Extending the VoDKA architecture to improve resource modelling[*]

Juan José Sánchez Penas
LFCIA, Computer Science Department,
University of Corunha
Campus de Elvinha S/N. 15071, A Corunha,
Spain
juanjo@lfcia.org

Carlos Abalde Ramiro
LFCIA, Computer Science Department,
University of Corunha
Campus de Elvinha S/N. 15071, A Corunha,
Spain
carlos@lfcia.org

ABSTRACT

VoDKA is a Video-on-Demand server developed using Erlang/OTP. In this paper, the evolution of the core architecture of the system, designed for improving resource modelling, is described. After explaining the main goals of the project, the steps taken towards an optimal architecture are explained. Finally, a new architecture is proposed, solving all the problems and limitations in the previous ones. Special attention is paid to the use of design patterns, implementation behaviours, and reusable software components.

Keywords

multimedia servers, cluster computing, functional programming, distributed computing, design patterns

1. INTRODUCTION

The VoDKA [8, 4] (vodka.lfcia.org) system is a Video-on-demand server with a complex storage architecture that has been developed during the last years at the LFCIA Lab. During this period, the core system architecture has evolved from a more static one to a very flexible and dynamic solution, learning in each of the intermediate versions from the errors of the previous ones. In this paper, an analysis of the core system architecture is described, paying special attention to the new architecture, currently a prototype, proposed for its integration with the next main version of the system.

A VoD server is a system that provides to the end user an on-demand media object service. Using a multimedia application with support for one of the typical media streaming protocols, any user can request a media object to the server at any moment. The system should be able to stream the media object to the end user minimising the response time and, if possible, giving the user an estimation of the waiting time (time from the user request to the actual reception of the media).

VoDKA is a modular software developed using Erlang/OTP. The use of a distributed, high level, declarative language has made the frequent prototyping and restructuring of the system possible. Several of the main parts of the server have been successfully rewritten when new requirements came into the project.

The system has been developed in cooperation with a cable & telecommunications company called R, that plans to start offering the VoD service to the end user in the near future. VoDKA is now in production stage, at this moment working only as a prototype experiment for a selected group of users.

The main goals of the project are:

- To build an scalable (both upwards and downwards) architecture. This means that the system architecture should allow configurations where the requirements were minimum, and configurations where the server should be able to handle thousands of users and media objects.
- To build a distributed, fault tolerant system. It is an obvious requirement that the system should work 24x7, so features like hot code replacing and failure recovery are a main need for the system.
- To design a very flexible system, adaptable to the underlying network topology and multiple end-user protocols

Two main technologies were proposed as the base to achieve the goals:

- To use a high level, distributed functional language combined with design patterns [7, 6] for the analysis, design and implementation (behaviours). Erlang and the Open Telecom Platform were a really good candidate for developing such a system.
- To use an adaptation of the Beowulf architecture [3] as underlying hardware. This would be a good way of fulfilling the goals of the project whitout the need of a very expensive hardware solution.

[*]This work has been partially supported by Spanish MCyT TIC 2002-02859 and by the Galician Government PGIDIT03PXIC10502PN

Even though VoDKA is normally described as only being a VoD server, it is a distributed system composed by many modules with different responsibilities. Typically, these modules cooperate to define the following subsystems: the *management subsystem*, in charge of all the accounting processing and media object selection; the *streaming subsystem*, which sends the media object to the end user with a given quality and protocol supported by the user connection/application; and the *core subsystem*, which contains the scheduling and storage subsystem, and is the most innovative part of the server. For the analysis of the architecture, the focus will be on the last subsystem, which is closely related to the internal process architecture and communication protocols.

During the design and implementation of the system, the development team has put a lot of effort in the use of design patterns and implementation behaviours. All the shared functionality of the processes has been progressively abstracted into implementation behaviours, now part of a *behaviour library* of the VoDKA system (e.g. `trader`, `rgs`, `pipe`). Most of these abstractions are going to be described in the following sections of the paper.

2. CURRENT SYSTEM ARCHITECTURE

Before describing the architecture proposal for the next version of the server, we are going to explain the evolution of the system and the main limitations of the current internal protocols and architecture.

2.1 Initial three-tier architecture

The initial proposal for achieving the system goals was a three level architecture: streaming, cache and storage. The first one would be in charge of the protocol adaptation and the actual streaming of the media object. The second one would reduce the performance requirements of the massive storage subsystem, in a similar way to the computers memory architecture. And the last one would reach the obvious requirement of having capacity for an enormous amount of media objects, that should be available for the user at any moment. The specialisation of each of the architecture levels (with different requirements for each of them) allows the system to meet all the VoD requirements without increasing excessively the price of the final system.

2.2 Flexible multi-tier architecture

Soon the limitations of the described, fixed architecture were met. A new multi level architecture, based in the chain of responsibility [7] pattern and a common internal communication protocol for all the levels of the system was chosen. Each of the previous levels can be seen now as a component (a black box with a clear functionality for the rest of the system) with a well known communication API.

This new flexible architecture would be a combination of, mainly, four different *components* of the system:

- *Streaming component*: Its functionality is related with the protocol adaptation with the end user. The communication between the components inside the server is done by using internal, optimal (more adapted to the needs of the internal inter-component communication that the general purpose TCP/IP different protocol) communication protocols. However, the end user is going to request a given movie object with a concrete quality (bandwidth use profile) and one of the typical streaming protocols (RTP, H.263, HTTP, etc.). The streaming component is composed by three types of processes: the *streaming scheduler*; the *streaming group*; and the *protocol frontends/streamers*. The different frontends for each of the protocols receive the media object requests from the user; for each request, they build a *lookup message* ({`lookup, MediaObject, Profile`}, where *profile* is a description of the bandwidth quality requested by the user) and send it to the *streaming group*, which follows-up the request to the *streaming scheduler*, following a chain of responsibility pattern internal to the component. The streaming scheduler defines the logic for the whole component: it knows how to contact the next level (normally acting as an intermediate cache) and sends the request, waiting for an answer of the form: ({`lookupAns, Options`}). *Options* is a description of the possible sources from which the requested media object can be streamed to the end user. On the way back, the streaming scheduler could decide to choose among some of the obtained options. Once the source is chosen, the *getmo message* is sent following the same way through the system.

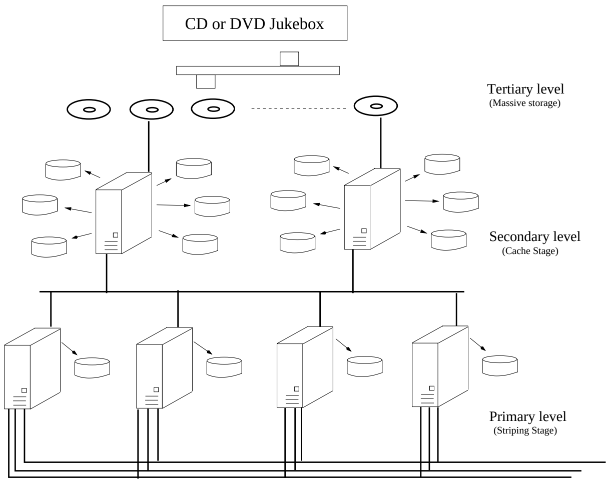

Figure 1: Initial three level architecture

- *Cache component*: The reason for configuring the server deployment using this component between two other components is to reduce the system requirements, by placing the media objects used more often closer to the user (the user could also be another component placed in a higher level of the system). The cache component is composed by three types of processes: the *cache scheduler*; the *cache group*; and the *cache drivers*. The *cache scheduler* receives the *lookup* and *getmo* messages from the previous level and implements the following protocol: (1) asks to the *cache group* about the availability of the media object inside the local cache of the component; if the MO is there, answers with a *lookupAns* message to the previous level/component; (2) in case the media object cannot be found locally, the request is forwarded to the next component. The *cache group* acts as a common facade for all the drivers. The *cache drivers* have all the implementation of the typical cache algorithms, and all the information about how (quality), where and which media objects are available.

- *Storage component*: It is quite similar to the cache component in its internal structure, but its main goal is to be able to store a huge amount of information. Thus, the bandwidth and response time requirements are lower than the ones for the intermediate level components like the cache one. This component has also the same three types of processes: *storage scheduler*, *storage group*, and the *storage drivers*. One storage driver could be, in a complex configuration of the system, a connection to another VoDKA server running in a different location, i.e. one server can act as storage device for the other.

- *Resource constraint subcomponent*: The VoDKA hardware architecture configurations can be really quite heterogeneous. One can, for example, run each of the components in a different set of GNU/Linux machines; in this kind of systems, a lot of access and communication constraints need to be defined to avoid system overload. In order to *plug a constraint* at any point of the system, both in the inter-component communication and inside each of the components, an special simple component with only one process was defined. The constraint process implements the same API than the rest of the components of the system. A restriction about the bandwidth of the network that connects two components can be placed between them putting a transparent proxy process (decorator [7] design pattern) that implements this restriction and reduces the *options* answered from a component to the higher level, depending on the available resources at each moment.

Once the *lookup* and *getmo* messages go all the way through the system architecture, the actual transmission of the media object to the end user and (possibly) between different components of the system takes place. With that purpose, the source and destination components create a couple of *pipes* that communicate in what is called a *transfer*. A *pipe* is a communication abstraction between a source and a destination, and a *transfer* is a combination of two pipes where the destination of the first one communicates with the source of the second one.

The different components can be combined in different ways depending on the needs for each of the deployments. Each of these combinations are called *configurations* of the system. A frequently used and simple configuration would be the one composed by one *storage component*, one or more cache components, and one streaming level, some of then combined with different resource constraint subcomponents.

This architecture has proved to be really interesting for the needs of the VoDKA project, but still has some limitations that should be solved:

- The protocol is not optimal and uniform: sometimes too many messages are sent to the different nodes of a system configuration in order to obtain all the source options for streaming the media object.

- Limited constraints configuration: there are places in a system configuration where the resource constraints

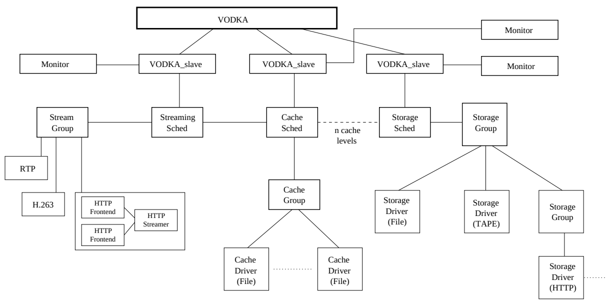

Figure 2: Configuration example

cannot be placed, thus making impossible to deploy some of the configurations. There are problems with the sharing of constraints between different processes of the system.

- Behaviours are half-abstracted only: some of the code that is shared by different modules can still be abstracted, giving place to new behaviours.

- Filter and scheduling architecture is not clear and consistent.

In general, after including progressively a set of nice features and design innovations to the system, the whole architecture needed to be redefined in order to mature all these modifications. The next architecture of the server, therefore, tries to conciliate the new design choices with the solutions to the previous architectures' limitations.

3. PROPOSED SYSTEM ARCHITECTURE: VODKA ARCHITECTURE 2.0

3.1 Architecture components

The main components of the new architecture are the *streaming component*, the *cache component*, and the *storage component*. All of them can be connected in a system configuration (graph), and deployed in different hardware architectures. The constraint component disappears as an independent process and becomes part of the component structure; a detailed description of the new constraint architecture can be found in the following sections. Two more components are part of a system configuration: the *monitoring component* and the *application component*. The former works as described in the observer design pattern, and can be connected to any component in order to monitor all the events that are happening to the component during its working time; this component is an evolution of the monitoring subsystem presented in [9]. The later can be seen as a complex supervision tree, that knows about the whole system architecture and takes care of the actions needed when some of the parts of the system have crashed (this component follows the design principle defined for the Erlang/OTP platform). An example of how the system components can be configured in a concrete server deployment can be seen in figure 3.

3.2 Internal communication protocol

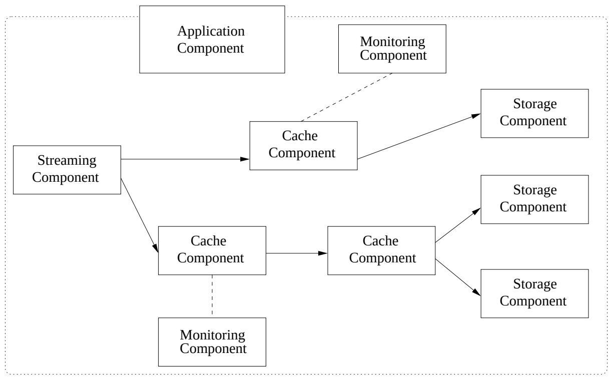

Figure 3: VoDKA components configuration example

The internal protocol is as simple as possible. The traders (all the intermediate processes in the system architecture) are Erlang gen_servers with four different services: *lookup*, *getmo*, *alloc* and *putmo*.

Traders are connected using the chain of responsibility pattern [7], whose goal is to avoid coupling of request and receiver, giving to a set of objects the chance to handle the request; the first object in the chain gets the request and can either handle it or reject and pass to the next object in the chain. The same is repeated until one of the objects answers or the end of the chain is reached. In the case of the VoD server, the request is satisfied if the MO is in local storage and, therefore, one of the local devices can act as source for the data transmission.

Four couples of messages are used:

- Lookup messages, checking the media object availability:
  - {`lookup, MO, Profile`}: `MO` is a unique identifier of the media object that it is going to be streamed; `Profile` describes the quality (bandwidth) range requested by the user.
  - {`lookupAns, Options`}: `Options` describes all the possible sources (provider processes) for the media object.
- Get messages, asking for the actual transmission of a media object:
  - {`getmo, MO, PipeOpts`}: Once one of the options is selected, the getmo message is sent through the system architecture in order to check if the resources are still available (if that is the case, the allocation of resources to the transmissions is carried out). The `PipeOpts` describes the internal transmission options for the synchronisation of the processes that are going to communicate.
  - {`ok, Pid`}: Confirmation of the transmission, indicating the process identifier of the source from which the media object is going to be sent.
- Alloc messages, checking if there are resources available to store a new media object in a storage-type process:
  - {`alloc, MO, Profile`}: similar to the `lookup` message, the media object and the bandwidth range are described in order to know if there are resources available to do the actual transmission.
  - {`allocAns, Options`}: `Options` describes all the possible destinations for the media object.
- Put messages, asking for the actual transmission of a media object in order to store it:
  - {`putmo, MO, PipeOpts`}: similar to the `getmo` message, but now in order to store a new media object in a given location, with the transference options described by PipeOpts.
  - {`ok, Pid`}: Confirmation of the transmission, indicating the process identifier of the destination where the object is going to be stored.

All these couples of messages are sent all the way through the system architecture. The resources are only booked with the *getmo* and *putmo*; there is not any kind of reservation during the *lookup* and *alloc*.

3.3 Process architecture: implementation patterns

The main implementation behaviours (new Erlang behaviours, some of them designed ad hoc for this application and some others quite general and reusable) proposed for this architecture are:

- RGS (Reflective Generic Servers): It is an extension of the Erlang/OTP `gen_server` with metainformation and an standard API that lets any other process ask to a RGS about its capabilities and its internal state. One interesting use of this kind of pattern is to build generic components that can ask a given server for the supported services. In VoDKA, for example, a service invocation interface has been developed using these introspection capabilities. This is also very useful for system monitoring.

  In order to save the metainformation for the server process, a data structure based on Erlang records is defined:

```
-define(RGS_TAG, rgs).
-define(NO_NAME, "My RGS").
-define(NO_DESC, "No description available").
-define(NO_CHECK, fun (_) -> ok end).
-define(ANY_TYPE, any).
-define(TYPE_CHECK, type_check).
-define(GEN_PRETTY, fun (X) -> rgs:gen_pretty(X) end).
-define(GEN_PARSER, fun (X) -> rgs:gen_parser(X) end).
-define(GEN_CLASS, ?MODULE).
-define(MANDATORY, '$MANDATORY').

-record(metainfo,
        {
          name       = ?NO_NAME,
          desc       = ?NO_DESC,
          class      = ?GEN_CLASS,
          implements = [],    %% interfaces
          properties = [],    %% read-only attributes
          methods    = []     %% available services
         }).

-record(property,
        {
          name,
          desc    = ?NO_DESC,
          pretty  = ?GEN_PRETTY, %% pretty-printer
          type    = ?ANY_TYPE    %% type description
          }).

-record(method,
        {
          name,
          desc    = ?NO_DESC,
          params  = [],          %% required parameters
          check   = ?NO_CHECK,   %% check parameters
          pretty  = ?GEN_PRETTY,
          type    = ?ANY_TYPE
         }).

-record(param,
        {
          name,
          desc    = ?NO_DESC,
          type    = ?ANY_TYPE,
          default = ?MANDATORY,
          pretty  = ?GEN_PRETTY,
          parser  = ?GEN_PARSER, %% generic parser
          check   = ?TYPE_CHECK  %% check parameter
         }).
```

In order to implement the reflective generic server as a new Erlang behaviour, an extension of the `gen_server` interface is defined using the behaviour_info function:

```
behaviour_info(callbacks) ->
   [{rgs_meta,1}, {get_property,2},
    {get_config,1}, {get_links,1},
    {configure_links,2},
    {init,1}, {handle_call,3},{handle_cast,2},
    {handle_info,2},{terminate,2},{code_change,3}].
```

The new properties need to be defined by an implementation of the RGS behaviour in order to be able to introspect the metainformation from the generic code. The main part of this Erlang behaviour implementation, related with the message management, is similar to the following source code:

```
handle_call(?META_TAG, From, {Module,State}) ->
    {reply, Module:rgs_meta(State), {Module,State}};
handle_call({?GET_PROPERTY_TAG, PropName}, From,
                                    {Module,State}) ->
    {reply, Module:get_property(PropName, State),
            {Module,State}};
handle_call({?CALL_TAG, MethodName, Params}, From,
                                    {Module,State}) ->
    case catch dispatch(MethodName, Params,
                        Module, State) of
        {'EXIT', {shutdown, Reason, Response, NewState}} ->
            {stop, Reason, Response, {Module,NewState}};
        {'EXIT', Reason} ->
            {reply, {error, Reason}, {Module,State}};
        {Reply, NewState} ->
            {reply, Reply, {Module,NewState}}
    end;
handle_call(?GET_LINKS, From, {Module,State}) ->
    {reply, Module:get_links(State), {Module,State}};
handle_call(?GET_CONFIG_TAG, From, {Module,State}) ->
    {reply, [ {module, Module}, {node, node()} |
              Module:get_config(State) ],
             {Module,State}}.

handle_call(Message, From, {Module,State}) ->
    case Module:handle_call(Message, From, State) of
        {reply, Reply, NewState} ->
                {reply, Reply, {Module,NewState}}
    end;
```

When the received message is directly related with the RGS metainformation, the local code is used; in other cases, the request is forwarded to the `handle_call` of the callback module. The same should be done for all the `gen_server` functions.

- Trader: The internal protocol messages need to be understood by all the modules in the core system. Instead of repeating all the code, the protocol implementation is abstracted in a behaviour that trades in order to obtain the source and destination for each of the transmissions inside the system. The Product Trader is a well known design pattern [5].

  The structure of the code used to define the trader behaviour, an extension of the RGS, is similar to the idea depicted in the RGS behaviour. The interface is extended again, using `behaviour_info`, in order to include the new three mandatory functions for an implementation of the trader behaviour:

```
behaviour_info(callbacks) ->
   [{rgs_meta,1}, {get_property,2}, {get_config,1},
    {get_links,1}, {configure_links,2},
    {init,1},{handle_call,3},{handle_cast,2},
    {handle_info,2},{terminate,2},{code_change,3},
    {lookup,3}, {getmo,3}, {playmo,3}, {alloc,3}].
```

The `handle_call` is in charge of parsing the received message tags and decide which is the function of the callback module that should be called. If the received message is not one of the known ones, the request is passed forward to the callback implementation.

```
handle_call({lookup,MO,Options}, From,
            {Module,State}) ->
    {reply, Module:lookup(State,MO,Options),
            {Module,State}};
handle_call({getmo, MO, PipeOpts}, From,
            {Module,State}) ->
    {reply, Module:getmo(State,MO, PipeOpts),
            {Module,State}};
handle_call({alloc, MO, PipeOpts}, From,
            {Module,State}) ->
    {reply, Module:alloc(State,MO, PipeOpts),
            {Module,State}};
handle_call({putmo, MO, PipeOpts}, From,
            {Module,State}) ->
    {reply, Module:putmo(State, MO, PipeOpts),
            {Module,State}};

handle_call(Message, From, {Module,State}) ->
 case Module:handle_call(Message, From, State) of
      {reply, Reply, NewState} ->
              {reply, Reply, {Module,NewState}}
      ...
 end;
```

Most of the other functions are going to be wrappers, forwarding the work to the callback module. Nevertheless, in some cases, generic code is placed directly in the behaviour module. As an example: when building the `meta_rgs` function for the trader generic code (behaviour), it is easier to call to the function `Module:rgs_meta(State)`, being `Module` the variable with the callback module name, and then to add to the metainformation the description of the four generic functions for the behaviour.

The pipe options, that are used in the traders in order to specify transmission conditions, is defined by the following record structure:

```
-record(pipe_options,
        {
          notify = {monitor, no_notify},
          policy = normal,
          type = transfer,
           id = {generic, now()},
           trigger = {inf, none},
           blockSize = 16384,
           range = complete
          }).
```

The lookup operation returns a list of trade options (which is also used through the other three trading functions), whose structure is depicted in the following Erlang record:

```
-record(trade_option,
        {
         pid,        % process that offers the MO
         transfer,   % protocols to be used
         moinfo,     % information for the MO
         cost,       % cost of transfer
         throughput, % estimated throughput
         start       % estimated time to start
         }).
```

Therefore, `trader` behaviour is defined as an extension of `rgs` behaviour, which is another extension of the standard `gen_server`. Each time we do an extension, new functions are added to the interface, and the name of the callback module is handled as part of the server state. Besides this interface extension, we could think in two parts for the global state: one for the callback module and the other one for the generic implementation (this option of splitting the state is still not used in the current implementation). With these two features in combination, we could implement something in some sense equivalent to the inheritance of the object oriented world, being able to define new behaviours as an extension of a previously defined one, adding functions (object methods) and increasing the state information (object properties).

- Pipe & Transfer: They are a data movement abstraction for the internal data communication [8]. A *pipe* has as its creation parameters the source and destination (both Erlang modules that implement the send and reception protocols plus some initialisation parameters), and some general options about the way the transmission must be done. The Erlang modules implement three mandatory functions: `init` (protocol initialisation), `proc` (read and write) and `done` (destructor). The *pipe* works as a supervisor: if there is any problem with the transmission, it propagates the error. By using the *pipe* abstraction, any two levels of the server can be interconnected. Indeed, a whole server could play the role of a source for the tertiary level of other server.

  An implementation of a trivial *pipe* is depicted in the following code example:

```
init({DS_module,DS_initparam},
     {DD_module,DD_initparam}, _) ->
    {ok, DS_info, DS_state0} =
                DS_module:init(DS_initparam),
    {ok, DD_state0} =
                DD_module:init(DS_info, DD_initparam),
    {DS_statef, DD_statef} =
        pipe_while(DS_module, DS_state0,
                   DD_module, DD_state0),
    ok = DS_module:done(DS_statef),
    ok = DD_module:done(DD_statef).

pipe_while(DS_module, DS_state,
           DD_module, DD_state) ->
    case DS_module:read(DS_state) of
        {ok, Data, DS_statenext} ->
            {ok, DD_statenext} =
               DD_module:write(Data, DD_state),
            pipe_while(DS_module, DS_statenext,
                       DD_module, DD_statenext);
        {done, DS_statef} ->
            {DS_statef, DD_state}
    end.
```

  Different kinds of *pipes* with different features (kind of bitrate, for example) have been implemented in the real system, but the base idea is always the same one. In order to abstract a little bit more the source code of the server, source and destination of the pipe could also be implemented using Erlang behaviours, specifying the three described functions as the mandatory ones for the interface of the callback module. Even the generic *pipe* source code could be abstracted into a generic behaviour, using this behaviour with the right callback module in order to implement different strategies.

- Scheduler: In order to decide which of the obtained source options for the media object are going to be chosen, the *cost* of each of them is calculated when the protocol messages are sent through the system. In the scheduling processes of the system, two kind of functions need to be evaluated each time one of that protocol messages are handled, one for updating the cost, depending on the available resources of the component, and the other one for selecting which options are going to be filtered out. Thus, in all these processes, abstracted to the scheduler behaviour, the functions *filter* and *combine_cost* need to be implemented.

  A new Erlang behaviour, called `scheduler`, an extension of the `trader behaviour`, can be defined with the following `lookup` function (one of the mandatory functions for an implementation of a trader):

```
lookup({Module,[ResourceManager, UsedResources,
                Filter | State]}, MO, PipeOptions) ->
   {lookupAns, Options} =
     Module:lookup(MO, PipeOptions, State),
   NewOptions =
     filter(ResourceManager, UsedResources, PipeOptions),
   {lookupAns, Options}.
```

  In this function, it is assumed that the state includes information about the resource manager process, the identifiers of the resources used by the scheduler, and the module implementing filtering and cost functions. Inside the `filter` function, the cost is updated according to the state of the process and the related resources:

```
filter (ResourceManager, UsedResources, Options) ->
  UpdatedCostOptions =
     Filter:combine_cost (ResourceManager,
                          UsedResources, Options);
     Filter:filter(ResourceManager,UsedResources, Options).
```

  This new behaviour makes mandatory for the callback module to give a `Filter` module at initialisation time in order to call the right functions to filter and combining the cost. This filter can be changed without stopping the system using the message {`change_filter, Filter`}, that is going to be managed by the `handle_call` of the callback module.

3.4 Constraint architecture

One of the biggest and more important changes in this new system architecture is the way the resource restrictions (or constraints) are managed. The process oriented methodology followed in the previous versions of the architecture has some limitations when trying to model complex restrictions. As restrictions where simply decorators included in the responsibility chain, a problem appeared when a resource needed to be shared between two different chains that were disjunct after that resource. There were also problems when one resource needed to be used in two different points of a chain. Some workarounds could be used to try to solve this, but not for all the possible system configurations.

The new architecture is based on a *component oriented constraint management*. Each of the system components (normally associated with a physical computer in the deployment) is going to have a *constraint manager process* that is going to handle all the restrictions for that component. The constraint manager stores a resources table with the following information: {`resourceId, cost_function, check_avail, free_resources, alloc_resources ResourceState`}. This information (data and functions) can be stored in a distributed database, helping this way to obtain the fault tolerance for the server.

Each of the processes inside the component (storage scheduler and device drivers) is associated with a list of resources that are used by that process. For any checking or resource booking, the devices or intermediate traders/schedulers have to communicate with the constraint manager of their component. As the data is centralised, the restriction composition is easy to implement, and more complex restriction configurations are possible.

In order to implement this new architecture, a modification of the `trader` is proposed, where the state is extended to handle the resource manager and the list of used resources. A new message needs to be managed by the `handle_call` in order to be able to free the resources used when a transmission actually ends.

```
handle_call({lookup,MO,Options}, From,
      {Module,[ResourceManager, ResourceList | State]}) ->
    Answer = case check_available(ResourceManager,
                            ResourceList, Options) of
       ok -> Module:lookup(State,MO,Options)
       error -> {error, resources_not_available}
    end,
    {reply, Answer, {Module,State}};

handle_call({getmo, MO, PipeOpts},
            From, {Module,State}) ->
    Answer = case alloc_resources(ResourceManager,
                            ResourceList, Options) of
           ok -> Module:lookup(State,MO,Options)
           error -> {error, resources_not_available}
    end,
    {reply, Answer, {Module,State}};

...

handle_call({free, MO, PipeOpts}, From,
      {Module,[ResourceManager, ResourceList | State]}) ->
       free_resources(ResourceManager,
              ResourceList, PipeOptions)
       {reply, ok,
  {Module,[ResourceManager, ResourceList | State]}}
```

As we have depicted previously, the filtering and cost combination is an important part of the new architecture. The decision making processes (schedulers in each of the components) have to implement the filter and cost combination functions. The scheduler continues to be completely distributed among the different components of the system, but the behaviour of this part of the server is now much more homogeneous. An example of the messages exchanged by the processes is depicted in figure 4.

In figure 5, an example of a simple configuration of the new architecture is shown.

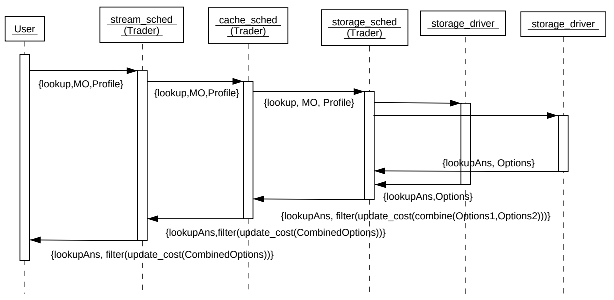

Figure 4: Sequence diagram for a simple responsibility chain

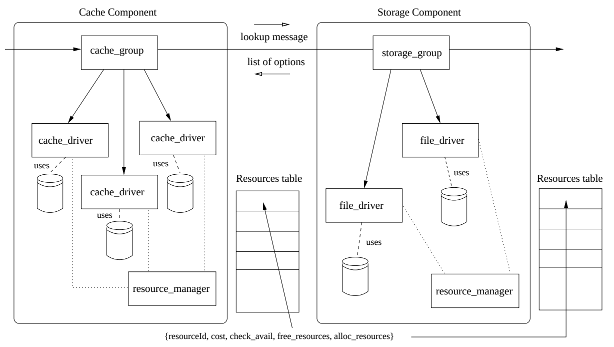

Figure 5: VoDKA architecture 2.0

4. ERLANG TECHNOLOGY AND VODKA

The use of Erlang/OTP as the development platform for the VoD system was one of the main design choices. The main advantages of using Erlang technology were:

- The nature of the language: using a high level distributed functional language makes easier to develop a distributed (soft) real-time system like the VoD server we are describing. The process oriented nature of the language helped in dividing the system architecture into separate components, communicating by message passing.
- Short time-to-market: due to the fact that this system has been developed in cooperation with a company, the fact of been able to have prototypes very soon was also a good advantage of using the language. In fact, during the project development, some important parts of the system could be completely rewritten in order to meet new requirements.
- Lot of libraries and design principles: the OTP set of libraries were really powerful for the needs of the

project, and the set of design principles were part of the design of the system kernel since the first stages of the project. In fact, most of the main components of the system are Erlang implementation behaviours, either based in the original behaviours or implemented as new ones in the language.

- Interface with low level languages: due to the performance needs or to the need of native interfaces with some devices, a system should need to interact with low level languages. Erlang interfaces have shown to be powerful enough for the needs of the VoDKA system development.

- Very good performance: Erlang performance was very good in the system tests both with real users and in simulations. When the project started, the design team thought that the I/O and other low level operations should be carried out by C modules, due to performance requirements. In the first prototypes, those modules were developed, due to time restrictions, using only pure Erlang. Surprisingly, bulk performance is not as important, and the performance of this modules is good enough for moving the bottleneck to a different part of the system (normally it is in the I/O communication of the commodity hardware).

Of course, the project team found also some disadvantages in the language; the most important ones are:

- Lack of type system: a big set of the errors found during the system development could be avoid (or its solution could be simplified) with the help of a type system for the language.

- Lack of module system: in a real system, with a quite big set of modules, a good module system makes easier the maintainment of the project evolution. Some proposals have been made in order to solve this problem in the language, and we found that they would be a really good contribution to the improvement of the Erlang/OTP platform.

- Small coverage: despite of the exponential grow of the Erlang user and developer community, it seems clear that the language is still not one of the best known ones, and this makes the task of finding good programmers harder.

- OO-world adaptation to Erlang nature: there is a quite broad work in the definition of design patterns in the object oriented programming. When designing the system, several tools (e.g.: UML) and concepts (e.g.: design patterns, composition, inheritance) from the object oriented world were used. Although most of the times translating these concepts into a process oriented approach is possible, this task is not always trivial. An example of a object oriented concept that is not easy to translate to Erlang is object inheritance.

5. CONCLUSIONS AND FUTURE WORK

VoDKA is a complete Video-on-Demand server with a complex distributed storage architecture. The different stages in the architecture design, going from the initial design to the flexible and adaptable architecture proposed for the next release of the system, have been shown in this paper.

The use of design patterns, implementation patterns, software components, a language with a high level of abstraction (Erlang) and a powerful technology framework (OTP), have made possible this evolution towards an optimal architecture.

We have defined a new architecture, specially based in several design patterns, that makes easier the system configuration and the constraint management. The `trader behaviour` (both in the simple version and in the one including constraint management), and `scheduler behaviour` now are the core part of the system implementation, which makes clearer the implementation of the system internal communication protocol.

In the near future, work in the project should be made in the following topics: End the prototype implementation that is now under development; integration of the prototype with the real working system; and apply formal methods tools [2, 1] in order to know more about the system and to do small redesigns in the architecture to solve possible errors.

6. REFERENCES

[1] T. Arts and C. Benac Earle. Verifying Erlang code: a resource locker case-study. In *Int. Symposium on Formal Methods Europe*, volume 2391 of *LNCS*, pages 183–202. Springer-Verlag, July 2002.

[2] T. Arts and J. J. Sánchez. Global scheduler properties derived from local restrictions. In *Proceedings of the ACM Sigplan Erlang Workshop at the Principles, Logics, and Implementations of high-level programming languages.* ACM, October 2002.

[3] M. Barreiro and V. M. Gulías. *Cluster setup and its administration. In Rajkumar Buyya, editor, High Performance Cluster Computing, volume I.* Prentice Hall, 1999.

[4] M. Barreiro, V. M. Gulías, J. L. Freire, J. Mosquera, and J. J. Sánchez. An erlang-based hierarchical distributed vod system. In *Proceedings of Seventh International Erlang/OTP User Conference.* Ericsson Utvecklings AB, September 2001.

[5] D. Baumer and D. Riehle. Product trader. In *In Pattern Languages of Program Design 3 (PLoPD3)*, pages 29–46. Addison-Wesley, 1998.

[6] U. Ekstrom. Design patterns for simulation in erlang/otp. In *Master Thesis, University of Upsala*, 2000.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison Wesley, Reading, 1996.

[8] V. M. Gulías, C. Abalde, and J. J. Sánchez. Lambda goes to hollywood. In *Fifth International Simposium on Practical Aspects of Declarative Languages (PADL'03)*, volume 2562 of *LNCS*. Springer-Verlang, January 2003.

[9] A. Valderruten, V. M. Gulías, J. J. Sánchez, J. L. Freire, and J. Mosquera. Implementación de un modelo de monitorización para un servidor de vídeo bajo demanda en erlang. In *Proceedings of XXVII Conferencia Latinoamericana de Informtica.* Jonás Montilva, September 2001.