Erlang Users Conference 2010 Tutorial / UBF Basics and Hands ON

Joseph Wayne Norton <norton@geminimobile.com>

Revision History

Revision 0.1	2010/11/15	JWN

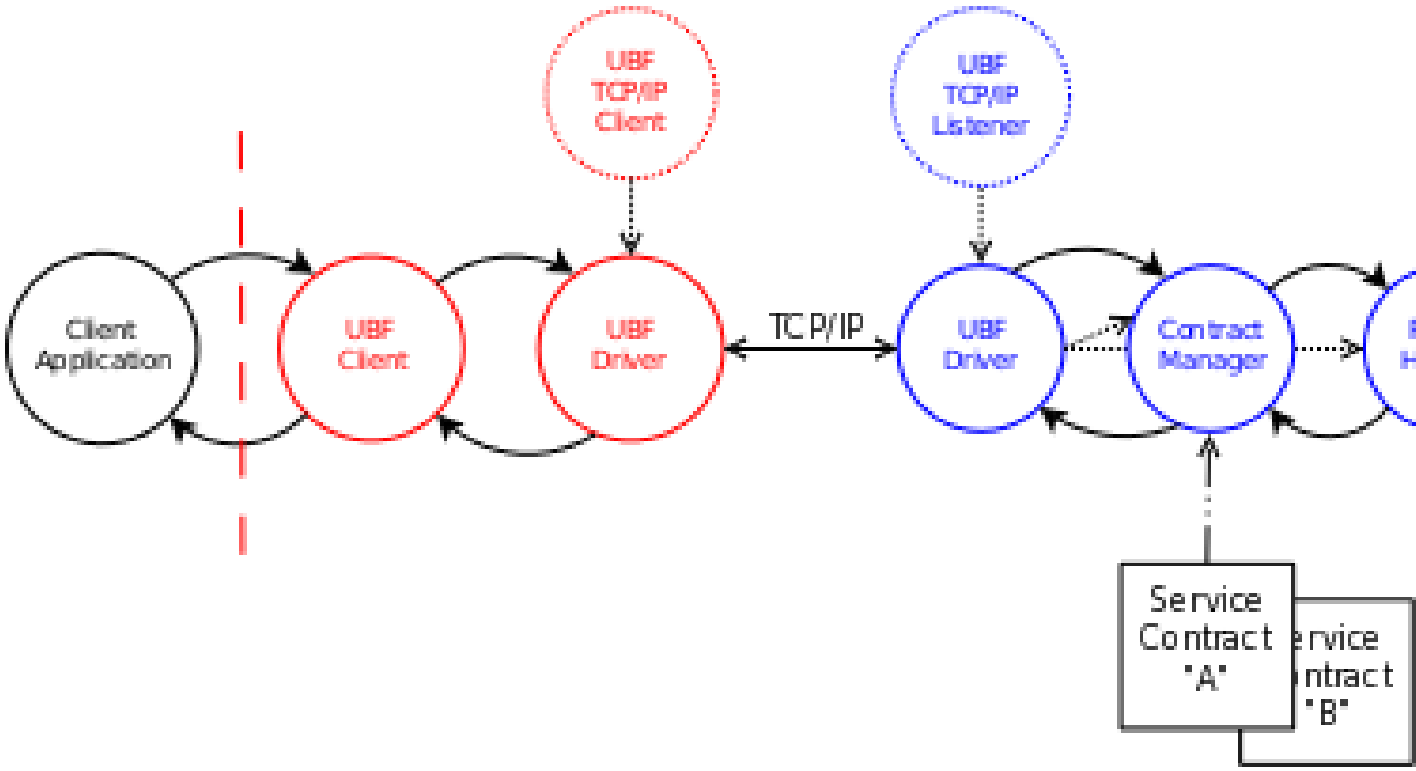
Table of Contents

1. UBF Basics

UBF is a language for transporting and describing complex data structures across a network. It has three components:

- UBF(a) is a "language neutral" data transport format, roughly equivalent to well-formed XML.
- UBF(b) is a programming language for describing types in UBF(a) and protocols between clients and servers. This layer is typically called the "protocol contract". UBF(b) is roughly equivalent to Verified XML, XML-schemas, SOAP and WDSL.
- UBF(c) is a meta-level protocol used between a UBF client and a UBF server.

Figure 1. Programming By Contract



2. Specifications: UBF(a)

Integer	[-][0-9]+
String	"…"
Binary	[0-9]+ ~…~
Atom	'…'
Tuple	{ Obj1 Obj2 … ObjN-1 ObjN }
List	# ObjN & ObjN-1 & … & Obj2 & Obj1
Term	represent primitive types and compound types
White space	\s \n \r \t , %…%

Register	>C C
Object	Term or Register Push or Register Pop

The operator *$* (i.e. "end of object") signifies when objects are finished.

3. Specifications: UBF(a) Example

For example, the following UBF(a) object:

```
'person'>p # {p "Joe" 123} & {p 'fred' 3~abc~} & $
```

Represents the following UBF(b) term, a list that contains two 3-tuples:

```
[{'person', 'fred', <<"abc">>}, {'person', "Joe", 123}].
```

4. Specifications: UBF(b)

UBF(b) is a language independent type system and protocol description language to specify a "contract".

All data sent by both the client and the server is verified by the "Contract Manager" (an Erlang process on the "server" side of the protocol). Any data that violates the contract is rejected.

A contract is defined by 2 mandatory sections and 3 optional sections.

Name	+NAME("…"). *mandatory*
Version	+VSN("…"). *mandatory*
Types	+TYPES.
State	+STATE.

- Defines a finite state machine (FSM) to model the interaction between the client and server.
- Symbolic names expressed as "atoms" are the states of the FSM.
- Transitions expressed as request, response, and next state triplets are the edges of the FSM (a.k.a. synchronous calls).
- States may also be annotated with events (a.k.a. asynchronous casts).

Anystate	+ANYSTATE.

- Defines request and response pairs and define events that are valid in *all* states of the FSM.

5. Specifications: UBF(a) Types

Definition	X() = T

Integer	[-][0-9]+ *or* [0-9]+#[0-9a-f]+
Range	[-][0-9]+..[-][0-9]+ *or* [-][0-9]+.. *or* ..[-][0-9]+
Float	[-][0-9]+.[0-9]+
Binary	<<"…">>
String	"…"
Atom	'…' *or* [a-z][a-zA-Z0-9_]*
Reference	R()
Alternative	T1 | T2
Tuple	{T1, T2, …, Tn}
Record	name#T1, y=T2, …, z=Tn
Extended Record	name##T1, y=T2, …, z=Tn
List	[T]
Predefined	P() *or* P(A1, A2, …, An)

6. Specifications: UBF(a) Predefined Types

	ascii	asciiprintable	nonempty	nonundefined
integer	X	X	X	X
float	X	X	X	X
binary	O	O	O	X
string	O	O	O	X
atom	O	O	O	O
tuple	X	X	O	X
list	X	X	O	X
proplist	X	X	O	X
term	X	X	O	O
void	X	X	X	X

nonempty does not match: <<"">>, "", '', {}, and []. *nonundefined* does not match: 'undefined'.

7. Specifications: UBF(b) Example

```
+NAME("irc").

+VSN("ubf1.0").

+TYPES
```

```
info()            = info;
description()     = description;
contract()        = contract;

ok()              = ok;
bool()            = true | false;
nick()            = string();
oldnick()         = nick();
newnick()         = nick();
group()           = string();
groups()          = [group()];

logon()           = logon;
proceed()         = {ok, nick()};
listGroups()      = groups;
joinGroup()       = {join, group()};
leaveGroup()      = {leave, group()};
changeNick()      = {nick, nick()};
msg()             = {msg, group(), string()};

msgEvent()        = {msg, nick(), group(), string()};
joinEvent()       = {joins, nick(), group()};
leaveEvent()      = {leaves, nick(), group()};
changeNameEvent() = {changesName, oldnick(), newnick(), group()}.

+STATE start
   logon()       => proceed() & active. %% Nick randomly assigned

+STATE active
   listGroups()  => groups() & active;
   joinGroup()   => ok() & active;
   leaveGroup()  => ok() & active;
   changeNick()  => bool() & active;
   msg()         => bool() & active;    %% False if you have not joined a group

   EVENT         => msgEvent();         %% Group sends me a message
   EVENT         => joinEvent();        %% Nick joins group
   EVENT         => leaveEvent();       %% Nick leaves group
   EVENT         => changeNameEvent().  %% Nick changes name

+ANYSTATE
   info()        => string();
   description() => string();
   contract()    => term().
```

8. Specifications: UBF(c)

UBF(c) is a meta-level protocol used between a UBF client and a UBF server.

UBF(c) has two primitives: synchronous "calls" and asynchronous "casts".

Calls Request $ # {Response, NextState} $

- "Request" is an UBF(a) type sent by the client
- "Response" is an UBF(a) type and "NextState" is an UBF(a) atom sent by the server

Casts {'event_in', Event} $

- "Event" is an UBF(a) type sent by the client

Casts {'event_out', Event} $

- "Event" is an UBF(a) type sent by the server

9. Specifications: UBF(c) errors

Calls - Request

If clients sends an invalid request, server responds with "client broke contract":

```
{{'clientBrokeContract', Request, ExpectsIn}, State} $
```

Calls - Responses

If server sends an invalid response, server responds with "server broke contract":

```
{{'serverBrokeContract', Response, ExpectsOut}, State} $
```

Casts

If client or server send an invalid event, the event is ignored and dropped by the server.

10. Contracts and Plugins

"Contracts" and "Plugins" are the basic building blocks of an Erlang UBF server. - Contracts are a server’s specifications. - Plugins are a server’s implementations.

A contract is a UBF(b) specification stored to a file. By convention, a contract’s filename has ".con" as the suffix part.

11. Contracts: +TYPES only

For example, a "+TYPES" only contract having the filename "irc_types_plugin.con" is as follows:

```
+NAME("irc_types").

+VSN("ubf1.0").

+TYPES
info()            = info;
description()     = description;
contract()        = contract;

ok()              = ok;
bool()            = true | false;
nick()            = string();
oldnick()         = nick();
newnick()         = nick();
group()           = string();
groups()          = [group()];
```

```
logon()           = logon;
proceed()         = {ok, nick()};
listGroups()      = groups;
joinGroup()       = {join, group()};
leaveGroup()      = {leave, group()};
changeNick()      = {nick, nick()};
msg()             = {msg, group(), string()};

msgEvent()        = {msg, nick(), group(), string()};
joinEvent()       = {joins, nick(), group()};
leaveEvent()      = {leaves, nick(), group()};
changeNameEvent() = {changesName, oldnick(), newnick(), group()}.
```

12. Contracts: +STATE and/or +ANYSTATE only

For example, a "+STATE" and "+ANYSTATE" contract having the filename "irc_fsm_plugin.con" is as follows:

```
+NAME("irc").

+VSN("ubf1.0").

+STATE start
   logon()       => proceed() & active. %% Nick randomly assigned

+STATE active
   listGroups()  => groups() & active;
   joinGroup()   => ok() & active;
   leaveGroup()  => ok() & active;
   changeNick()  => bool() & active;
   msg()         => bool() & active;    %% False if you have not joined a group

   EVENT         => msgEvent();         %% Group sends me a message
   EVENT         => joinEvent();        %% Nick joins group
   EVENT         => leaveEvent();       %% Nick leaves group
   EVENT         => changeNameEvent().  %% Nick changes name

+ANYSTATE
   info()        => string();
   description() => string();
   contract()    => term().
```

13. Plugins

A plugin is just a "normal" Erlang module that follows a few simple rules. For a "+TYPES" only contract, the plugin contains just the name of it’s contract.

The plugin for the "+TYPES" only contract having the filename "irc_types_plugin.erl" is as follows:

```
-module(irc_types_plugin).

-compile({parse_transform,contract_parser}).
```

```
-add_contract("irc_types_plugin").
```

Otherwise, the plugin contains the name of it’s contract plus the necessary Erlang "glue code" needed to bind the UBF server to the server’s application.

Check the UBF User’s Guide for possible ways that a plugin’s contract may fail to compile.

14. Plugins - Importing Types

A plugin can also import all or a subset of "+TYPES" from other plugins. This simple yet powerful import mechanism permits sharing and re-use of types between plugins and servers.

The plugin for the "+STATE" and "+ANYSTATE" contract having the filename "irc_fsm_plugin.erl" is as follows:

```
-module(irc_fsm_plugin).

-compile({parse_transform,contract_parser}).
-add_types(irc_types_plugin).
-add_contract("irc_fsm_plugin").
```

The "-add_types('there')" directive imports all "+TYPES" from the plugin named 'there' into the containing plugin.

An alternative syntax "-add_types({'elsewhere', ['t1', 't2', …, 'tn']})." for this directive imports a subset of "+TYPEs" from the plugin named 'elsewhere' into the containing plugin.

15. Transports: TCP/IP - UBF/EBF/JSF/TBF

The following TCP/IP based transports are supported:

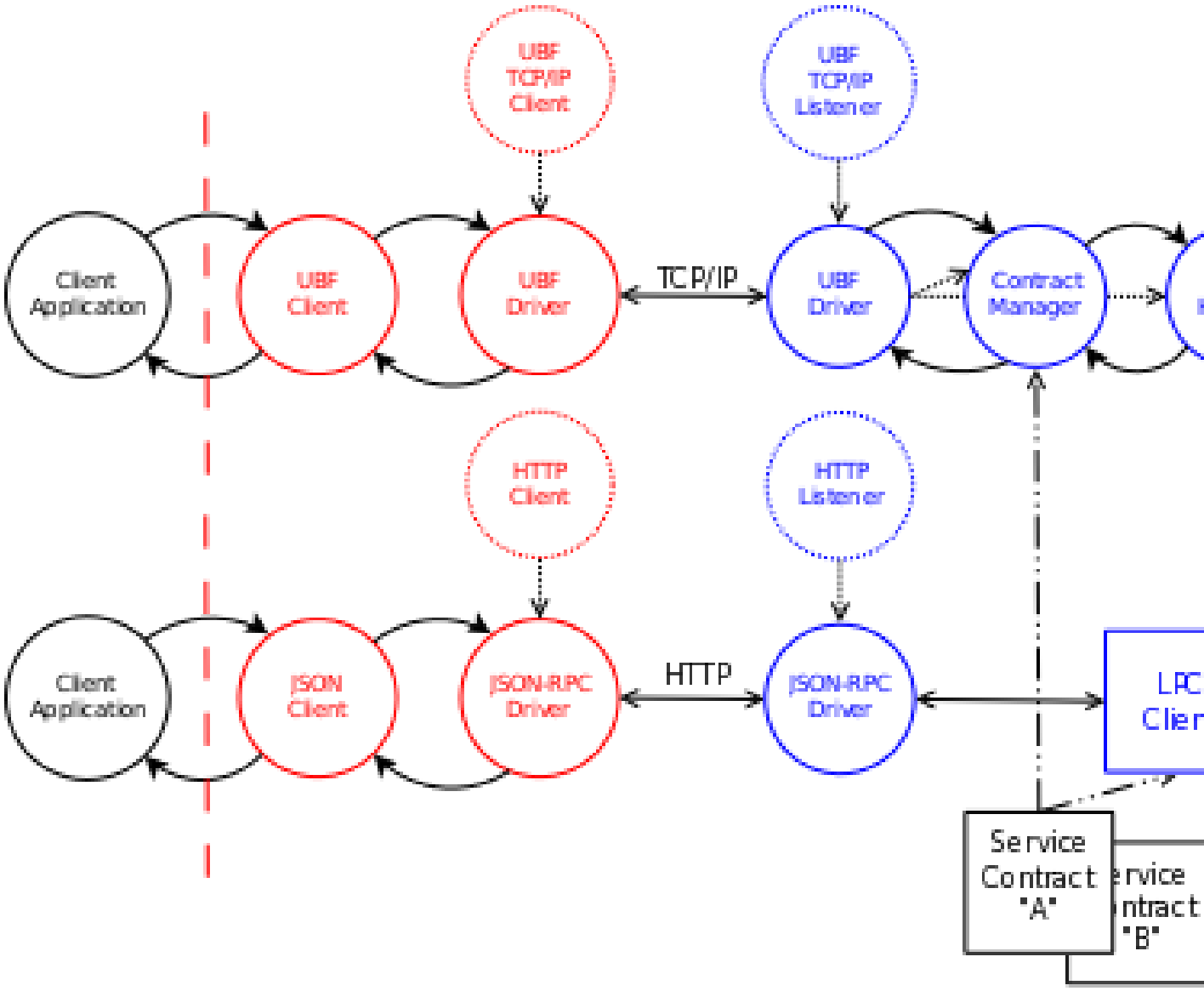
UBF	Universal Binary Format
EBF	Erlang Binary Format
JSF	JavaScript Format
TBF	Thrift Binary Format

16. Transports: HTTP - JSON-RPC

JSON-RPC is a lightweight remote procedure call protocol similar to XML-RPC.

The UBF framework implementation of JSON-RPC brings together JSF’s encoder/decoder, UBF(b)'s contract checking, and an HTTP transport.

Figure 2. Programming By Contract w/ Multiple Transports



Any data that violates the *same* contract(s) is rejected regardless of the transport.

17. Transports: ETF and LPC

Several transports that do not require an explicit network socket have been added to the UBF framework.

ETF	Erlang Term Format *(Erlang's Native Distribution)*
LPC	Local Procedure Call *(Calls made directly to a Plugin)*

These transports permit an application to call a plugin directly without the need for TCP/IP or HTTP.

18. Servers

The UBF framework provides two types of Erlang servers: "stateless" and "stateful". The stateless server is an extension of Joe Armstrong’s original UBF server implementation. The "stateful" server is Joe Armstrong’s original UBF server implementation.

UBF servers are introspective - which means the servers can describe themselves. The following commands (described in UBF(a) format) are always available:

'help' $	Help information
'info' $	Short information about the current service
'description' $	Long information about the current service
'services' $	A list of available services
'contract' $	Return the service contract
{'startSession', "Name", Args} $	To start a new session for the Name service. Args are initial arguments for the Name service and is specific to that service.
{'restartService', "Name", Args} $	To restart the Name service. Args are restart arguments for the Name service and is specific to that service.

19. Servers: start *or* start_link

The "ubf_server" Erlang module implements most of the commonly-used server-side functions and provides several ways to start a server.

```
-module(ubf_server).

-type name() :: atom().
-type plugins() :: [module()].
-type ipport() :: pos_integer().
-type options() :: [{atom(), term()}].

-spec start(plugins(), ipport()) -> true.
-spec start(name(), plugins(), ipport()) -> true.
-spec start(name(), plugins(), ipport(), options()) -> true.

-spec start_link(plugins(), ipport()) -> true.
-spec start_link(name(), plugins(), ipport()) -> true.
-spec start_link(name(), plugins(), ipport(), options()) -> true.
```

Check the UBF User’s Guide for supported configuration options.

20. Servers: Stateless

The plugin callback API for the stateless server.

```
%% common callback API
-spec info() -> string().
-spec description() -> string().
-spec handlerStop(Handler::pid(), Reason::term(), StateData::term()) ->
                  NewStateData::term().

%% stateless callback API
-spec handlerStart(Args::term()) ->
                  {accept, Reply::term(), StateName::atom(), StateDate::term()} |
                  {reject, Reply::term()}.
-spec handlerRpc(Call::term()) -> Reply::term().
```

21. Servers: Stateful

The plugin callback API for the stateful server.

```
%% common callback API
-spec info() -> string().
-spec description() -> string().
-spec handlerStop(Handler::pid(), Reason::term(), StateData::term()) ->
                  NewStateData::term().

%% stateful callback API
-spec handlerStart(Args::term(), Manager::pid()) ->
                  {accept, Reply::term(), StateName::atom(), StateDate::term()} |
                  {reject, Reply::term()}.
-spec handlerRpc(StateName::atom(), Call::term(), StateDate::term(), Manager::pid()) ->
                {Reply::term(), NewStateName::atom(), NewStateData::term()}.

-spec managerStart(Args::term()) ->
                   {ok, ManagerData::term()}.
-spec managerRestart(Args::term(), Manager::pid()) ->
                     ok | {error, Reason::term()}.
-spec managerRpc(Args::term(), ManagerData::term()) ->
                 {ok, NewManagerData::term()} | {error, Reason::term()}.
```

22. Clients: Erlang RPC

The "default" Erlang client is the "rpc" client and it supports TCP/IP and ETF transports.

The "ubf_client" Erlang module implements most of the commonly-used client-side functions and contains the implementation for all types of Erlang clients.

```
-module(ubf_client).

-type host() :: nonempty_string().
-type ipport() :: pos_integer().
-type name() :: atom().
-type server() :: name() | pid().
-type plugin() :: module().
-type plugins() :: [plugin()].
-type options() :: [{atom(), term()}].
-type service() :: {'#S', nonempty_string()} | undefined.
-type statename() :: atom().
-type tlogger() :: module().
```

```
-spec connect(host() | plugins(), ipport() | server()) ->
             {ok, Client::pid(), service()} | {error, term()}.
-spec connect(host() | plugins(), ipport() | server(), timeout()) ->
             {ok, Client::pid(), service()} | {error, term()}.
-spec connect(host() | plugins(), ipport() | server(), options(), timeout()) ->
             {ok, Client::pid(), service()} | {error, term()}.

-spec stop(Client::pid()) -> ok.

-spec rpc(Client::pid(), Call::term()) -> timeout | term() | no_return().
-spec rpc(Client::pid(), Call::term(), timeout()) -> timeout | term() | no_return().
```

Check the UBF User's Guide for the "lpc" client.

23. UBF Hands On

Provide an opportunity for hands-on experience to download, to build, to develop, and to test a **real** UBF contract, **real** UBF client, and **real** UBF server.

The goal of this exercise is to learn more about UBF and to implement and to test your own Bert-RPC server using the UBF framework.

First, let's briefly review the Bert-RPC [./BERTandBERT-RPC1.0Specification.mht] specification.

24. Setup

UBF requires Erlang/OTP R13B01 or newer. UBF has been tested most recently with Erlang/OTP R13B04.

1. Copy the *ubf-bertrpc.tgz* tarball, *ubf-tutorial.tgz* tarball, and *ubf-user-guide.tgz* tarball from the USB stick to your home directory.

2. Make work directory and untar each of the tarballs:

```
$ mkdir -p ~/work/
$ cd ~/work/
$ tar -xvzf ~/ubf-bertrpc.tgz
$ tar -xvzf ~/ubf-tutorial.tgz
$ tar -xvzf ~/ubf-user-guide.tgz
```

3. Build

```
$ cd ~/work/ubf-bertrpc
$ env BOM_FAKE=1 ./bom.sh co src/erl-tools/ubf-bertrpc
$ env BOM_FAKE=1 ./bom.sh make
$ make ERL=/usr/local/hibari/ert/R13B04/bin/erl
```

Please specify the path to your erlang system's erl executable.

Adding DEBUG="+debug_info" will produce DEBUG enabled beam files.

4. Unit Test

```
$ make ERL=/usr/local/hibari/ert/R13B04/bin/erl test
```

25. BERTRPC Types

Simple Data Types

- integer
- float
- atom
- tuple
- bytelist
- list
- binary

Complex Data Types

- nil
- boolean
- dictionary
- time
- regex

26. BERP

BERP is same as EBF.

EBF is an implementation of UBF(b) but it does not use UBF(a) for the client and server communication. Instead, Erlang-style conventions are used instead:

- Structured terms are serialized via the Erlang BIFs term_to_binary() and binary_to_term().
- Terms are framed using the *gen_tcp* {packet, 4} format: a 32-bit unsigned integer (big-endian?) specifies packet length.

```
+-------------------------+-------------------------------+
| Packet length (32 bits) | Packet data (variable length) |
+-------------------------+-------------------------------+
```

The name "EBF" is short for "Erlang Binary Format".

27. BERT-RPC

Synchronous RPC

- {call, Module, Function, Arguments}
- {reply, Result}

"Asynchronous" RPC

- {cast, Module, Function, Arguments}
- {noreply}

Errors

- {error, {Type, Code, Class, Detail, Backtrace}}
- Protocol Error Codes
- Server Error Codes

Info Directives

- {info, Command, Options}
- {info, callback, [{service, Service}, {mfa, Mod, Fun, Args}]}

Caching Features

- Expiration Caching
- Validation Caching

Streaming Features

- Streaming Binary Request
- Streaming Binary Response

28. UBF-BERTRPC: Application (1 of 2)

1. Change directory to the ubf-bertrpc application.

```
$ cd ~/work/ubf-bertrpc/src/erl-tools/ubf-bertrpc__HEAD
```

2. List directory of the ubf-bertrpc application.

```
$ ls -R
```

```
.:
BOM.mk
ChangeLog
ebin
GMBOM
include
LICENSE
priv
README
src

./ebin:

./include:
bertrpc.hrl
bertrpc_impl.hrl

./priv:
sys.config

./src:
Makefile
bert.erl
bert_driver.erl
ubf_bertrpc_plugin.con
ubf_bertrpc_plugin.erl
Unit-EUnit-Files

./src/Unit-EUnit-Files:
bertrpc_plugin.app
bertrpc_plugin_app.erl
bertrpc_plugin_sup.erl
bertrpc_plugin_test.erl
```

29. UBF-BERTRPC: Application (2 of 2)

3. Review key files of the ubf-bertrpc application.
   - src/ubf_bertrpc_plugin.con
   - src/ubf_bertrpc_plugin.erl
4. Review key files of the ubf-bertrpc application’s eunit tests.
   - ./src/Unit-EUnit-Files/bertrpc_plugin.app
   - ./src/Unit-EUnit-Files/bertrpc_plugin_app.erl
   - ./src/Unit-EUnit-Files/bertrpc_plugin_sup.erl
   - ./src/Unit-EUnit-Files/bertrpc_plugin_test.erl
5. Review ubf-bertrpc application’s Makefile.

- src/Makefile

The command make target "run-erl1" starts an erlang shell that can be used for interactive development, debugging, and testing.

30. Basic Exercises (1 of 3)

1. Implement and test BERT-RPC's call/3 and reply/1 primitives:
   a. Modify ubf_bertrpc_plugin.con
   b. Modify ubf_bertrpc_plugin.erl
   c. Add new unit test to bertrpc_plugin_test.erl that uses erlang:now()

31. Basic Exercises (2 of 3)

2. Implement and test BERT-RPC's error/1 primitive for Server Error Codes:
   a. Modify ubf_bertrpc_plugin.con
   b. Modify ubf_bertrpc_plugin.erl
   c. Add new unit test to bertrpc_plugin_test.erl that tests calling an unknown module "foobar".
   d. Add new unit test bertrpc_plugin_test.erl that tests calling an unknown function "erlang:foobar".

32. Basic Exercises (3 of 3)

3. Implement and test BERT-RPC's cast/1 primitive:
   a. Modify ubf_bertrpc_plugin.con
   b. Modify ubf_bertrpc_plugin.erl
   c. Add new unit test to bertrpc_plugin_test.erl that uses error_logger:error_report/1. Manually check if your test triggers a message to stderr.

33. Advanced Exercises (1 of 4)

1. Implement and test BERT-RPC's info callback/2 primitive:
   a. modify ubf_bertrpc_plugin.con

b. modify ubf_bertrpc_plugin.erl

c. add new unit test to bertrpc_plugin_test.erl that tests using erlang:now().

Re-use the ubf_client.erl client inside the ubf_bertrpc_plugin.erl implementation. Re-use the same test server as target service.

To implement the "{info,…}" construct, changes are required for the bert.erl and bert_driver.erl modules. The module must maintain state and should convert the "{info, …}" BERP and it's corresponding "{call,….}" BERP into a 2-tuple that forms a single UBF request. To avoid such headaches, send such a 2-tuple directly from the UBF client to your server as a work-around.

34. Advanced Exercises (2 of 4)

2. Implement and test BERT-RPC's error/1 primitive for Protocol Error Codes:

   a. modify bert_driver.erl

   b. add new unit test to bertrpc_plugin_test.erl

35. Advanced Exercises (3 of 4)

3. Implement Caching Directives

   a. modify ubf_bertrpc_plugin.erl

   b. add new unit test to bertrpc_plugin_test.erl

36. Advanced Exercises (4 of 4)

4. Implement Streaming Binary Request and Response

   a. modify bert_driver.erl

   b. add new unit test to bertrpc_plugin_test.erl

Create a new ubf_bertrpc_client.erl implementation by implementing a wrapper over the standard ubf_client.erl module.

37. Thank You

For UBF, please check UBF's GitHub repository and webpage for updates.

UBF	https://github.com/norton/ubf

For Hibari - an open-source key-value implemented in Erlang, please check one or more of the following links for updates.

Hibari Open Source project	http://sourceforge.net/projects/hibari/
Hibari Twitter	@hibaridb Hashtag: #hibaridb
Gemini Twitter	@geminimobile
Big Data blog	http://hibari-gemini.blogspot.com/
Slideshare	http://www.slideshare.net/geminimobile