Safe Interprocess Communication with bin-prot

Bin-prot vs marshal

Properties

* No segmentation faults when the data doesn't match the expected type!
* Comparable performance. (slower unmarshal, faster marshal)
* Almost as easy to use

Implementation

* A camlp4 macro generates custom marshal/unmarshal functions for each type
* Efficient binary protocol

Example

Imagine a hierarchical database where each node is a set of pairs.

```
type attribute = {
  key: string;
  values: string list
} with bin_io

type node = {
  name: (string * string) list;
  attributes: attribute list;
} with bin_io

type response =
| Error of string
| Results of node list
with bin_io
```

We might query the database like this,

```
type value_match = {
  attribute: string;
  data: string;
} with bin_io

type query_expr =
| Equal of value_match
| Match of value_match
| And of query list
| Or of query list
| Not of query
with bin_io

type query = {
  start_from: (string * string) list;
  scope: [`Subtree | `Base | `Onelevel];
  query: query_expr;
} with bin_io
```

Example Query

So if we had a database containing people working at Z inc, then a query to find a specific person might be,

```
{ start_from = [ “organization”, “Z inc” ];
  scope = `Subtree;
  query =
    Equal {attribute = “family-name”;
           data = “Furuse”} }
```

And you might find Furuse-san

```
{ name = [“organization”, “Z inc”;
          “ou”, “engineering”;
          “uid”, “jfuruse”];
  attributes =
   [ {key=”organization”;values=[“Z inc”]};
     {key=”ou”;values=[“Z inc”]};
     {key=”uid”;values=[“jfuruse”]};
     {key=”first-name”;values=[”Jun”]};
     {key=”family-name”;values=[“Furuse”]};
     {key=”title”;values=[“Engineer”]} ] }
```

The “with bin_io” directive is read by the camlp4 macro, and used to generate a bunch of functions, here is an overview. You get a few more functions than this, but this is the essential set.

```
(* Writing *)
val bin_size_query : query → int
val bin_write_query : buf → pos:pos → query → pos

(* Reading *)
val bin_read_query : buf → pos_ref → query

(* All the functions in a handy record *)
val bin_t : query Type_class.t
```

Nice utility functions

```
val bin_read_stream :
  ?max_size : int ->
  read : (buf -> pos : int -> len : int -> unit) ->
  'a reader -> 'a

val bin_dump :
  ?header : bool → 'a writer → 'a → buf
```

So given the right kind of IO function, bin_read_stream function will take care of all the details needed in order to read a whole value, and bin_dump will format values so they can be read by bin_read_stream.

So now we can write a function to send a query to the database,

```
let query db q : con → query → response =
  let buf =
    bin_dump ~header:true bin_query.writer q
  in
  send db buf;
  bin_read_stream ~read:(read db) bin_response.reader
;;
```

Pretty easy right!

The server side is nearly the same,

```
let process_client client f :
  con → (query → response) → unit =
  let q =
    bin_read_stream ~read:(read client)
      bin_query.reader
  in
  let response = f q in
  let buf =
    bin_dump ~header:true bin_response.writer
  In
  send client
;;
```

So after this small amount of code (plus a little IO code) you're back to working with ML types.

What About Protocol Changes?

Say we're using this application, and we want to add some extensions to the protocol. For a specific example, say that we want to add a field to query that tells the server whether to use a depth first vs a bredth first traversal when querying. Can we just change the type of query?

```
type query = {
  start_from: (string * string) list;
  scope: [`Subtree | `Base | `Onelevel];
  query: query_expr;
  search_options: [`Dfs | `Bfs] option;
}
```

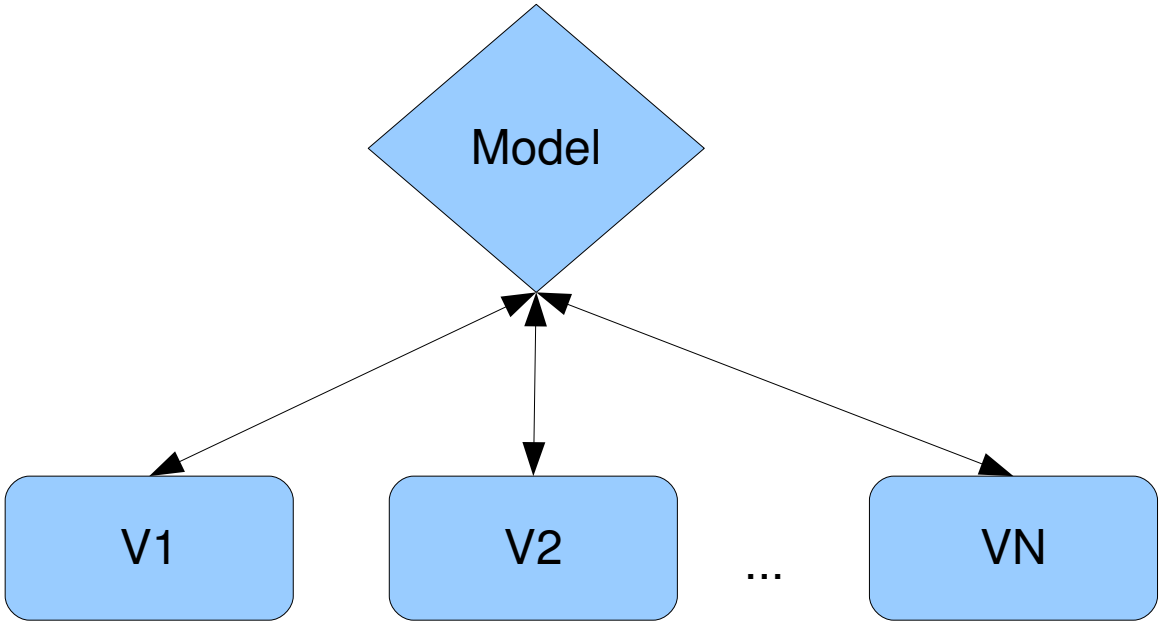
If we control all the clients then we can just upgrade them all at once. However if we don't, then the old client will not work as soon as we upgrade the server. This is because bin_prot will expect this additional record field 'search_options' to be present, and it won't be.

So how do we effectively deal with protocol changes. Here are two ways,

- Design a protocol that isn't tightly coupled
- Version your types

The non tightly coupled basically means using simple ML types like lists and strings to define extensible structures. If you go this way you'll end up manually writing functions to interpret these simple structures, and probably turn them back into ML types, but it will still be easier than doing all the marshaling yourself.

Type Versioning



Model.ml

```
type attribute = V1.attribute
type node = V1.node
type response = V1.response
Type value_match = V1.value_match
Type query_expr = V1.query_expr
type query = V2.query

let query_of_v1 v =
  { V2.start_from = v.V1.start_from;
    scope = v.V1.scope;
    query = v.V1.query_expr;
    search_options = None }
;;
```

```
let v1_of_query q =
  { V1.start_from = q.start_from;
    scope = q.scope;
    query = q.query
    (* We have to ignore search_options
       because it isn't supported in protocol
       Version 1 *) }
```

I'll leave out the necessary changes to the server, but they are pretty simple. So with this mechanism you can build a server that can speak both versions of the protocol, that way you don't need to upgrade all your clients at once.