Extended Process Registry for Erlang

Ulf T. Wiger

Ericsson AB

ulf.wiger@ericsson.com

Abstract

The built-in process registry has proven to be an extremely useful feature of the Erlang language. It makes it easy to provide named services, which can be reached without knowing the process identifier of the serving process.

However, the current registry also has limitations: names can only be atoms (unstructured), processes can register under at most one name, and it offers no means of efficient search and iteration.

In Ericsson's IMS Gateway products, a recurring task was to maintain mapping tables in order to locate call handling processes based on different properties. A common pattern, a form of index table, was identified, and resulted in the development of an extended process registry.

It was not immediately obvious that this would be worthwhile, or even efficient enough to be useful. But as implementation progressed, designers found more and more uses for the extended process registry, which resulted in significant reduction of code volume and a more homogeneous implementation. It also provided a powerful means of debugging systems with many tens of thousand processes.

This paper describes the extended process registry, critiques it, and proposes a new implementation that offers more symmetry, better performance and support for a global namespace.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – abstract data types, patterns, control structures.

General Terms: Algorithms, Languages.

Keywords.: Erlang; process registry

1. Introduction

IMS Gateways is a design unit at Ericsson developing a family of products which are spin-offs of the venerable AXD 301 multiservice ATM switch [1]. That is to say, many of the developers took part in developing the AXD 301, and many concepts have been reused – even though the hardware architecture is entirely new, and much of the software re-written.

The AXD 301 was largely uncharted territory, as no product with such complexity had ever been attempted in Erlang. One of the early challenges was that processes were a scarce resource. Over time, previous limits have been lifted, hardware has become more powerful, and the market more geared towards rapid development of new features. Having previously developed once complex product with a release cycle of 1-1.5 years, we now develop 5-10 similar products in parallel, and with much shorter release cycles. It has become obvious that our programming style must also evolve to fit these new circumstances.

We find that we increasingly move towards programming in "textbook Erlang" style, using a large number of processes, and refraining from low-level optimizations as far as possible. We feel that this style of programming will pay off especially as we now start introducing multi-core processors in our products.

There are drawbacks, however. We have seen that in some of our applications, modeling for the natural concurrency patterns may lead to as many as 200 000-400 000 concurrent processes. While the Erlang virtual machine can handle this many processes without performance degradation, several questions arise:

- How does one debug a system with nearly half a million processes on each processor?
- How does one efficiently operate on data that is spread out across several thousand processes rather than residing in an ETS table?
- How is debugging affected by hiding much information inside process state variables, rather than keeping it in the in-memory database (ETS or Mnesia)?
- How much memory overhead do we get, when we increase the number of process heaps to this extent?

In many cases, trying to address these concerns, designers tend to reduce the number of processes, storing data in ets tables for efficient retrieval. Unfortunately, this often leads to convoluted concurrency models – a problem which grows over time, complicates debugging and hampers product evolution [2]. But if we need to choose between elegant code that might blow up (or at least cause significant disturbance) in live sites during debugging, or complicated code with predictable characteristics, the latter always wins in non-stop systems. The ideal would of course be elegant patterns that also scale to very large systems.

When discussing ways to simplify the code, it became apparent that many of the tasks in our programming could be summarized as "finding the right process(es)", and out of this grew the idea of creating a generic process index. This paper describes this index, called the extended process registry. As is common in commercial product development, time constraints forced us to launch the solution before the concept had been fully understood. This led to some code bloat and inconsistencies, but overall a significant improvement over the model it replaced. We critique the existing implementation and propose a cleaner model, which also works as a global registry.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Erlang'07 October 5, 2007, Freiburg, Germany.

Copyright © 2007 ACM 978-1-59593-675-2/07/0010…$5.00.

2. Current situation

Erlang processes provide more than just execution context and memory protection. They also have a globally unique identity (the PID), and error handling characteristics (process links, trapping exit signals, cascading exits) that make them a powerful component when modeling and structuring a system. In many respects, it is perhaps useful to think of Erlang processes as “agents”. Yet, in many Erlang programs, the processes are surprisingly anonymous. When debugging, a process is primarily identified through its registered name (if it has one), the initial function, and the current function. This is often insufficient. The function sys:get_status/1 may offer more insight, but only if the process is responsive, and supports OTP system messages. As a last resort, one may generate a stack dump of the process and try to decode the contents.

We describe the operations performed on processes as process inspection and process selection.

2.1 Process inspection

As described by Cronqvist [3], powerful debugging tools can be built on top of the process metadata and tracing facilities. Cronqvist describes how debugging large systems is often done by scanning all processes for problem indicators, and then focusing on specific processes for more detailed information. It is noted that grouping related processes is convenient, and that it is sometimes difficult to single out an erroneous process if the number of processes is “large” (600-800). Erlang offers convenient process properties, such as heap_size, reductions, etc. for giving an overview of resource usage.

The programmer cannot add information elements to the standard set of process_info() elements, except by storing data in the process dictionary. OTP offers a library function, sys:get_status(P), which fetches the internal state from a process that conforms to the OTP system message protocol [4]. But iterating over all processes and calling sys:get_status/1 for each could have dire consequences. Not only does it assume that each process supports the OTP system messages (and there is no way to find out except to try it); it also assumes that all processes are ready to answer a query. If a process is busy or blocked waiting for some other message, our query will block as well, and perhaps time out. In a large system, this approach is extremely expensive at best.

2.2 Process selection

It is very common to have programming patterns where a process must find the intended recipient(s) of a message.

2.2.1 Finding a process by unique name

In the simplest case, name registration is used to publish the identity of a known service. Erlang/OTP has two naming services: the built-in registry, and the global name server (called ‘global’). The global name server allows registration of structured names, whereas the built-in registry does not. The built-in registry is blindingly fast, which cannot be said for the global registry.

In the case where a local resource is needed, but where one does not want to register a unique name, things get more convoluted. The following intricate code serves to locate the shell evaluator for the current process. The first snippet was taken from the module group.erl in the OTP kernel application, but nearly identical code is found in other places (user_drv, user). We note with a modicum of glee that the key trick is to peek inside the process dictionary of another process:

Code example 1: Locating IO interface processes (group.erl)

```
%% Return the pid of user_drv and the shell process.
%% Note: We can't ask the group process for this info since it
%% may be busy waiting for data from the driver.
interfaces(Group) ->
  case process_info(Group, dictionary) of
        {dictionary,Dict} ->
          get_pids(Dict, [], false);
        _ ->
          []
  end.

get_pids([Drv = {user_drv,_} | Rest], Found, _) ->
  get_pids(Rest, [Drv | Found], true);
get_pids([Sh = {shell,_} | Rest], Found, Active) ->
  get_pids(Rest, [Sh | Found], Active);
get_pids([_ | Rest], Found, Active) ->
  get_pids(Rest, Found, Active);
get_pids([], Found, true) ->
  Found;
get_pids([], _Found, false) ->
  [].
```

These functions are used from the module shell.erl in the OTP stdlib application. The formatting unfortunately breaks down due to the deep nesting of case statements:

Code example 2: Finding the current shell evaluator (shell.erl)

```
%% Find the pid of the current evaluator process.
whereis_evaluator() ->
 %% locate top group leader,
 %% always registered as user
 %% can be implemented by group (normally)
 %% or user  (if oldshell or noshell)
   case whereis(user) of
     undefined ->  undefined;
     User ->
       %% get user_drv pid from group,
       %% or shell pid from user
       case group:interfaces(User) of
         [] ->  % old- or noshell
           case user:interfaces(User) of
             [] -> undefined;
             [{shell,Shell}] ->
               whereis_evaluator(Shell)
           end;
         [{user_drv,UserDrv}] ->
           %% get current group pid
           %% from user_drv
           case user_drv:interfaces(
                   UserDrv) of
             [] -> undefined;
             [{current_group,Group}] ->
               %% get shell pid from group
               GrIfs =
                 group:interfaces(Group),
               case lists:keysearch(
                      shell, 1, GrIfs) of
                 {value,{shell,Shell}} ->

                   whereis_evaluator(Shell);
                 false ->
                   undefined
               end
           end
       end
    end.
```

We do not claim that the code is badly written; it was written by one of the creators of the Erlang language. The reader is encouraged to think of better alternative solutions.

We can guess at why registered names are not used instead. Most likely, one would like to use structured names. Alternatively, one could register a non-unique name, which could be queried and matched against the group_leader() result for the current process. But Erlang/OTP lacks support for this.

2.2.2 Finding all processes sharing a common property

There is no common pattern for grouping processes in Erlang/OTP, yet it is quite common to do so, using various tricks.

In the OTP Release Handler, the following code is executed during soft upgrade. The purpose is to find all processes which have indicated in their child start specification that they need to be suspended when reloading a certain module:

Code example 3: Finding processes to suspend (release_handler_1.erl)

```
suspend(Mod, Procs, Timeout) ->
  lists:zf(
    fun({_Sup, _Name, Pid, Mods}) ->
         case lists:member(Mod, Mods) of
           true ->
             case catch sys_suspend(Pid, Timeout) of
               ok -> {true, Pid};
               _ ->
                 ..., false
             end;
           false -> false
         end
    end, Procs).
```

The Procs variable was generated using the following function:

```
get_supervised_procs() ->
  lists:foldl(
    fun(Application, Procs) ->
      case application_controller:get_master(Application) of
        Pid when pid(Pid) ->
          {Root, _AppMod} =
            application_master:get_child(Pid),
            case get_supervisor_module(Root) of
            {ok, SupMod} ->
              get_procs(supervisor:which_children(Root),
                              Root) ++
                              [{undefined, undefined,
                                 Root, [SupMod]} | Procs];
            {error, _} ->
              error_logger:error_msg(...),
              get_procs(
                supervisor:which_children(Root), Root) ++ Procs
          end;
        _ -> Procs
      end
    end, [],
    lists:map(
      fun({Application, _Name, _Vsn}) ->
        Application
      end,
      application:which_applications())).
```

The information needed by the release_handler is actually statically defined (most of the time), and exists in the local state of the supervisors. An obvious limitation is that the release handler can only find processes that are members of an OTP supervision tree. The suspend/code_change/resume operation should work just as well for processes that are not, but they have no way of making themselves known.

Again, the code is by no means badly written, but we feel that it is significantly more convoluted than it should be. We also note that the two examples given here use distinctly different strategies for representing process properties. The result is that vital information about processes is scattered all over the place, and the person debugging a system needs to master a wide range of techniques for finding it.

2.2.3 The process dictionary

The process dictionary is a special set of properties. It is usually seen as a built-in hash dictionary for the process, but it plays a special role in crash reports generated by SASL. All OTP behaviours have data about their ancestors in the process dictionary. This is mainly of use when a process dies and a crash report is printed.

Code example 4: Crash report info gathering (proc_lib.erl)

```
crash_report(normal,_)      -> ok;
crash_report(shutdown,_)    -> ok;
crash_report(Reason,StartF) ->
    OwnReport = my_info(Reason,StartF),
    LinkReport = linked_info(self()),
    Rep = [OwnReport,LinkReport],
    error_logger:error_report(crash_report, Rep),
    Rep.

my_info(Reason,StartF) ->
    [{pid, self()},
     get_process_info(self(), registered_name),
     {error_info, Reason},
     {initial_call, StartF},
     get_ancestors(self()),
     get_process_info(self(), messages),
     get_process_info(self(), links),
     get_cleaned_dictionary(self()),
     get_process_info(self(), trap_exit),
     get_process_info(self(), status),
     get_process_info(self(), heap_size),
     get_process_info(self(), stack_size),
     get_process_info(self(), reductions)
    ].

get_ancestors(Pid) ->
    case get_dictionary(Pid,'$ancestors') of
        {'$ancestors',Ancestors} ->
            {ancestors,Ancestors};
        _ ->
            {ancestors,[]}
    end.

get_cleaned_dictionary(Pid) ->
    case get_process_info(Pid,dictionary) of
        {dictionary,Dict} -> {dictionary,clean_dict(Dict)};
        _                 -> {dictionary,[]}
    end.
```

```
clean_dict([E|Dict]) when element(1,E) == '$ancestors' ->
    clean_dict(Dict);
clean_dict([E|Dict]) when element(1,E) == '$initial_call' ->
    clean_dict(Dict);
clean_dict([E|Dict]) ->
    [E|clean_dict(Dict)];
clean_dict([]) ->
    [].

get_dictionary(Pid,Tag) ->
    case get_process_info(Pid,dictionary) of
        {dictionary,Dict} ->
            case lists:keysearch(Tag,1,Dict) of
                {value,Value} -> Value;
                _             -> undefined
            end;
        _ ->
            undefined
    end.
```

The process dictionary offers a way to use “global variables” within a process. As we have seen previously, it is sometimes also used to extract information from other processes, but doing so is inefficient, since a remote process can only extract the entire dictionary as a list of {Key,Value} tuples.

3. Detailed requirements

We identified the following specific requirements to amend the current situation with our own extended process registry.

3.1 Structured registered names

In our applications, the most intuitive value to use as a registered name for a call handling process is the Call ID. Depending on type of call, the structure of this ID will vary, but it is always unique. Since our systems are designed to handle millions of calls per hour, we cannot possibly generate a unique atom for each, as this would quickly exceed the capacity of the atom table. Because of this, we have traditionally refrained from registering call handling processes, and instead implemented a “dispatch” table, mapping process identifiers to Call ID – a specialized process registry!

3.2 Multiple registered names per process

Through the generations of our products, we have tried different process models for call handling. Each model has its pros and cons. For H.248-based call handling, it makes sense to have one process representing the “context”, and perhaps one process per “termination”[1]. But in some respects, terminations may just as well be handled as properties of the context process. Most of the time, we do not need to address a termination individually, but in some cases, we do. If “only one name per process” is an invariant, we must either decide to register the terminations, and lock ourselves into having one process per termination, or leave them unregistered, and address the context instead (which may, depending on process model, pass on the message to a termination process).

While all call handling is H.248-centric, this is only a minor problem, as terminations cannot exist without a context. However, we also deal with other protocols, e.g. SIP, H.323, and some proprietary, and we have been around long enough to know that this set of protocols changes over time.

The entity needing to send a signal to the process handling the termination, really does not care whether that process is also handling some other resource. It does make sense, however, to require that only one process is the point of contact for any given termination. In terms of the process registry, limiting the number of registered names per process imposes an artificial limitation, but being able to uniquely identifying one process through the use of an alias is obviously useful.

3.3 Registration of non-unique properties

A very common pattern is that a process depends on some other resource, e.g. a signaling link. If the signaling link dies, all calls established through that link must normally be removed. This could easily be accomplished by publishing a “property” identifying the signaling link being used for a given call. Normally this is handled through an ETS table, mapping link ID with Call ID.

By allowing registration of structured names, we could possibly hope to incorporate this information in the registered name, but this can only work for one or a few properties, if at all.

3.4 Efficient selection and inspection of related processes

When performing operations on batches of calls/processes, it is vital that the operations scale to tens or hundreds of thousand processes. Furthermore, while debugging a live system, the support engineer or designer needs to be able to browse process listings without risking damage or overload to the system. Due to the complexity of the software, it is important that dependencies between processes are as visible and regular as possible.

4. Possible solutions

Some of these problems can be addressed with partial solutions. We will discuss them here.

4.1 Registering structured names

The most straightforward way of solving registration of structured names would be to modify the registration BIFs to accept names that are not atoms. Most likely, this would have noticeable performance impact on all registrations, and is therefore frowned upon.

4.2 Atom table garbage collection

Many have called for garbage collection, but it is a problem not easily solved. One possible approach was suggested by Thomas Lindgren in [5]. This would allow more liberal use of dynamically created atoms, but does not really address the problem of registering structured names.

4.3 Process ETS table

One suggestion put forth in the discussion was to represent process metadata in a manner similar to an ETS table, allowing for select()-style searches. This sounds like a good idea, but its implementation seems decidedly non-trivial. It is vital that this projection does not add significant overhead to the normal process handling.

[1] Contexts and terminations are H.248 terms. A context is typically a phone call, and the terminations are the specific data paths on which to send data. A context can include several terminations.

4.4 Exported process dictionary items

A few things could be made to make it more convenient to use the process dictionary for “publishing” metadata. Allowing processes to inspect individual dictionary objects in another process’ dictionary would be a partial solution, and would not be too difficult to implement. Adding some way to “publish” certain keys, thus adding them to a global index might be very useful, but would pose a greater implementation challenge. However, it is often argued that making the process dictionary more convenient risks leading beginners astray, and might serve to promote a style of get/put programming generally considered harmful in Erlang.

4.5 Separate registry

This was our preferred approach, partly since it is relatively easy to implement, and does not require changes to the runtime system. We initially set out to implement a simple registry for unique structured names and non-unique properties. But as is often the case with first implementations, experiments with this simple registry brought new ideas to the table, and new features (counters, registration by proxy, aggregated counters) were added.

The remainder of this paper describes the first implementation of the registry, and then proposes a new solution, which we believe is more regular. It is faster than the first version, while offering more functionality (both global and local scope).

5. The extended process registry (proc)

We created a module called ‘proc’ to implement the extended process registry (sysProc in our products, since we insist on using prefixes to manage the namespace). An Open Source version is available at Jungerl. It is implemented using a gen_server and a set of ordered_set ETS tables. Because it is a regular Erlang implementation, and not integrated into the virtual machine, we have introduced some extra functions for optimization purposes.

The most basic functions are:

- reg(Name) – registers a unique name. Name can be any term.
- unreg(Name) – unregisters a name
- where(Name) -> pid() | undefined – looks up a name
- send(Name, Msg) – sends Msg to a registered process
- add_property(Property) – publishes a non-unique property.
- del_property(Property) – un-publishes a property

In addition to these basic functions, a number of retrieval functions exist:

- fold_names(Fun, Acc, Patterns) -> NewAcc
- fold_properties(Fun, Acc, Patterns) -> NewAcc
- select_names(Patterns) -> [{Name, Pid}]
- select_properties(Patterns) -> [{Property, Pid}
- first_name()/next_name(Prev)/last_name()
- first_property()/...
- info(Pid, Item), where Item = properties | names | ...

In each of the functions above, Patterns is a match pattern similar to the ones used in ets:select_count(Tab, Pattern), where Tab would be an ordered_set table with {Name, Pid} objects.

6. Examples of usage

6.1 Finding processes that share a common property

Going back to the release_handler example in chapter 2.2.2, had the release handler instead used the extended process registry, the corresponding implementation could have looked like this:

Code example 5: Idea for suspending processes at code change

```
suspend(Mod, T) ->
  proc:fold_properties(
    fun({_, Pid}, Acc) ->
      case catch sys:suspend(Pid, T) of
         ok -> [Pid | Acc];
         _ -> Acc
      end
    end, [], {sasl, suspend_for, Mod}).
```

Not only would this version be significantly faster, it would also work for any Erlang processes, whether or not they are part of an OTP supervision tree.

Looking closer at this code, one may observe a simple publish/subscribe pattern. All that is needed is the subscribe function:

Code example 6: Registering modules for code change (idea)

```
suspend_for(Mod) ->
  proc:add_property({sasl, suspend_for, Mod}).
```

We have now managed to further enhance the release_handler to allow processes to add modules on the fly, for which they need to be suspended.

Furthermore, while the existing implementation is obscure and pretty complicated, requiring extensive study of OTP principles and the release handler for full understanding, the alternative solution, based on a common pattern, becomes more transparent. Designers easily spot this (previously hidden) property of a process, e.g. when calling the function proc:info(Pid, properties), and are also quickly able to retrieve these processes from an erlang shell prompt, e.g. using the function proc:pids({sasl, suspend_for, M}).

(While the above code might look contrived, it reuses the ETS select patterns, and can be quickly assimilated, especially if the process registry functions become the primary portal to all published process metadata in the system).

To further bash the existing solution, our proposed alternative is largely insensitive to the number of processes in the system. The process registry uses an ordered_set ETS table, so as long as we search for properties where at least the first parts of the property are known, the relevant subset will be extracted quite efficiently, rather than iterating through the entire supervision tree for each module.

6.2 Finding a given instance of a specific service

The example in section 2.2.1, finding the current shell evaluator, can be rather simply managed with the extended process registry. Indeed, since the code assumes only one current evaluator, we could do with a unique name, such as {shell, current_evaluator}. We could also opt for a publish/subscribe pattern as described in section 6.1.

Code example 7: Registering current evaluator (idea)

```
whereis_evaluator() ->
  proc:where({?MODULE, current_evaluator}).
```

7. Slippery Slope

The first prototype of the extended process registry mimicked the process dictionary, storing {Key, Value} tuples. This felt contrived when dealing with unique process names. It was also not obvious that it would be a good model for shared properties. As a result, the Value part was dropped, and only the Key part was kept.

However, a request to be able to store counters in the process registry was quickly put forth and while this felt like borderline abuse of the model, the request was granted. There were some good reasons: all call-handling processes need to maintain counters, and there were situations were iterating over the counters in a fashion similar to that of properties was desirable. Unfortunately, counters behave like properties most of the time, but also have a value. While some counters needed to be summarized in real-time, aggregated counters were subsequently added, which continuously maintain a total of all registered counters with the same name. To maintain the aggregated counters, some process needs to monitor the owners of individual counters, and update the aggregate accordingly if they die. Since the process registry had a process performing this monitoring already, it made perfect sense to install the aggregated counters there. As this was a last-minute request, the easiest way to do it was to add them a little bit on the side. Once added, an aggregated counter must be explicitly deleted, and is not included when properties and/or counters are queried.

While counters caused the ‘proc’ implementation to start cracking at the seams, we want to emphasize that counters are extremely useful. Among the built-in process properties, there are also counters (reductions), and, in general, value properties (heap_size, fullsweep_after, etc.). Indeed, the model should allow for value properties in order to be complete.

Another late requirement was to allow property registration by proxy. This called for an access control list to allow for some control of who added or removed properties for a given process. This was a concession to the practical problems of rewriting too much code, but made property registration more costly than it should be. Without the ability to register by proxy, the process could simply have inserted its properties into the central ETS table, without risking any race condition. This issue is analogous to that of the BIF register/2. As it is possible to register another process, it is quite easy to write code that is not timing-safe.

We recommend that a proper extended process registry would only allow processes to register their own unique names and shared properties. Registering of unique names must still be serialized, but registering of non-unique properties then becomes trivial. We will later show how registration by proxy can still be accomplished by adding an OTP system message.

8. Improved model and implementation (gproc)

The first implementation helped us understand which features were desired in our registry. We will now describe a second version, gproc [6] (where the “g” indicates the addition of global scope).

The main purpose is to index process metadata to allow efficient iteration over similar processes. We identify different types of metadata (properties), using the following distinctions:

- Names or properties
- Manual or automatic properties
- Global or local properties

8.1 Names or properties

We make the distinction between ‘names’, which are unique, and ‘properties’, which are not. Both names and properties have a key and a value part, for the sake of symmetry. We give counters special treatment, as they are commonplace and performance-critical. Since we have figured out how to maintain aggregated counters with minimal overhead, we allow this as a special category as well.

8.2 Manual or automatic properties

The emulator maintains a set of properties for each process. Since they are guaranteed to exist for all processes, it would seem that little is gained by indexing them. However, it would be useful to perform queries such as “list all processes running on high priority”, or “list all gen_servers in the system” (the latter would require exposing another implicit property of processes, which is vital to the understanding of program behaviour).

Some of the automatic properties change far too rapidly to allow any form of indexing. Examples are ‘reductions’ and ‘current_function’. If it were possible to call process_info() BIF from inside ets:select(), we could include these properties as if they *were* part of the index.

The aggregated counter is another example of an automatic property. There may be a generic pattern allowing for the safe and efficient handling of user-defined automatic properties, but if so, we have yet to discover it.

8.3 Global or local properties

So far, our own use of the process registry has been purely local. However, it should be noted that the global name server in OTP does indeed allow registration of structured names, as well as multiple registered names per process.

We make an effort to allow for registration of all categories mentioned above with either global or local scope.

8.4 Permissions

We impose the restriction that a process can only add, delete, and modify its own properties – never properties belonging to another process. Not only does this allow for a much more efficient implementation. It also removes a potential source of race conditions from the interface. This is a distinct departure from the existing ‘proc’ implementation, which allowed registration by proxy, but we can accomplish the same thing by extending the sys.erl module.

8.5 Symmetric data representation

One of the main advantages of the extended process registry is the support for searches á la ets:select() and ets:fold(). Unfortunately, properties, names and counters are not consistently represented, which led to a proliferation of search functions, special guards, etc.

We propose to weigh heavily on Erlang's pattern-matching capabilities, and use the following representation:

```
{Key, Pid, Value}
Key :: {Type, Context, Name}
Type :: n | p | c | a
Context :: g | l
```

Types and context identifiers are abbreviated mainly in order to get more compact query expressions and listings. There is no efficiency gain as atoms are passed by reference both locally and between Erlang nodes.

The types above are:

```
n = name
p = property
c = counter
a = aggregated counter
```

The contexts are simply:

```
g = global
l = local
```

We want to store all values in the same table in order to efficiently filter out objects based on different criteria. But to distinguish between unique and non-unique keys, we need to make a small concession: we must add something to the key to allow different processes to register the same key. It is simplest to use the Pid:

Code example 8: Storage structure of unique objects

```
{{Key, Pid}, Pid, Value}
```

To preserve symmetry, we wrap unique keys in a similar way, but insert a constant rather than a Pid. We chose the Type atom:

Code example 9: Storage structure of non-unique objects

```
{{Key, Type}, Pid, Value}
```

We want to hide this wrapper in the search functions, but this is a rather straightforward rewrite of the select pattern. As it needs to be performed at execution time, it will add startup overhead to any query from the Erlang shell.

8.6 Reverse mapping

For each object inserted, we also insert a reverse mapping object, {Pid, Key}. This allows us to efficiently retrieve all objects belonging to a given process, e.g. when the process dies. Writing multiple objects does not mean jeopardizing thread safety, if we're careful. The reverse mapping object is unique, and will only exist in that table if the regular object is there. Thus, a process can register a property atomically by calling the function:

Code example 10: Inserting property object and reverse mapping

```
ets:insert_new(
  ?TAB, [{{Key,self()}, self(), Value},
         {{self(), Key}}]).
```

This function call will return false if the object already existed. The 'proc' library raises an exception if a process tries to register a property twice. We see no reason to change that.

8.7 Handling monitors

If all registrations go through the server, it has the option to start a monitor for each object, but it only needs one monitor per process. It is not expensive to have lots of monitors, but it can be expensive to handle lots of 'DOWN' messages from each process.

We ensure that we have only one monitor for each process by inserting a marker in the table. We apply a test-set operation on the marker using ets:insert_new(), and set a monitor if needed.

Code example 11: Setting a monitor if required (server-side)

```
case ets:insert_new(?TAB, {Pid}) of
    false -> ok;
    true -> erlang:monitor(process, Pid)
end.
```

In the case where the client registers a property (and does not talk to the server, it performs a similar check:

Code example 12: Setting a monitor if required (client-side)

```
case ets:insert_new(?TAB, {self()}) of
    false -> ok;
    true -> cast(monitor_me)
end.
```

One obvious drawback is that we will never demonitor – once monitored, processes will stay monitored. We could address that by making the marker a counter, and doing cleanup when the counter reaches 0. But if the registry were to be fully integrated with Erlang/OTP, there will not be a need for an explicit monitor.

8.8 Aggregated counters

Aggregated counters are similar to normal counters, but are updated automatically, whenever a regular counter with the same name (and scope) is updated:

Code example 13: Updating aggregated counters

```
update_counter({c,g,_} = K, Incr) ->
  leader_call({update_counter,K,self(),Incr});
update_counter({c,l,Name}, Incr) ->
  Prev = ets:update_counter(
           ?TAB, {{c,l,Name},self()}, Incr),
  catch ets:update_counter(
      ?TAB, {{a,l,Name},a}, Incr),
  Prev.
```

To keep the overhead of update_counter() predictable, we have made aggregated counters unique. There will be at most one aggregated counter to update for each update_counter() call.

8.9 QLC

We have added a QLC table generator, to support the use of Query List Comprehensions. Apart from the obvious benefit of supporting very complex queries, QLC also makes all queries reentrant, hiding the continuation-passing loop that would otherwise be needed.

9. Demonstration

We made gproc part of the kernel application in an OTP R11B-5 build, and instrumented a few modules to automatically register some properties with gproc.

We illustrate how we can now rather easily display key characteristics of the system – in this case a node started with the command 'erl –boot start_sasl'.

Code example 14: Querying supervisor flags (using gproc)

```
=PROGRESS REPORT==== 5-Jul-2007…
        application: sasl
        started_at: nonode@nohost
Eshell V5.5.5  (abort with ^G)
1> Q1 = qlc:q([{P,Fs} ||
    {{p,l,supflags},P,Fs} <-
       gproc:table(props)]).
{qlc_handle,{qlc_lc,…}}
2> qlc:eval(Q1).
[{<0.10.0>,{one_for_all,0,1}},
 {<0.27.0>,{one_for_one,4,3600}},
 {<0.32.0>,{one_for_one,0,1}},
 {<0.33.0>,{one_for_one,4,3600}}]
```

Here, we obtain a concise listing of the supervisor restart strategies in the system. This also implicitly gives us all supervisors, but that particular information can also be retrieved by searching on behaviour:

Code example 15: Finding all processes with behaviour: supervisor (using gproc)

```
3> Q2 = qlc:q([P ||
     {{p,l,behaviour},P,supervisor} <-
        gproc:table(props)]).
{qlc_handle,…}
4> qlc:eval(Q2).
[<0.10.0>,<0.27.0>,<0.32.0>,<0.33.0>]
```

Going back to the example in Chapter 5.1, we have thus far registered information that *almost* gives us the answer we seek:

Code example 16: Finding processes to suspend (using gproc)

```
8> rr(code:lib_dir(stdlib) ++
      "/src/supervisor.erl").
[child,state]
12> Q3 = qlc:q([{C#child.pid,S,
                C#child.modules} ||
    {{p,l,{childspec,_}},S,C} <-
       gproc:table(props)]).
{qlc_handle,…}
13> qlc:eval(Q3).
[{<0.34.0>,<0.33.0>,dynamic},
 {<0.20.0>,<0.10.0>,[code]},
 {<0.19.0>,<0.10.0>,[file,file_server,
                    file_io_server,
                    prim_file]},
 {<0.18.0>,<0.10.0>,[global_group]},
 {<0.12.0>,<0.10.0>,[global]},
 {<0.9.0>,<0.10.0>,[gproc]},
 {<0.16.0>,<0.10.0>,[inet_db]},
 {<0.26.0>,<0.10.0>,[kernel_config]},
 {<0.27.0>,<0.10.0>,[kernel]},
 {undefined,<0.10.0>,[erl_distribution]},
 {<0.35.0>,<0.33.0>,[overload]},
 {<0.36.0>,<0.32.0>,[]},
 {<0.11.0>,<0.10.0>,[rpc]},
 {<0.33.0>,<0.32.0>,[sasl]},
 {<0.21.0>,<0.10.0>,[user_sup]}]
```

We note that the information needed for the release handler is readily available, as is the rest of the child start specification.

The sketched solution in Chapter 5.1 is difficult to implement in practice, since the supervisor module does not get to execute code in the child's process. Thus, it cannot easily insert a property in its context. A more serious disadvantage to this is that the supervisor cannot tell a restarted process why it restarted, or how many times it has done so. This makes it difficult to do anything different during an escalated restart. Using gproc, the supervisor could store such information in its own context, and make the information available.

9.1 Registration by proxy

We have also modified the sys module so that a process could be asked to register a property:

Code example 17: Registration by proxy (using gproc + sys)

```
1> gproc:info(whereis(gproc),gproc).
{gproc,[{{p,l,behaviour}, gen_leader}]}
2> sys:reg(gproc, {p,l,test}, 17).
true
3> gproc:info(whereis(gproc),gproc).
{gproc,[{{p,l,behaviour},gen_leader},
        {{p,l,test},17}]}
```

The function sys:handle_system_msg/6, which is the recommended way to handle system messages, will call the function Mod:system_reg(Misc, Key, Value), if exported; otherwise, it will try to register directly. The callback allows behaviours to selectively disable registration by proxy.

9.2 Performance

It is important to assess the cost of collecting and indexing metadata. While it would be very convenient to have all forms of process and application characteristics indexed and readily available, the overhead involved might not be justifiable. For demonstration purposes, we used gproc to publish behaviour information, child specifications, supervision flags, etc. This served the point of illustrating how much can potentially be harvested.

For the cost comparison, we compare performance with a lightweight alternative (e.g. normal registration, or storing a simple object in an ets table), and with sysProc, the predecessor to gproc.

9.2.1 Property registration

Lightweight alternative: storing a mapping and a reverse mapping in an ets table. We measure storing 100 such objects (pairs).

Ets	gproc	sysProc
1	3.0	5.4

9.2.2 Local name registration

Lightweight alternative: the register()/unregister() BIFs. We measure the cost of registering and unregistering a name 100 times, as the BIFs cannot register multiple names.

register BIFs	gproc	sysProc
1	68.8	74.7

The big difference here can be attributed to gproc and sysProc having to use a gen_server to prevent race conditions, while the register BIFs execute atomically inside the VM (in the non-SMP version, even without mutexes). If the ability to register any Erlang term, rather than just an atom, would be introduced as a BIF, a slight performance penalty should be expected, but not nearly as much as these figures suggest.

It should also be noted that register()/unregister() is extremely cheap. On most modern machines, even in gproc and sysProc, local name registration takes significantly less than 100 usec.

9.2.3 Global name registration

Lightweight alternative: global:register_name(). We measure with four nodes running on the same machine, doing 100 register/unregister operations, since 'global' actually issues a warning each time a name is registered to a process that already has a registered name.

Global	gproc	sysProc
1	0.16	N/A

Obviously, the "lightweight" alternative is not so lightweight after all. Global registration using gen_leader is much faster than the voting mechanism used in global. sysProc has no global registration facilities.

10. Future work

At IMS Gateways, it still remains to be determined whether the gproc prototype can replace the existing extended process registry. It would also be interesting to study whether the local part of gproc could be implemented in the runtime system, for better performance.

The gproc prototype is based on the gen_leader behaviour. Gen_leader was developed by Arts and Wiger [7], and subsequently re-written and verified by Arts and Svensson [8]. Gen_leader makes it quite straightforward to implement a set of servers that manage both global and local scope.

While developing the gproc module, we encountered a problem with gen_leader: if there is only one known leader candidate, gen_leader will still enter an election loop and wait for responses from other candidates. Leader election is triggered by such responses, which will, of course, never arrive. Luckily, the fix is trivial: when there is only one candidate, that candidate can elect itself and skip the negotiation.

A more serious problem with using gen_leader is that it requires advance knowledge of all candidate nodes. However, in our prototype integration with OTP, we want to start the process registry as soon as possible – before the Erlang distribution is started. To do this, we modified gen_leader to start in "local-only" mode, and wait for the user to trigger election mode. This extension of gen_leader has not been verified, but seems to work satisfactorily for the purposes of the prototype.

Perhaps even more serious is gen_leader's lack of support for dynamically reconfigured networks, and for de-conflicting the states of two leaders (which is presumably the most difficult part of adding nodes on the fly). Adding this type of support to gen_leader is a topic for further research. We hope that this prototype will be able to spark interest in such an effort.

Acknowledgments

I want to thank John Hughes and Thomas Arts, for ruthlessly revealing fatal flaws, using their amazing QuickCheck tool, in the very first prototype; to Thomas Arts and Hans Svensson for brilliant work with gen_leader; to Kurt Jonsson and Mats Cronqvist for many fruitful design discussions; and to Bo Fröderberg, Bo Bergström and Mats Andersson for interpreting the specific requirements from the application development side.

References

[1] Ulf Wiger, "Four-fold Increase in Productivity and Quality", FemSYS, Munich, Germany, 2001
http://www.erlang.se/publications/Ulf_Wiger.ppt

[2] Ulf Wiger, "Structured Network Programming", Erlang User Conference, Stockholm, Sweden, 2005
http://www.erlang.se/euc/05/1500Wiger.ppt

[3] Mats Cronqvist, "Troubleshooting a Large Erlang System", ACM SIGPLAN Erlang Workshop, Pittsburgh, USA, 2004
http://doi.acm.org/10.1145/1022471.1022474

[4] Erlang/OTP Design Principles,
http://www.erlang.org/doc/design_principles/part_frame.html

[5] Thomas Lindgren, "Atom Garbage Collection", ACM SIGPLAN Erlang Workshop, Tallinn, Estonia, 2005
http://doi.acm.org/10.1145/1088361.1088369

[6] Ulf Wiger, gproc Source Code Repository (Subversion)
http://svn.ulf.wiger.net/gproc

[7] Arts, Claessen, Svensson, "Semi-formal Development of a Fault-Tolerant Leader Election Protocol in Erlang" *Fourth International Workshop on Formal Approaches to Testing Software*, volume 3395 of *LNCS*, pages 140-154 Linz, Austria, 2004
http://www.ituniv.se/~arts/papers/fates04.pdf

[8] Svensson, Arts, "A New Leader Election Implementation" ACM SIGPLAN Erlang Workshop, Tallinn, Estonia, 2005
http://doi.acm.org/10.1145/1088361.1088368