. And adding a feature often implies a protocol change. With Continuous Deployment it happens many times a day.

And one of the requirements in this process is that the new protocol always remains backward-compatible with the old one. While for text-based⁴ protocols backward compatibility can usually be achieved relatively easily, for binary protocols (and games almost-universally use binary encodings due to the traffic constraints, see “Publishable State: Delivery, Updates,



“How much can be gained by each of such specialized encodings – still depends on the game, but if you can try-and-test a dozen of different encodings within a few hours – it will usually allow you to learn

Interest Management, and Compression” section above for discussion) it is a much more difficult endeavour, and requires certain features from the IDL.

What we need from an IDL compiler is a mode when it tells whether one IDL qualifies as a “backward-compatible version” of another one

quite a few things about your traffic (and to optimize things both visually and traffic-wise too).

Ok, this feature would certainly be nice for code maintenance (and as a part of build process), but are we sure that it is possible to implement such a feature? The answer is “yes, it is possible”, and there are at least two ways how it can be implemented. In any case, let’s observe that two most common changes of the protocols are (a) adding a new field, and (b) extending an existing field.⁵ While other protocol changes (such as removing a field) do happen in practice, it is usually rare enough occurrence, so that we will ignore it for the purposes of our discussion here.

The first way to allow adding fields is to have field names (or other kind of IDs) transferred alongside with the fields themselves. This is the approach taken by Google Protocol Buffers, where everything is always transferred as a key-value pair (with keys depending on field IDs, which can be explicitly written to the Protocol Buffer’s IDL). Therefore, to add a field, you just adding a field with a new field-ID, that’s it. To be able to extend fields (and also to skip those optional-fields-you-dont-know-about), you need to have size for each of the fields, and Google Protocol Buffers have it too (usually implicitly, via field type). This approach works good, but it has its cost: those 8-additional-bits-per-field⁶ (to contain the field ID+type) are not free.

The second way to allow adding fields into encoded data is a bit more complicated, but allows to deal with not-explicitly-separated (and therefore not incurring those 8-bits-per-field cost) data streams, including bitstreams. To add/extend fields to such non-discriminated streams, we may implement the following approach:

- introduce a concept of “fence” into our Encodings. There can be “fences” within structs, and/or within RPC calls
 - one possible implementation for “fences” is assuming an implicit “fence” after each field; while this approach rules out certain encodings, it does guarantee correctness



“introduce a concept of

“fence” into our Encodings

- between “fences”, IDL compiler is allowed to reorder/combine fields as it wishes (though any such combining/reordering MUST be strictly deterministic).
- across “fences”, no such reordering/combining is allowed
- then, adding a field immediately after the “fence” is guaranteed to be backward-compatible as soon as we define it with a default value
 - within a single protocol update, several fields can be added/extended simultaneously only after one “fence”
 - to add another field in a separate protocol update, another “fence” will be necessary
- extending a field can be implemented as adding a (sub-)field, with a special interpretation of this (sub-)field, as described in the example below

⁴ such as XML-based

⁵ that is, until we’re throwing everything away and rewriting the whole thing from scratch

⁶ Google Protocol Buffers use overhead of 8 bits per field; in theory, you may use something different while using key-value encodings, but the end result won’t be that much different

Let’s see how it may work if we want to extend the following Encoding:

```
1  ENCODING(MYENCODINGA) PUBLISHABLE_STRUCT Character {
2    UINT16 character_id;
3    DELTA {
4      FIXED_POINT(0.01) x;
5      FIXED_POINT(0.01) y;
6      FIXED_POINT(0.01) z;
7      FIXED_POINT(0.01) vx;
8      FIXED_POINT(0.01) vy;
9      FIXED_POINT(0.01) vz;
10   }
11 };
12 //MYENCODINGA is a stream-based encoding
13 // and simply serialized all the fields
14 // in the specified order
```

Let’s assume that we want to extend our UINT16 character_id field into UINT32, and to add another field UINT32 some_data. Then, after making appropriate

changes to the IDL, our extended-but-backward-compatible Encoding may look as follows:

```
1  ENCODING(MYENCODINGA) PUBLISHABLE_STRUCT Character {
2    UINT16 character_id;
3    DELTA {
4      FIXED_POINT(0.01) x;
5      FIXED_POINT(0.01) y;
6      FIXED_POINT(0.01) z;
7      FIXED_POINT(0.01) vx;
8      FIXED_POINT(0.01) vy;
9      FIXED_POINT(0.01) vz;
10 }
11 //Up to this point, the stream is exactly the same
12 // as for "old" encoding
13
14 FENCE
15
16 EXTEND character_id TO UINT32;
17 //at this point in the stream, there will be additional 2 bytes placed
18 // with high-bytes of character_id
19 // if after-FENCE portion is not present – character_id
20 // will use only lower-bytes from pre-FENCE portion
21
22 UINT32 some_data DEFAULT=23;
23 // if the marshalled data doesn't have after-FENCE portion,
24 // application code will get 23
25 };
```

As we can see, for the two most common changes of the protocols, making a compatible IDL is simple; moreover, making an IDL compiler to compare these two IDLs to figure out that they're backward-compatible – is trivial. Formally, IDL B qualifies as a backward-compatible version of IDL A, if all of the following stands:

- IDL B starts with full IDL A
- after IDL A, in IDL B there is a FENCE declaration
- after the FENCE declaration, all the declarations are either EXTEND declarations, or new declarations with specified DEFAULT.

On Google Protocol Buffers

Google Protocol Buffers is one IDL which has recently got a lot of popularity. It is binary, it is extensible, and it is reasonably efficient (or at least not unreasonably inefficient). Overall, it is one of the best choices for a usual business app.

However, when it comes to games, I still strongly prefer my own IDL with my own IDL compiler. The main reason for it is that in Google Protocol Buffers there is only one encoding, and the one which is not exactly optimized for games. Delta compression is not supported, there are no acceptable ranges for values, no rounding, no dead reckoning, and no bit-oriented encodings. Which means that if you use Google Protocol Buffers to marshal your in-app data structures directly, then compared to your own optimized IDL, it will cost you in terms of traffic, and cost a lot.

Alternatively, you may implement yourself most of the compression goodies mentioned above, and then to use Google Protocol Buffers to transfer this compressed data, but it will clutter your application-level code with this compression stuff, and still won't be able to match traffic-wise some of the encodings possible with your own IDL (in particular, bit-oriented streams and Huffman coding will be still out of question⁷).

Therefore, while I agree that Google Protocol Buffers are good enough for most of the business apps out there, I insist that for games you usually need something better. MUCH better.



“Therefore, while I agree that Google Protocol Buffers are good enough for most of the business apps out there, I insist that for games you usually need something better. MUCH better.”

⁷ that is, unless you're using Google Protocol Buffers just to transfer pre-formatted bytes, which usually doesn't make much sense

[[To Be Continued...



This concludes beta Chapter VII(d) from the upcoming book “Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)”. Stay tuned for beta Chapter VIII, “Engine-Centric Architecture: Unity 5, Unreal Engine 4, and Photon Server from MMO point of view”]]

Acknowledgement

Cartoons by Sergey Gordeev^{RL} from Gordeev Animation Graphics, Prague.

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: client, game, IDL, marshallng, multi-player, network, protocol, server

Copyright © 2014-2016 ITHare.com