JFP **15** (3): 403–430, 2005. © 2005 Cambridge University Press
doi:10.1017/S0956796805005502 Printed in the United Kingdom

*VoDKA: Developing a Video-on-Demand Server using Distributed Functional Programming**

VICTOR M. GULIAS, MIGUEL BARREIRO
and JOSE L. FREIRE

MADS Group - LFCIA, Department of Computer Science, University of Corunna, Spain
(*e-mail:* {gulias,enano,freire}@lfcia.org)

Abstract

In this paper, we present some experience of using the concurrent functional language Erlang to implement a distributed video-on-demand server. For performance reasons, the server is deployed in a cheap cluster made from off-the-shelf components. The demanding system requirements, in addition to the complex and ever-changing domain, suggested a highly flexible and scalable architecture as well as a quite sophisticated control software. Functional programming played a key role in the development, allowing us to identify functional abstractions throughout the system. Using these building blocks, large configurations can be defined using functional and process composition, reducing the effort spent on adapting the system to the frequent changes in requirements. The server evolved from a prototype that was the result of a project supported by a regional cable company, and it is currently being used to provide services for real-world users. Despite our initial concerns, efficiency has not been a major issue.

1 Introduction

As high speed networks and processors have become commodity hardware, affordable and reasonably efficient clusters are flourishing everywhere. Also, while still expensive, more traditional clustered systems are steadily getting somewhat cheaper. The result is that clusters are no longer very specific, very restricted access systems with completely unique requirements. However, the programming of such systems is still a difficult task. The combination of imperative languages (C, C++, Fortran,...) with message-passing libraries (PVM, MPI,...) originates distributed applications which are both difficult to understand and to reason about. Lack of abstraction, explicit handling of data, dangerous side-effect computing, or explicit memory management are some of the sources of program errors, quite difficult to debug.

On the other hand, the combined use of design patterns and distributed functional programming has been pointed out as a key factor to quickly produce correct

* Partially supported by MCyT TIC2002-02859 and Xunta de Galicia PGIDT02TIC00101CT

distributed systems running on a cluster of computers, with a high degree of adaptability, fault tolerance and scalability. With these ideas in mind, a large successful system has been developed: VoDKA (*Video-on-Demand Kernel Architecture*, `http://vodka.lfcia.org`), an extremely scalable clustered video-on-demand server providing cost-efficient storage, location and transport of media objects. In this paper, we introduce some of our experience in designing and implementing such a system. Its main novelty is that it has been developed almost entirely using a declarative language, the concurrent functional language Erlang (Armstrong *et al*., 1996).

The VoDKA server has evolved from a prototype that was the result of a project supported by the Galician regional cable company in an attempt to provide streaming services to about one hundred thousand potential users. The scope and requirements for such a service were not clearly specified at the beginning of the project. The demanding system requirements, in addition to the complex and ever-changing domain, suggested a highly flexible and scalable architecture: a distributed control software based on components and deployed in a cheap cluster made from off-the-shelf elements. In the project, functional programming played a key role in the development, allowing us to identify recurrent functional abstractions or patterns in the distributed system. After defining these building blocks, large configurations can be conceived using both functional and concurrent composition, thus reducing the effort spent on adapting the system to the frequent changes of requirements, hardware details, network topology, streaming protocols, scheduling algorithms, etc. Moreover, despite our initial concerns, the resulting system is efficient enough to successfully address this real-world problem.

This paper is structured as follows: First, a general overview of VoDKA is shown, introducing the main aspects of the video-on-demand domain as well as the design decisions undertaken in the project. Section 3 briefly introduces the functional language Erlang, its strengths and weaknesses, focusing on its support for concurrent and distributed programming. Section 4 presents some abstractions or patterns that have simplified issues such as device heterogeneity or multi-protocol data movement. After some performance remarks, we finally conclude.

2 The VoDKA project

In 2000, a project partially supported by the cable operator *R Cable y Comunicaciones de Galicia S.A*. was started to provide video-on-demand services to its clients, mainly broadband-quality contents. This company had been studying different options but they realized that most of them were expensive, closed, non-scalable and non-adaptable solutions. Thus, the proposed goal of the project was to build an extremely scalable, fault tolerant, multiprotocol, adaptable (both to the network topology and to end-user protocols) streaming server: The VoDKA server.

The industrialization of the early prototype has motivated the creation of a spin-off company (*LambdaStream*, `http://www.lambdastream.com`) devoted to continuing the development. Currently, VoDKA deployments are in use in many locations such as cable operators, railroad stations, bank offices, and so on.

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

2.1 A glimpse of video-on-demand servers

A *Video-on-Demand* server is a system that provides video services to several clients simultaneously. A user can request a particular video at any time, with no pre-established temporal constraints. Video streaming is a particular case of media streaming and both terms are used indifferently in our presentation. A video-on-demand server must satisfy some critical requirements, including:

- *Large storage capacity*: large number of on-line or near-line videos; each video may be large (hundreds of megabytes to tens of gigabytes).
- *Many concurrent users*: thousands of concurrent sessions while attending to new requests. Quality of service is relevant: It is admissible to reject a new request if the system cannot guarantee the required resources; however, once a streaming session starts, it should continue until completion.
- *High bandwidth*: high aggregate bandwidth to serve all the concurrent users receiving high bitrate media. There are many potential bottlenecks in such a system: network, disks, CPU, etc.
- *Reliability*: considering video-on-demand service time, reliability is a must (think, for instance, of a two-hour movie show). That forces us to ship error-free code, to recover gracefully from hardware or software errors, and to provide mechanisms to incorporate new features without stopping the system.
- *Scalability*: both upwards and also downwards scalability. A simple configuration can be enough to attend only a few clients, but it should be possible to increase system resources as soon as new potential users arise.
- *Adaptability*: the system should adapt to the underlying topology, making efficient use of the available network resources.
- *Low cost*: our goal is to reduce the costs involved in the whole system – hardware, software and network usage.

In recent years many companies have been developing video-on-demand related solutions. Some of them are well suited for low-bandwidth streaming (possibly over the *Internet*), like the popular RealNetworks RealVideo Server (now Helix), Apple Darwin Streaming Server (Apple Computer Inc., 2004), or the proprietary Microsoft Windows Media Server (Microsoft, 2004). Other solutions are more focused to high bandwidth LAN streaming, like IBM DB2 Digital Library Video Charger (Wilkinson *et al*., 1999; IBM, 2004) or Kasenna MediaBase. Other systems are optimized for the digital TV VoD environment, like MidStream MVS, SeaChange MediaCluster, Concurrent MediaHawk, or nCube n4x.

2.2 The design of VoDKA

Clusters built from cheap off-the-shelf components (Barreiro & Gulias, 1999) represent an affordable solution for a large variety of applications that demand huge amounts of resources. Linux is our preferred operating system option due to its excellent performance and the possibility of low-level tuning of important performance parameters. However, the main problem is how to design and implement the distributed application to control the cluster, achieving the demanding requirements

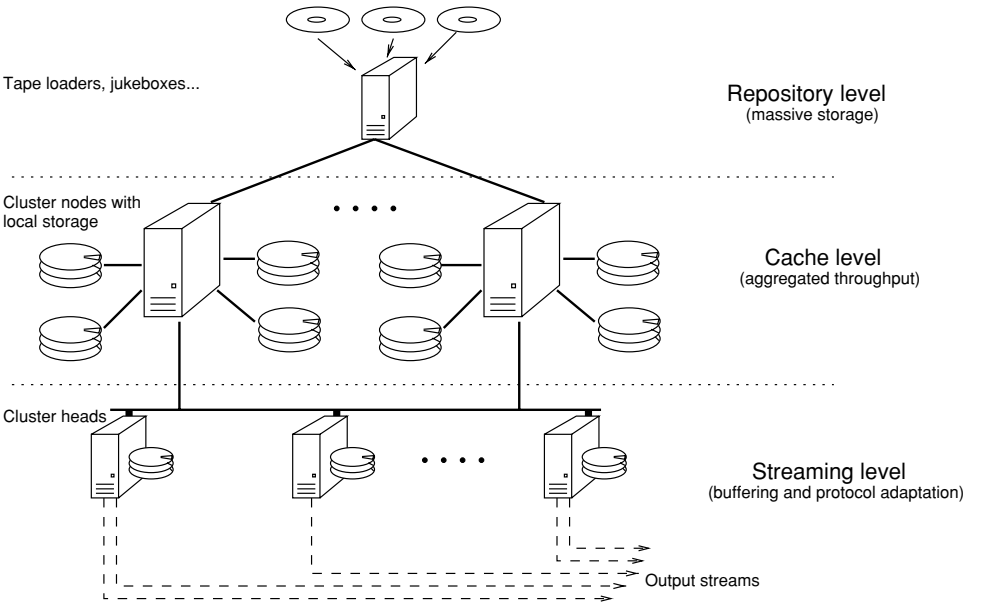

Fig. 1. The initial hierarchical structure.

of a video-on-demand system. It should be as flexible as possible to quickly adapt to the ever-changing domain (target clients, communication protocols, types of storage, ...). Moreover, developers are scarce resources in the project, thus we have to be very productive to succeed —a good challenge for functional programming.

2.2.1 *Initial approach: storage hierarchy*

As first approach, cluster resources are organized as a hierarchical storage system as proposed in Chan & Tobagi (1997). Figure 1 shows three levels in the hierarchy:

- *Repository level*: It stores all the available media using different technologies: tape loaders, DVD jukeboxes, disk arrays, etc. It has *large capacity* which usually means high latency and low throughput to be cost-effective.
- *Cache level*: Cluster nodes in charge of storing videos read from the tertiary level, before being streamed. It provides a large aggregate throughput that alleviates the usual deficiencies of the repository level.
- *Streaming level*: Collection of nodes in charge of protocol adaptation and media streaming to the client using the appropriate format. It isolates several details, such as buffering or bitrate variability, from the client.

2.2.2 *VoDKA web of traders: chain of responsibility approach*

For practical reasons, the original three-level approach results inadequate: it is unnecessary for small installations while even more flexibility is required to cope with complex topologies. Thus, the hierarchical approach was replaced by a more general

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

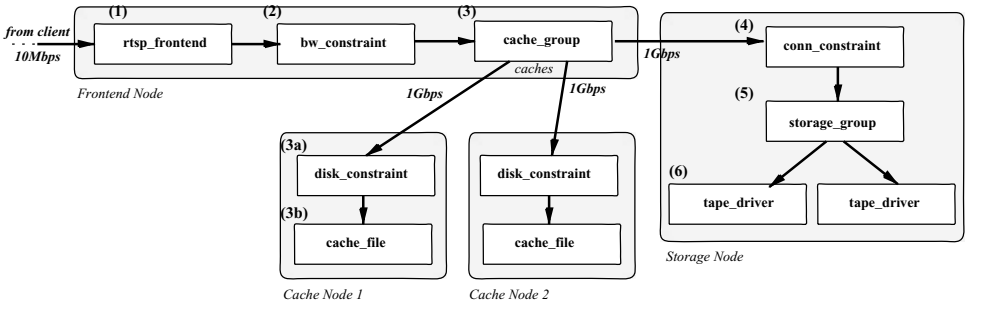

Fig. 2. Sample deployment of a VoDKA configuration.

architecture composed of *traders*. Traders are software processes (*Erlang servers*, as presented later) responsible of creating a path from one of the video sources to the client. To make this possible, traders cooperate using a communication chain that connects them, defining a particular *configuration*. This interaction resembles the well-known design pattern *chain of responsibility* (Gamma *et al*., 1995). Each trader can (potentially) be deployed in a different physical node. The sample shown in Figure 2 presents the interactions when a client request is submitted.

1. The request is received by a *frontend trader* that interacts with the client using a particular streaming protocol. In this case, for example, the `rtsp_frontend` defines an RTSP adaptation. The frontend requests the media to the next trader in the chain of responsibility using a uniform internal protocol.
2. The request is received by a *bandwidth constraint trader* (`bw_constraint`) to model the network limitation of 10Mbps with clients. If there is enough available bandwidth, the request is delegated to the next trader in the chain; otherwise, it is rejected.
   In general, a constraint trader manages only one particular resource (disk concurrent accesses, network bandwidth, number of connections, CPU, etc.)
3. The request is received by a *distributed cache controller* (`cache_group`). This trader submits the request to all the available cache chains in parallel. For each cache chain, the interaction is the following:
   (a) The request is received by a *disk constraint trader* (`disk_constraint`). It models the specific disk behaviour for the node. The request is delegated to the next trader if there is enough disk bandwidth.
   (b) The request is received by a *local cache controller* (`cache_file`) which handles a disk cache; the request can be attended if media is available.

   If some of the cache traders can solve the request (*cache hit*), the controller chooses the best option according to a particular cost-based policy. Otherwise (*cache miss*), the request is propagated to the next trader in the chain. This will load the media in the most suitable cache from a cost point of view.
4. The request is received by a *connection constraint trader* (`conn_constraint`). It limits the number of concurrent streaming sessions.

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

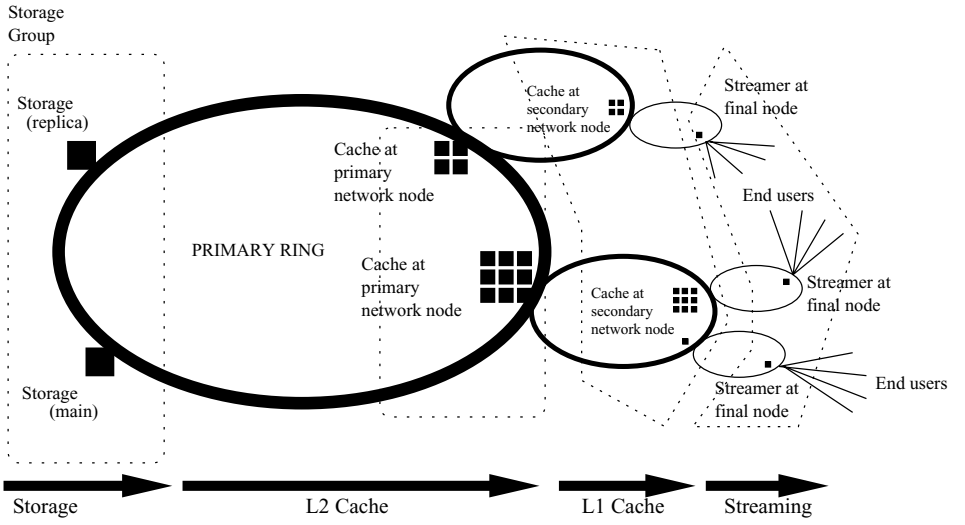

Fig. 3. Configuration of the server on a complex network topology.

5. The request is received by a *storage composite* (`storage_group`), which groups trader chains. Following a given policy, the request is propagated through different chains (all in parallel, sequentially with a given priority, etc.)
6. The request is received by a *storage controller*, in this case a tape controller (`tape_driver`). If media is available, the request can be successfully attended.

When a path is established, a group of processes are created at each node to move the media from the source (storage or local cache) to its destination (local cache or client). This is commented in section 4.4 when introducing the *pipe* abstraction.

Using this compositional approach, simpler configurations can be defined by removing components such as cache-related or unnecesary constraint traders. Moreover, complex configurations can be designed to adapt the service to particularly difficult underlying topologies. For example, Figure 3 outlines the usual setup for a cable network provider. In this case, storage is distributed (mirrored) at two different points and a two-level cache is deployed throughout the provider network. Hence, media can be moved closer to end users, optimizing network usage and improving availability.

3 The development platform: Erlang/OTP

After choosing commodity hardware clusters for the underlying hardware architecture, *concurrent functional programming* was selected for implementing the design presented in section 2. The identification of patterns that appear recurrently seemed to be a key factor in developing distributed applications and, not surprisingly, the concurrent functional paradigm offers a good framework to abstract these building blocks out.

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

3.1 Why Erlang?

In the past, we explored the connection between concurrency/distribution and functional programming using languages such as ML or Haskell (Gulias, 1999). However, we chose Erlang (Armstrong *et al*., 1996), a distributed and concurrent functional programming language developed by Ericsson AB for telecommunication applications. Erlang is a notable successful exception of a functional language applied to real-world problems (Wadler, 1998). There were many reasons that supported this decision:

- *Nature*: The target system is inherently concurrent and Erlang has been designed with concurrency and distribution in mind.
- *Experience*: There are successful examples of large distributed Erlang applications in 24×7 conditions.
- *Libraries*: Erlang offers a powerful collection of libraries, the *Open Telecom Platform* (OTP), that eases the development of distributed applications. OTP includes a distributed database, graphical interfaces, an ASN.1 compiler, a CORBA object request broker, COM and Java interfaces, among others.
- *Interface*: Erlang has a clean interface with low-level languages (C, for example), crucial for dealing with devices or for performance critical modules.
- *Efficiency*: Erlang is surprisingly efficient, in particular when it handles a large number of concurrent processes. Even input/output libraries are fast enough for our purposes, in many cases. If necessary, it also has a native-code compiler, *Hipe* (Johansson *et al*., 2000).
- *Soft realtime behaviour*: Erlang runtime system has been designed to implement soft realtime applications. In particular, each process has a separate heap, the garbage collector does not block all the processes, and context-switching is cheap (especially if most of the processes are waiting for I/O). This is important in order to maintain low response time and meet timing requirements of streaming, even when the workload is high.

There were, however, three major inconveniences with Erlang that became more obvious during the project:

- *No static type checker*: Type errors are not immediately caught at compile time. This is a problem when changing a type definition or when using higher-order communications (sending/receiving functional values). Just an optional simple type checker could have saved hours of debugging. Some approaches have been tempted in the past (Marlow & Wadler, 1997).
  To reduce this problem, a small runtime type checker was included as part of our servers to check the parameters of each service. Each server stores metainformation about (a) services available, and (b) type templates for each service parameter. Before dispatching a request, the server dynamically checks the type of a parameter against the expected type.
- *Module system*: This is twofold. First, related with the type system issue, an interface/implementation module definition is desirable: it is not possible to check if a module satisfies a required interface. Second, modules have a

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

primitive flat namespace. Fortunately, a hierarchical module system has been included in the latest releases of Erlang (Carlsson, 2003).

- *Coverage*: As with most of functional languages, we need more programmers, more books, more patterns, more experience...

3.2 Erlang as a sequential functional language

Erlang has no construct inducing side-effects with the exception of communications among processes. It evaluates expressions eagerly as other strict functional languages such as ML.

Values in Erlang (i.e. non-reducible expressions) range from numbers and *atoms* (symbolic constants, lower-case in Erlang syntax) to complex data structures (lists, tuples) and functional values, which are treated as first-class citizens. A function is defined by a set of equations, each stating a different set of constraints based primarily on the structure of the arguments (*pattern-matching*). Identifiers bound during pattern matching, *variables*, are upper-case in Erlang syntax. Iterative control flow is carried out by using function recursion. Lists are written `[a,b,c]` with `[]` being the empty list, and a list whose first element is `X` and whose rest is `Xs` is denoted `[X|Xs]`. Thus, a function that tests for membership in a list is defined as:

```
member(X,[])     -> false;
member(X,[X|Xs]) -> true;
member(X,[Y|Ys]) -> member(X,Ys).
```

Functions are grouped into modules, and a subset of those functions can be exported, declaring both function name and arity, to be used in other modules.

```
-module(example).
-export([member/2]).
```

Hence, `example:member(3, [1,2,3])` evaluates to the atom `true`, while the expression `example:member(5, [1,2,3])` reduces to `false`. The language lacks a static type system present in other modern functional languages; lists can contain values with different types such as `[a, [], 1]`. Besides lists, Erlang programmers also can use *tuples* which are constructed in arbitrary but finite length by writing {`A,B,...,C`}. Lists and tuples can hold any valid Erlang value, from numbers and atoms to lists, tuples and even functional values. *Records* are also provided as useful syntactic sugar for accessing tuples by name instead of by position. For instance, `P#point.x` denotes the field `x` of value `P`, according to a previously declared record template `point`.

Erlang also offers a *sequential operator*: E_1`,`E_2 which evaluates E_1 (perhaps performing communications or binding variables using pattern matching), discards the computed value and then it computes E_2 with the variables bound in E_1. Thus, an expression X`=`E_1`,`E_2 is similar to ML's `let` x`=`E_1 `in` E_2.

As in other functional languages, many additional features are available such as higher-order functions, list comprehensions (`[`$f(S)$ `||` S `<-` Gen`,` $p(S)$`]`), anonymous function definitions (`fun (`Pat`) ->` $Expr$ `end`), etc.

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

3.3 Support for concurrency and distribution

What makes Erlang different from other functional languages is its support for concurrency and distribution. With Erlang's primitives for concurrency, it resembles formal calculi such as Milner's CCS (Milner *et al.*, 1992) or Hoare's CSP (Hoare, 1985). Because of the absence of side-effects, limited to explicit inter-process communications, concurrent programming is far simpler in a functional than in an imperative language.

A new *process* is created by using the built-in primitive `spawn`. The expression `spawn(`M`,`F`,[`A_1`,`A_2`,...,`A_N`])` starts a new computation thread to evaluate $M : F(A_1, A_2, \ldots, A_n)$, and it returns the *process identifier* (`Pid`) of the newly spawned process.

The final result of a process computation is just discarded. Hence, explicit communications are necessary to define coordination among processes. A couple of asynchronous message passing primitives are available in the language for such purpose:

- *Asynchronous send*:

  ```
  Pid ! Msg
  ```

  `Msg` is sent to process `Pid` without blocking the sending process. If `Pid` exists, the message is stored in `Pid`'s *mailbox*, a sequential collection of incoming messages for `Pid`. Any valid Erlang value can be sent to other process, including complex data structures containing lists, tuples, functions, process identifiers, etc.

- *Mailbox pattern matching*:

  ```
  receive
        Pat1 -> Expr1;
        ...
        PatM -> ExprM
  end
  ```

  The process mailbox is sequentially searched for a message that matches one of the patterns $Pat_1, \ldots, Pat_M$. If no such message is found, the process blocks until received. If a message matches Pat_i, the evaluation of the whole `receive` expression will be the evaluation of $Expr_i$ with the bindings produced by matching Pat_i.

An Erlang virtual machine (*node*, in Erlang terminology) hosts several Erlang processes running concurrently. Usually, an Erlang node is mapped to an operating system process; an Erlang process is, in fact, a lightweight user-level thread with very little creation and context-switching overheads.

A distributed Erlang application consists of processes running in several nodes, possibly at different computers. Even though the initialization and management of each node is platform dependant, the nice feature is that communication among remote processes is semantically equivalent in a distributed framework – though remote communications are less efficient, of course. It is possible to explicitly create

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

a process on a remote node *Node*, by using `spawn`(*Node*, *M*, *F*, *Args*), while `spawn/3` just creates a process in the current node.

4 The role of functional programming in VoDKA

The use of a functional language, in particular the identification of *functional patterns*, has been a key factor to simplify the development. Like classical abstractions such as `map` or `foldr`, the identified patterns use higher-order functions to abstract particularities as functional parameters. Considering that an abstraction is specialized with different but related functions, two approaches have been used:

- To define a data structure that collects the required functional parameters.
- To use a module, a container of functions, as a parameter.

As samples of abstractions defined to solve recurrent problems found in VoDKA, the following sections will show some of the patterns used.

4.1 The basic server pattern

A *server* is an abstraction found in almost every concurrent and distributed application. A server behaves iteratively in the following way: (a) it receives a request from a client; (b) a response is computed based upon both the request information and the internal state of the server; (c) the response is sent back to the client; and (d) server state is updated for subsequent iterations. A simple model of this behaviour is a tail-recursive definition that supplies explicitly the server state as a parameter. For instance, the following code defines the behaviour of a memory allocator server.

```
-module(allocator).
-export([loop/1]).

loop(HeapPointer) ->
    receive
      {request, From, {alloc, N}} ->
            From ! {reply, HeapPointer},
            loop(HeapPointer+N)
    end.
```

Here, a request is represented as a 3-tuple {`request`, *From*, {`alloc`, *N*}}, where `request` is the atom that identifies a client request, *From* is the identity of the client process that should receive the response back, and {`alloc`, *N*} is the actual request. The expression `spawn(allocator, loop, [16])` creates a server with initial state `HeapPointer=16`. The client API is defined as:

```
call(Server, Request) ->
   Server ! {request, self(), Request},
   receive
     {reply, Reply} -> Reply
   end.
```

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

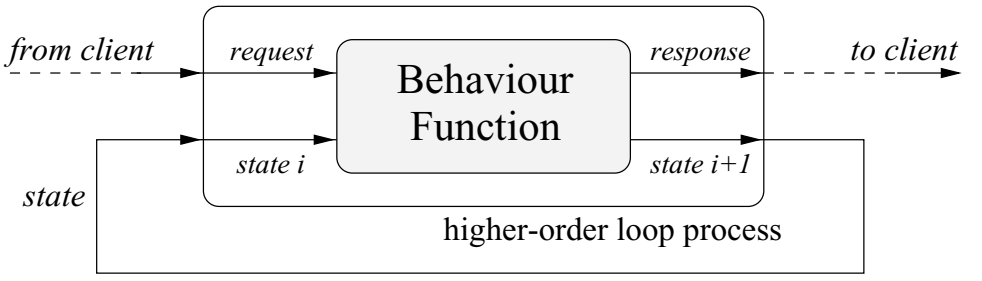

Fig. 4. Server behaviour.

where `Server` is the identity of the server process, and `self/0` is a built-in primitive that returns the current process Pid. If *PidS* is bound to the server Pid whose state is `100`, then `call(`*PidS*`, {alloc, 10})` returns `100`, changing server state to `110`.

Higher-order functions can be used to generalize the basic server pattern, avoiding the repetition of the same structure at different places. As shown in Figure 4, a function $Behaviour : Request \times State \rightarrow Response \times State$ is used to model the computation of the response and also the state for the next iteration from the request and the current state. While the behaviour is a side-effect-free function, the higher-order recursive definition, `loop`, deals with client-server interaction. As seen, request resolution and server state changes are inherently serialized.

```
-module(server).
-export([start/2, loop/2]).

start(Behaviour, State) ->
  spawn(server, loop, [Behaviour, State]).

loop(Behaviour, State) ->
  receive
   {request, From, Request} ->
       {Response, NewState} = Behaviour(Request, State),
       From ! {reply, Response},
       loop(Behaviour, NewState)
  end.
```

The abstraction can be specialized to build the original memory allocator server. In this case, an anonymous function is used to model the behaviour:

```
Allocator = server:start(fun ({alloc, N}, HeapPointer) ->
                                 {HeapPointer, HeapPointer+N}
                         end, 16)
```

4.2 Guarded suspension pattern

Sometimes, the simple approach of the basic server pattern is not enough for modeling more complex situations. Occasionally, a request must be suspended until

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

the server reaches a given state or leaves an exceptional state. The *guarded suspension pattern* (Grand, 1999; Lea, 1997) deals with the suspension of a synchronized service call until a precondition is satisfied.

Consider the implementation of a global queue. This queue can be the glue in a producer-consumer interaction. In VoDKA, for example, this structure is used for gathering packets coming from a cache node before being processed by a frontend streamer.

The queue is modeled using a server process with two services: `push` and `pull`. The state of the server is an immutable data structure implementing a queue using a pair of lists (Okasaki, 1996). Using the server pattern presented in section 4.1, the queue implementation looks like:

```
-module(queue).

-export([new/0, push/2, pull/1, queue/2]).

new() -> server:start(fun queue/2, {[],[]}).

push(Queue, X) -> server:call(Queue, {push, X}).
pull(Queue)    -> server:call(Queue, pull).

queue(pull, {[H | Hs], Tail})  ->  {H, {Hs, Tail}};
queue(pull, {[], Tail})        ->  queue(pull, {reverse(Tail), []});
queue({push, X}, {Head, Tail}) ->  {ok, {Head, [X | Tail]}}.
```

To use the abstraction, a queue is instantiated using the `new` function, as shown:

```
MyQueue = queue:new(),
...
queue:push(MyQueue, 5)
```

The serialization of both services (`push` and `pull`), inherent to the generic server definition, guarantees correct concurrent access to the queue. However, there is a possible deadlock situation when a client demands a `pull` service with an empty queue: the server gets locked because of an infinite loop when both `Head` and `Tail` lists are empty. No more requests are going to be attended, even if there are processes trying to push values into the queue. The guard condition can be considered as a special case, but the definition of the basic server forces us to compute the response *before* attending to further requests.

```
queue(pull, {[], []}) ->    %%% MUST SUSPEND REQUEST!!!!!!!
        ;
queue(...) -> ...
```

As a solution, the server pattern is extended to support guarded suspensions. Now, a request can be temporarily suspended until a precondition holds. Figure 5 shows a producer-consumer scenario showing the desired behaviour.

This pattern applies when (a) the services of a server must synchronize to access a critical section one process at a time because they update the server state, and (b)

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

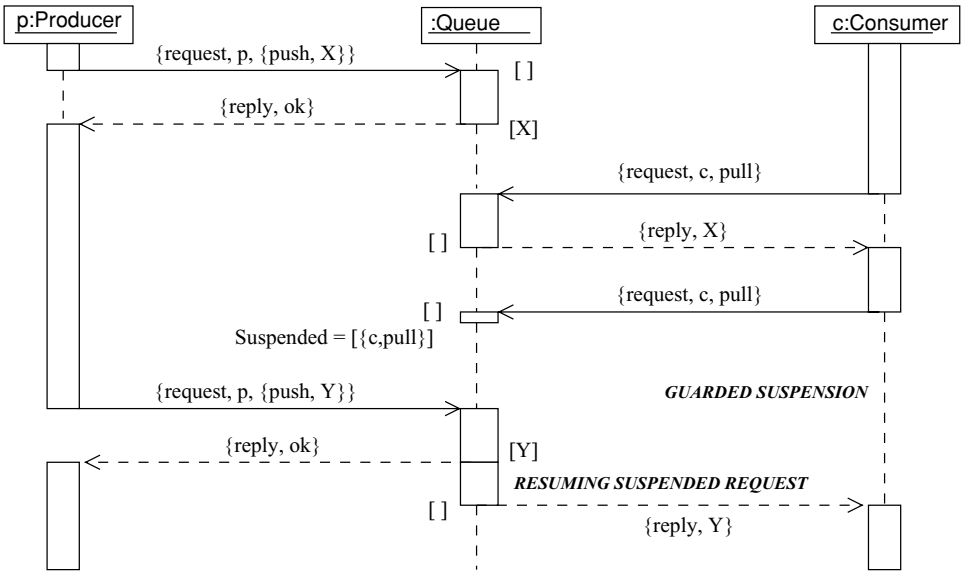

Fig. 5. A producer-consumer scenario using a global queue.

the actual state of the server makes it impossible for one of its services to execute to completion, and a call to one of the other synchronized services is the only mean to leave that state.

An extension of the basic server pattern defines as state all the suspended requests (`Suspended`) and, in addition to the *Behaviour* function, it includes two extra functional parameters:

- $Guard : State \times Req \rightarrow Boolean$, that checks whether a server state is valid for attending to the request, and
- $Resume : State \times [\{Client, Req\}] \rightarrow \texttt{none} \mid \{\texttt{ok}, \{Client, Req\}, [\{Client, Req\}]\}$, which checks whether one of the suspended request can be resumed.

The implementation of this new abstraction is:

```
-module(guardeds).
-export([start/4, loop/5]).

start(Behaviour, Guard, Resume, InitialState) ->
  spawn(guardeds, loop, [Behaviour, Guard, Resume, InitialState, []]).


loop(Behaviour, Guard, Resume, State, Suspended) ->
  receive
    {request, Client, Req} ->
      {NewState, NewSuspended} = dispatch(Behaviour, Guard, Resume,
                                          State, Suspended, Client, Req),
      loop(Behaviour, Guard, Resume, NewState, NewSuspended)
  end.
```

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

```
dispatch(Behaviour, Guard, Resume, State, Suspended, Client, Req) ->
  case Guard(State, Req) of
     true ->
       {Reply, NewState} = Behaviour(Req, State),
       Client ! {reply, Reply},
       case Resume(NewState, Suspended) of
          none -> {NewState, Suspended};
          {ok, {AnotherClient, AnotherReq}, NewSuspended} ->
               dispatch(Behaviour, Guard, Resume,
                        NewState, NewSuspended, AnotherClient, AnotherReq)
       end;
    false ->
       {State, [{Client, Req} | Suspended]}
  end.
```

With this abstraction, a queue with guarded suspension is implemented as:

```
-module(queue).
-export([new/0, push/2, pull/1, queue/2]).

new() ->
  guardeds:start(fun queue/2, fun pull_on_non_empty/2,
                 fun resume/2, {[],[]}).

%% GUARD is true when pulling from a non empty Queue
pull_on_non_empty(pull, {[],[]}) -> false;
pull_on_non_empty(_,_)           -> true.

resume({_, []},  _)       ->  none;
resume(_, [ {Client, Request} | MoreSuspended]) ->
                 {ok, {Client, Request}, MoreSuspended}.
```

4.3 Resource scheduler pattern

In the guarded suspension pattern, if multiple threads are suspended, the pattern does not establish beforehand the order in which each process is resumed. In the particular implementation presented, the last request suspended is the first resumed. The *scheduler pattern* (Lea, 1997) is proposed to deal with exclusive access to a resource shared among several processes. Each process must wait until the resource is available and assigned to it following a specific policy. For example, this pattern is found in VoDKA code that controls the loading of a tape into a tape drive.

The pattern is applied when (a) different calls must be synchronized to access a shared resource by one process exclusively, and (b) there is a particular policy on how to sequence accesses when several processes wait to seize the resource.

Figure 6 summarizes typical interactions. When a process needs a resource, it requests access invoking the `enter` method of a *scheduler*. This call will not return until the scheduler decides to grant access to the process. The process will hold the resource exclusively until it invokes scheduler's `done` service; at this moment the scheduler will choose among all the waiting processes.

The scheduler is implemented as a guarded suspension server with two services (`enter` and `done`). It is configured with a function $Policy : [\{Client, Req\}] \longrightarrow$

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

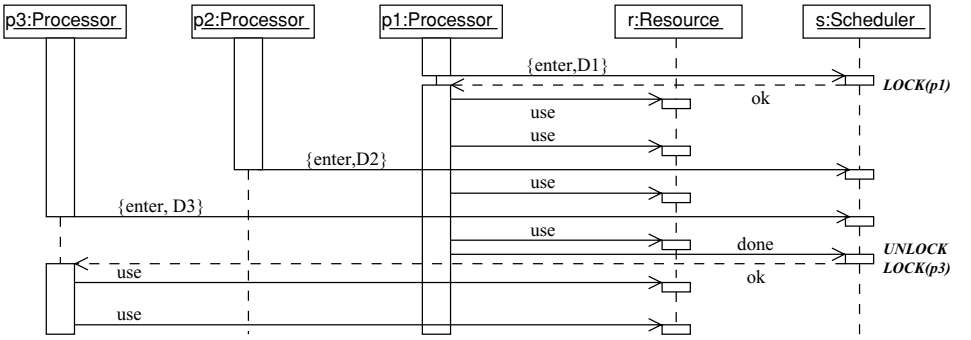

Fig. 6. Using a scheduler to access a shared resource.

$\{\{Client, Req\}, [\{Client, Req\}]\}$ for selecting the next process to be resumed. In addition to the policy, the state of the scheduler includes information about lock status (`busy` or `available`).

```
-module(scheduler).
-export([new/1, enter/2, done/1]).
-export([handle/2, must_suspend/2, resume/2]).

new(Policy) -> guardeds:start(fun handle/2, fun can_access/2,
                              fun resume/2, {Policy, available}).

enter(Scheduler, Data) -> server:call(Scheduler, {enter, Data}).
done(Scheduler)        -> server:call(Scheduler, done).

handle({enter, Data}, {Policy, available}) -> {ok, {Policy,busy}};
handle(done, {Policy, busy})               -> {ok, {Policy,available}}.

can_access({enter, Data}, {Policy, busy}) ->  false;
can_access(_,_)                           ->  true.

resume({Policy, available}, Suspended) ->
    {Request, MoreSuspended} = Policy(Suspended),
    {ok, Request, MoreSuspended};
resume(_, _) -> none.
```

Some policies can be defined as follows:

```
-module(policy).
-export([fifo/1, lifo/1, priority/1]).

  lifo([X|Xs])    -> {X,Xs}.
  fifo(Suspended) -> {last(Suspended), firsts(Suspended)}.
```

Or if *Data*, an integer, is the priority:

```
priority([ASuspended | MoreSuspended]) ->
   lists:foldl(fun (Req, {MaxReq, RestRequests}) ->
                  case higher_prio(Req, MaxReq) of
```

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

```
                    true  -> {Req, [MaxReq|RestRequests]};
                    false -> {MaxReq, [Req|RestRequests]}
                end,
            end, {ASuspended, []}, MoreSuspended).

higher_prio({_, {_, D1}}, {_, {_, D2}}) -> D1 > D2.
```

Once a scheduler is defined, a convenient API to access the resource can be defined as follows:

```
-module(myresource).
-export([start/1, doit/1]).

start(Policy) ->
    Scheduler = scheduler:start(fun (X) -> policy:fifo(X) end),
    ResourceID = ... % some resource initialization
   {Scheduler, ResourceID}.

doit({Scheduler, ResourceID}) ->
    scheduler:enter(Scheduler),

          ... work on ResourceID ...

    scheduler:done(Scheduler).
```

4.4 Data movement pattern

One of the most recurring tasks in VoDKA is media movement. For instance, videos are moved from storage to cache (to take advantage of the cache aggregate bandwidth) or from cache to a frontend node (as being sent to the user). Heterogeneity is an important problem as sources and destinations can be rather different: file systems, TCP connections, UDP flows, direct memory transfers, etc. Moreover, the movement of data involves some accounting that is, in essence, independent of the nature of sources and destinations. An abstraction, a *pipe*, is introduced to unify data movement from a *data source* to a *data destination*. Source and destination are both determined by a module and its particular initialization data.

The data source module exports, among others, the following functions:

- `init(DS_initparam) -> {ok, DS_info, DS_state} | {error, Reason}`

  It initializes the source with some initialization parameter. It returns a tuple with the atom `ok`, information about the data source such as name, size, MIME type, and so on (`DS_info` record), and the explicit state to read from the source (`DS_state`). If something goes wrong, it delivers an `error`.

- `read(DS_State) -> {ok, Data, DS_state} | {done, DS_doneparam}`
  `| {error, Reason}`

  It reads the next chunk of data from the data source, returning `ok`, the actual data read (`Data`, a binary value), and the state for the next iteration. If all

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

data in the source has been consumed, it delivers `done` and the state required for closing the data source; if something goes wrong, it delivers an `error`.

- `done(DS_doneparam) -> ok | {error, Reason}`

  It cleans up the data source after all its data has been consumed.

Main functionality of the destination module is:

- `init(DS_info, DD_initparam) -> {ok, DD_state} | {error, Reason}`

  It initializes the destination with some initialization parameter and information about the data source initialization. It returns a tuple with the atom `ok`, and the explicit state to write to the destination (`DD_state`). If something goes wrong, it delivers an `error`.

- `write(Data, DD_State) -> {ok, DD_state} | {error, Reason}`

  It writes the next chunk of data to the data destination, returning `ok` and the state for the next iteration. If something goes wrong, it returns an `error`.

- `done(DD_state) -> ok | {error, Reason}`

  It closes the data destination.

Now, we present a simplified implementation of the pipe abstraction used in VoDKA; actual pipes perform additional duties such as error handling, transmission rate control, logging, etc. A new pipe is created using `start/2`, which spawns a new process to perform the data movement. The transfer, modeled with `transfer/2`, initializes data source and destination and then it moves all the packets between them (`pipe_while/4`). After that, both data source and destination are closed.

```
-module(pipe).
-export([start/2, transfer/2]).

start(DS, DD) -> spawn(pipe, transfer, [DS, DD]).

transfer({DS_module,DS_initparam}, {DD_module,DD_initparam}) ->
    {ok, DS_info, DS_st0} = DS_module:init(DS_initparam),
    {ok, DD_st0} = DD_module:init(DS_info, DD_initparam),
    {DS_stf, DD_stf} = pipe_while(DS_module, DS_st0, DD_module, DD_st0),
    ok = DS_module:done(DS_stf),
    ok = DD_module:done(DD_stf).
```

The tail-recursive function `pipe_while/4` reads packets from the source and writes them down to the destination until data source is depleted.

```
pipe_while(DS_module, DS_state, DD_module, DD_state) ->
  case DS_module:read(DS_state) of
     {ok, Data, DS_statenext} ->
```

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

```
            {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.
```

For example, a data source that reads packets from a file is defined:

```
-module(read_file).
-export([init/1, read/1, done/1]).

init(Filename) -> case file:open(Filename, [read, raw, binary]) of
                      {ok, IoDevice} ->
                          PreRead = file:read(IoDevice, 64*1024),
                          {ok, [{name, Filename}], {IoDevice, PreRead}};
                      {error, Reason} ->
                          {error, Reason}
                  end.

read({IoDevice, eof})             -> {done, IoDevice};
read({IoDevice, {error, Reason}}) -> {error, Reason};
read({IoDevice, {ok, Data}})      ->
        {ok, Data, {IoDevice, file:read(IoDevice, 64*1024)}}.

done(IoDevice) ->
    file:close(IoDevice).
```

The module uses the `file` module to open, read, and close a file. Similarly, a data destination that dumps all the packets to a file is defined:

```
-module(write_file).
-export([init/2, write/2, done/1]).

init(_, Filename) -> file:open(Filename, [write, raw, binary]).

write(Data, IoDevice) ->
    case file:write(IoDevice, Data) of
        ok    -> {ok, IoDevice};
        Error -> Error
    end.

done(IoDevice) -> file:close(IoDevice).
```

A file copy is implemented with a pipe that uses `read_file` and `write_file`:

```
pipe:start({read_file, "oldfile.dat"}, {write_file, "newfile.dat"})
```

Figure 7 shows the chaining of pipes for a streaming session. In this case, a particular media is being moved from a tape (`tape_read`) to a cache (`cache_write`) and, simultaneously, from cache (`cache_read`) to the final client using a specific streaming protocol (`rtsp_send`). Inter-node transferences are done using TCP sockets (`tcp_send` and `tcp_recv`). Both the chain and inter-node communication protocols are configured as the result of the trader negotiation presented in section 2.2.2.

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

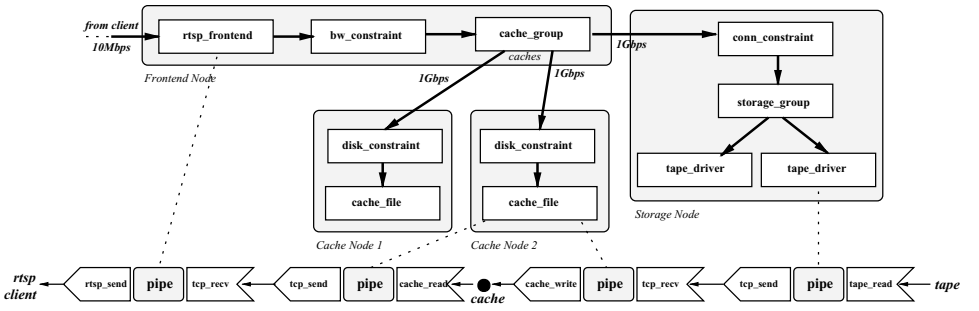

Fig. 7. Chaining of pipes in a transmision.

4.5 Process composition pattern

In VoDKA, videos are stored in several storage devices (hard disks, DVD, CD, other servers, etc.). Each type of storage has its own specific controller. Clients send their requests to the appropriate trader process to get the required video. To simplify this search, a special storage trader, a *storage group*, is defined as a composition of several storages. Hence, a tree of storages can be defined while clients keep the illusion of accessing a simple storage. This homogeneous treatment of leaves and groups is the essence of the *composite pattern* (Gamma *et al.*, 1995).

Declarative programming style is quite valuable when dealing with concurrent behaviour such as the storage composition. For example, a synchronous (sequential) delegation on each of the children of a given group can be implemented using list comprehension. In this case, if `call(Pid,M)` sends the message `M` to the process `Pid` and then it waits for a response, the following behaviour performs a sequential delegation on all the child processes:

```
composite_behaviour(Request, State) ->
    Combine([ call(S, Request) || S <- State#composite_state.children ]).
```

The composite process implements the `Request` service as a sequential delegation of the same service on all its children, and then it combines the results using the function `Combine([`*Response*`])` → *Response*. Observe that the state of the server, `State`, is represented as a `composite_state` record, with a field named `children`. Using this abstraction, it is possible to define a *lookup* service to find the best location of a media object:

```
sequential_storage_behaviour({lookup, MO}, State) ->
   MinCost = State#group_state.min_cost,
   MinCost([call(S, {lookup, MO}) || S <- State#group_state.children]).
```

In the example, the function `min_cost`, stored as part of the group state, is used as a *functional strategy* to choose the best answer from all the children. This allows to change the cost selection algorithm used by the storage group at runtime.

There are many different possible types of interaction between the composite and its children. Figure 8 shows two sequence diagrams describing the sequential and

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

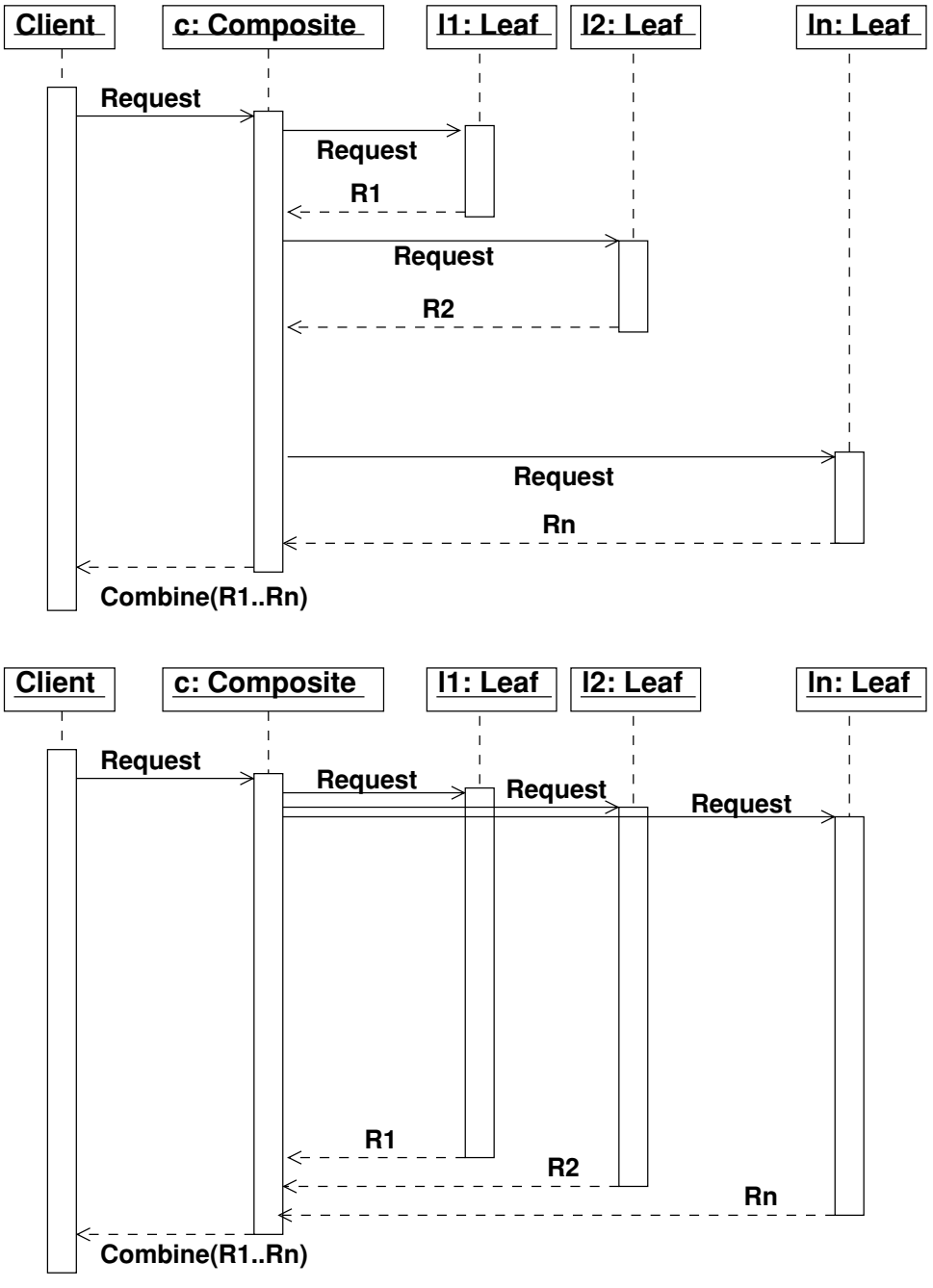

Fig. 8. Sequential and parallel interaction between composite and children.

parallel interactions. In the parallel composition, the composite sends the request to all the children in parallel, and then it receives the responses back. This parallel delegation of a composite process is defined as follows:

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

```
group_behaviour(Request, State) ->
  Combine([ async_recv(Token)
              || Token <- [ async_call(S, Request)
                            || S <- State#composite_state.children]]).
```

In the example, `async_call(Pid, M)` → `Token` sends the message `M` to process `Pid` and returns immediately a token which can be used later to get the response using `async_recv(Token)` → `Response`. All the requests are sent asynchronously and, after that, the responses are received. Finally, all the responses are combined using the `Combine` function to produce the response for the composite.

5 Performance considerations

At the beginning of the project, we decided that the distributed control system would be developed in Erlang, but time-sensitive I/O operations should be carried out by low-level C modules for performance reasons. To speed up development, some Erlang modules carrying out the basic input/output operations were implemented; they would be replaced by native counterparts only if efficiency goals were not achieved: our estimates showed Erlang/OTP input/output performance to be two to three times worse than raw C code.

A quick and correct implementation, very short development time, ease of adaptation to changing requirements and all the commented advantages of the Erlang/OTP platform soon offset the preconceived reasons to reimplement these input/output modules in C.

5.1 Measuring performance in a single computer

In order to evaluate an Erlang based system with heavy I/O, a minimalistic configuration with a streamer (`http_frontend`) and a storage controller (`file_driver`) was defined. Both components were running in different Erlang virtual machines in a single low-end computer (Pentium II/350 384MB 2*4.5GB SCSI). Another computer was connected over 1000Base-T ethernet to the cluster switch to simulate dummy clients performing media requests over a TCP based protocol (*progressive HTTP*). Figure 9 shows measurements performed on the system. Serving 100 concurrent 512 Kbps video streams, the system behaved smoothly with a CPU usage of about 50% and reasonable response times. Requesting 200 concurrent streams of the same bandwidth, 99% of server CPU time was busy, and a few requests were delayed. With 150 concurrent requests, performance was quite similar, the CPU occupation was about 95% and response times were kept within limits tolerable by clients. Trying with different rates, the server was able to attend 100 concurrent requests at 1Mbps, and about 50 concurrent at 2Mbps.

Much to our surprise, we found that while we were not achieving the maximum possible throughput for our hardware, performance for these Erlang modules was more than adequate for production use. In fact, we only need to deploy a very small number of native modules for extremely timing-sensitive operations or to

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

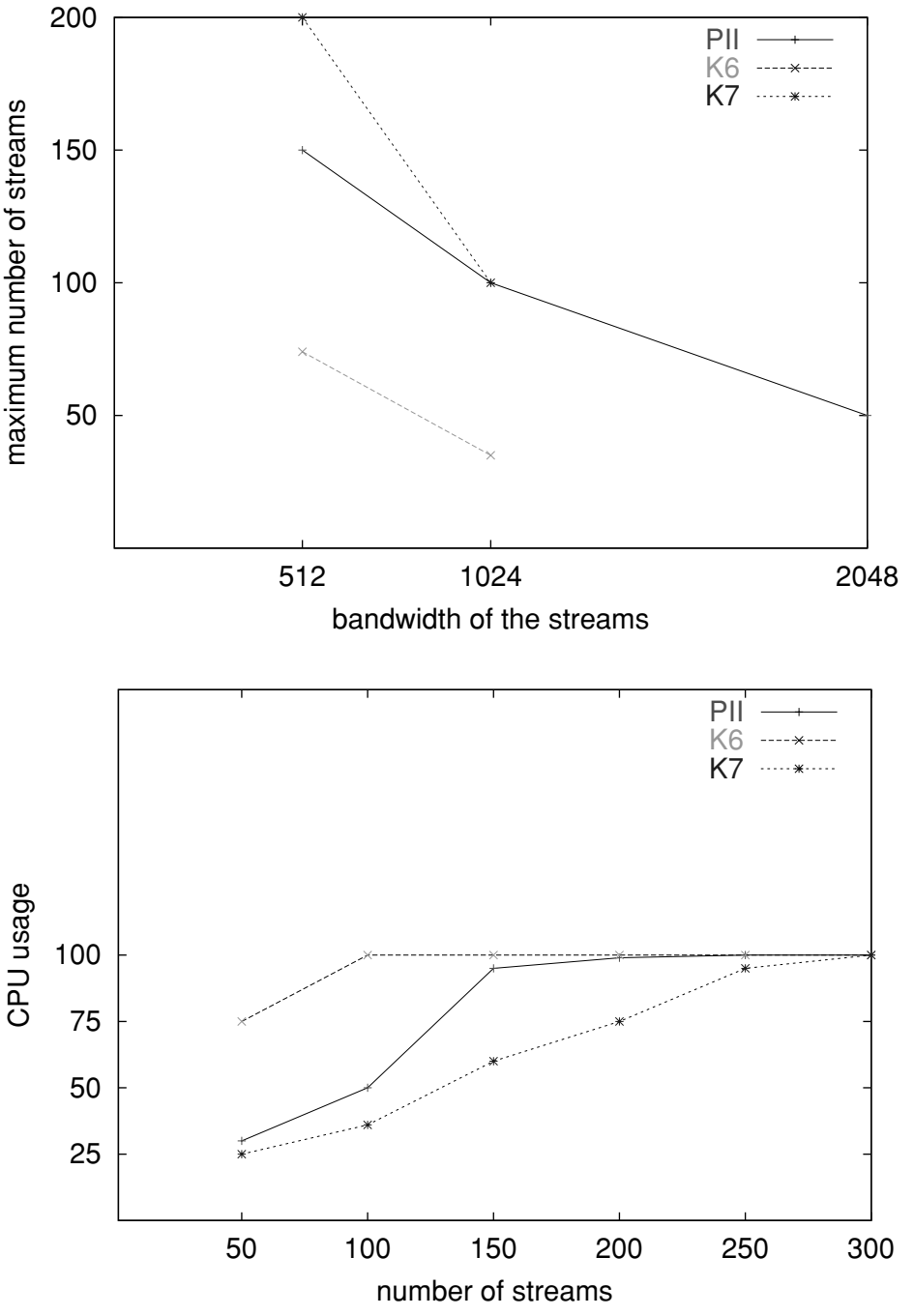

Fig. 9. Maximum number of streams and CPU usage for each of the servers.

interface certain hardware when Erlang does not provide the needed support. When performance goals are not met or surpassed, hardware cost is a low enough percentage of the total budget that adding a small number of physical nodes to the

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

streaming cluster to increase the available capacity is not a problem, plus it improves redundancy (Erlang built-in support helps developers define failover procedures to move services among nodes for system availability).

5.2 Performance in a cluster of nodes

The coarse nature of parallelism in our service – multiple streaming requests can always be mapped to different streaming servers, with a small coherence overhead – makes meeting performance goals much easier than in other development fields.

To show how easily the aggregate client-server throughput scales by adding frontend nodes, we configured a cluster of five frontend nodes, each with its own cache and sharing a common storage, and a redirector frontend to balance load. The whole system is deployed on identical nodes (IBM xSeries 200 with a Pentium-III 1GHz processor, 512MB RAM, single 18GB SCSI-160 disk, Broadcom 5700 1000Base-T interface, running Linux 2.4.19) over Gigabit Ethernet. Streaming frontends were restricted to 75 Mbps of client bandwidth. A simple client simulator was developed that requested a 3.8 Mbps MPEG stream and measured throughput, latency and packet loss. This was used as our test tool, creating a new request every five seconds. Figure 10 shows an average of total throughput, scaling linearly as the number of concurrent connections grows. This is to be expected, as the configured bandwidth is far below current hardware capacity (in this benchmark, over 115Mbps for each node). When more connections are requested, these are simply rejected. Figure 11 shows the effect of crossing node boundaries. For each request the redirector finds a suitable frontend node, where the client is redirected to. The chosen frontend then creates the needed *pipes* and starts a long-lived transfer. When the first request arrives at a certain node it has to load the media into cache and, after buffer is filled, it starts streaming. When another request comes, it finds the media already in cache and start latency is lower.

It may come as a surprise that load is not evenly distributed among nodes. While this behaviour is tunable (the redirector actually receives a functional parameter that decides how to choose a frontend) it is deliberate and intended to reuse cached media as much as possible. In real use scenarios, the capacity limiting factor is often the number of videos that can reside in cache simultaneously, as we lock space for the whole media.

5.3 Achieving higher throughput

Although in most occasions I/O performance has proven to be more than adequate, we have studied possible cases where current system throughput would not be enough and anticipated some alternative design for low-level I/O. The initial design of a low-level I/O server in C still posed a few problems such as integration and robustness guarantees. As an alternative, the idea was raised to extend the Erlang/OTP run-time to interface directly to the OS kernel `sendfile()` system call. This is a somewhat less general, but more integrated approach to high-throughput data moving from Erlang.

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

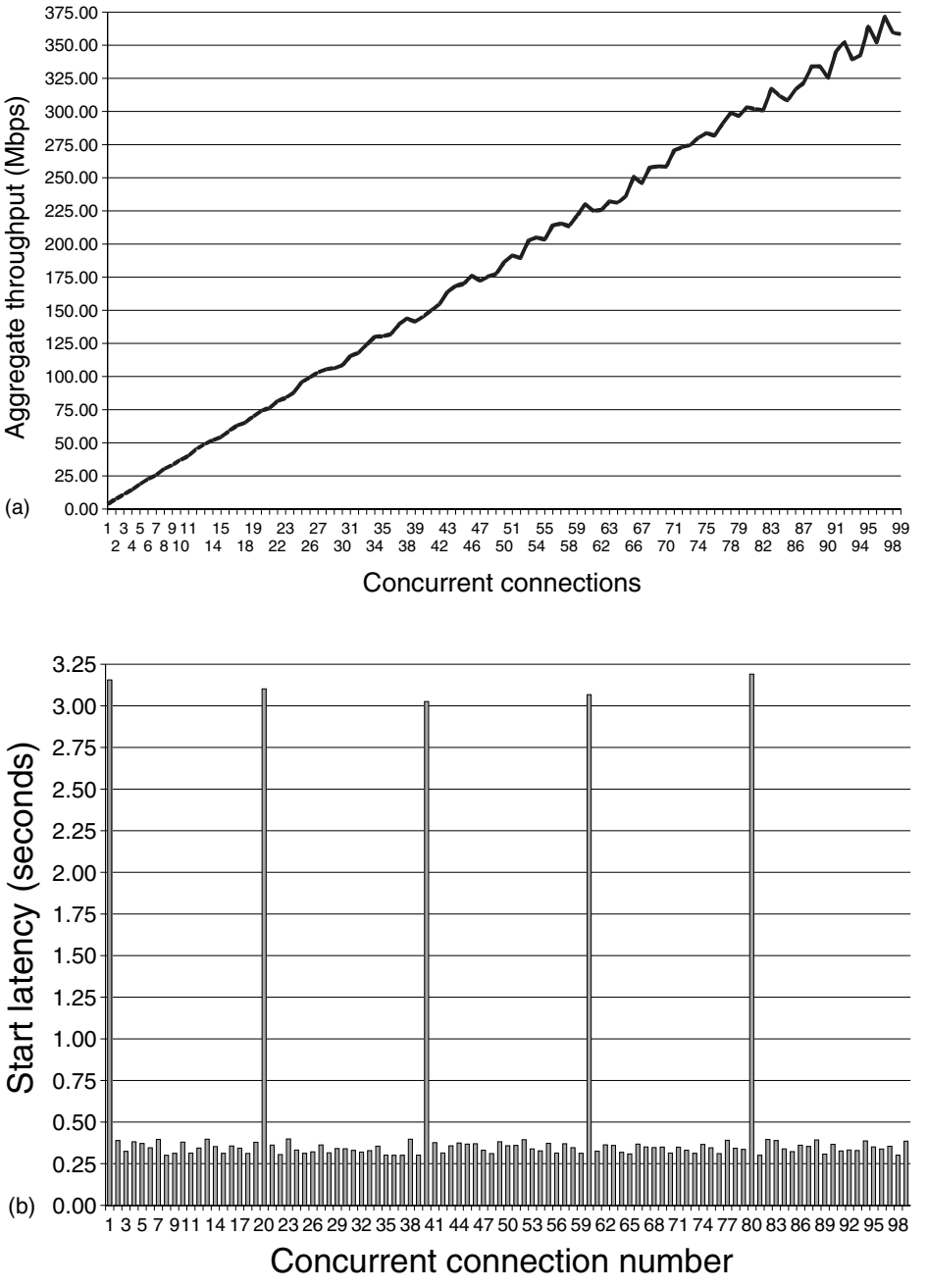

Fig. 10. (a) Aggregate bandwidth in a cluster. (b) Latency when starting connections in a cluster.

The `sendfile()` system call or a similar interface is present in all recent versions of the Linux, Solaris and AIX operating systems, among others. The OS kernel receives a source and a destination file descriptor, plus source offset and byte count, and directly copies the data from source to destination. Thus, two buffer copies from

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

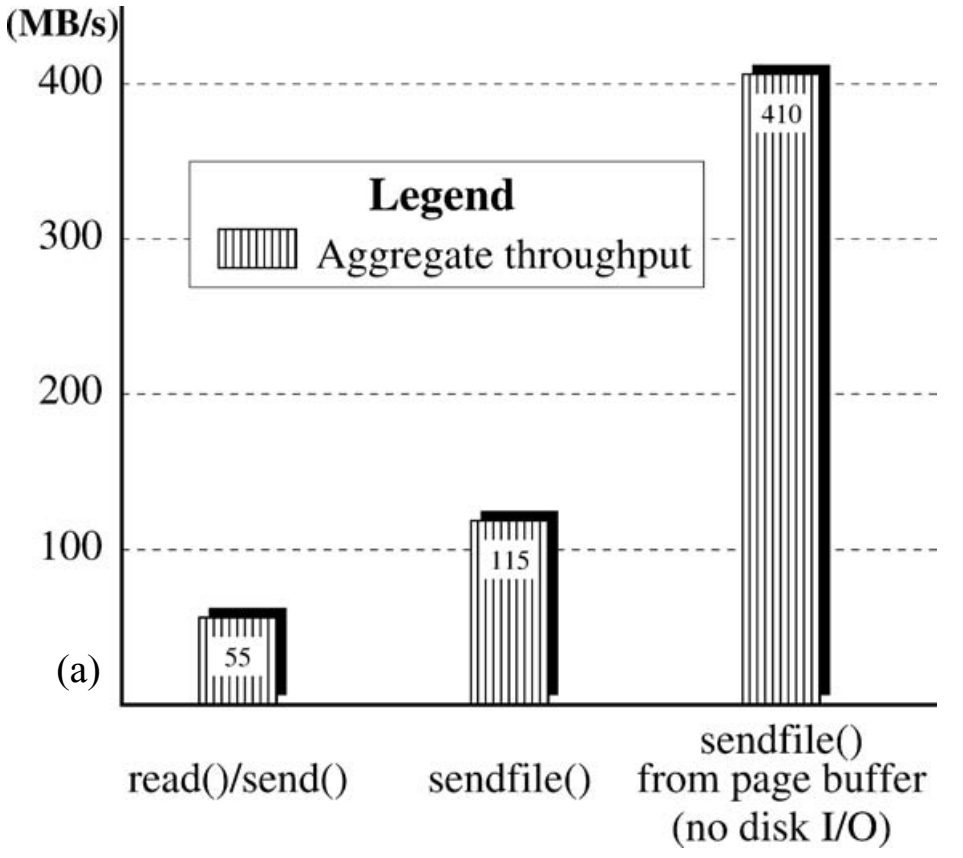

Fig. 11(a). Throughput comparison using different interfaces.

kernel to user space and vice versa are avoided, plus cache behaviour is improved; it was originally intended as a performance aid for web and file server workloads, where a large amount of processor time is spent copying data between user and kernel space. Additionally, we save the whole overhead of the Erlang memory management. The runtime does not move any data itself; it merely instructs the kernel to do it.

We exposed the `sendfile` semantics directly to the Erlang user code and compared the raw streaming performance of the previous hardware configuration in three scenarios: the server performing `read()/send()` cycles, using the `sendfile()` interface and this latter case when all the required buffers are in RAM; this was necessary as the disks ended up becoming our bottleneck.

Due to limitations in the network switch, 1500 byte frames were used; streams sent to clients do not tend to benefit much of 9KB jumbo frames, but inter-node transfers do, including cache loads. As an optimization, the kernel send and receive buffers were increased to 256KB from its defaults of 64KB. Values are the mean of 8 runs. 1GB was transferred in 4KB chunks in all cases. Tests were repeated with 2 to 8 concurrent transfers to ensure consistency.

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

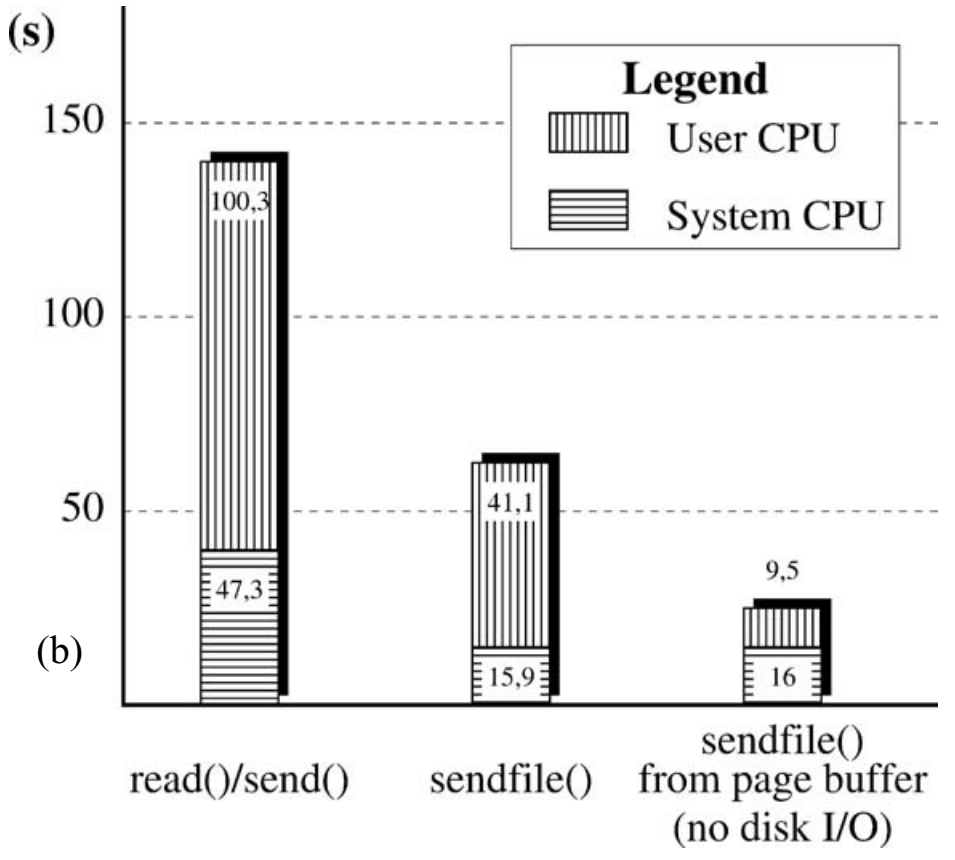

Fig. 11(b). CPU usage comparison with different interfaces.

From an initial result of about 55Mbps reading and sending data with the original Erlang I/O libraries, we got a speedup to 115Mbps just by using the developed `sendfile()` interface, which meant a trivial modification to our server code. In this case, the disk system was the limiting factor. Finally, the same test suite was modified to repeatedly send a subset of 256MB of data such that it would fit into RAM, and pages were locked into memory; this way disk I/O was eliminated for benchmarking purposes. The result of 410Mbps throughput over TCP is expected to be close to the maximum achievable on this hardware.

Certainly the Erlang I/O code developed using this interface will be in a rather imperative style; however so is the common Erlang file and network interface.

6 Conclusions

Some experience of using the concurrent functional language Erlang to implement a distributed video-on-demand server has been presented. VoDKA server is a real-world application that is currently in use successfully at different locations.

Clusters built from cheap off-the-shelf components represent an affordable solution for a large variety of applications that demand huge amounts of resources. The

https://doi.org/10.1017/S0956796805005502 Published online by Cambridge University Press

nature of the video-on-demand service, based upon multiple streaming sessions, fits extraordinarily well because of the independence between requests and the inherent geographic distribution of clients.

We consider that using a functional language has been a key success factor. The use of abstractions and compositionality help reducing the programming effort to adapt the system to the ever-changing domain. Erlang/OTP, in particular, has demonstrated to be a mature programming environment. The built-in concurrency constructs have simplified the design and implementation of the distributed control system. The runtime system seems to behave quite smoothly when dealing with thousands of concurrent processes with soft realtime requirements. Erlang design principles, tools and libraries have also proved to be quite valuable.

Regarding efficiency, our initial concerns were clearly unjustified. Even though we are aware that we are not achieving the maximum possible throughput for our hardware, Erlang input/output operations are efficient enough for prototyping the entire system and, in some cases, even for production use. Here, the abstraction of the communication transfers also helps us to identify which modules should be candidates for a low-level implementation: in production systems we have only a very small number of native modules for extremely timing-sensitive operations or to interface certain hardware. Moreover, the use of clusters encourages this idea, being cheaper to add new processing elements than the programming effort to optimize for a simple hardware. In addition, the whole system benefits of the increase in redundancy.

Among the criticisms to the language, we pointed out the lack of a static type checker and a structured module system that complicates the development at large. Even though these problems were known before starting the project, they became more pronounced as the server evolved. However, in spite of these notable deficiencies, the advantages overcome the problems.

References

Apple Computer Inc. (2004) *About darwin streaming server*. `http://developer.apple.com/darwin/projects/streaming/`.

Armstrong, J., Virding, R., Wikström, C. and Williams, M. (1996) *Concurrent Programming in Erlang, 2nd Ed*. Prentice-Hall.

Barreiro, M. and Gulias, V. (1999) Cluster setup and its administration. In: Buyya, R., editor, *High Performance Cluster Computing*, vol. I. Prentice Hall.

Carlsson, R. (2003) Parametrized modules in Erlang. *2nd ACM SIGPLAN Erlang Workshop*.

Chan, S.-H. and Tobagi, F. (1997) *Hierarchical storage systems for interactive video-on-demand*. CSL Technical Report CSL-TR-97-723. Computer Systems Laboratory, Stanford University, Stanford.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns Elements of Reusable Object-oriented Software*. Addison-Wesley.

Grand, M. (1999) *Patterns in Java: A catalog of reusable design patterns illustrated with UML, volume 1*. Wiley.

Gulias, V. (1999) *DFL: Distributed Functional Computing*. PhD thesis, University of Corunna, Spain.

Hoare, C. A. R. (1985) *Communicating Sequential Processes*. Prentice-Hall.

IBM. (2004) *Db2 content manager videocharger*. `http://www-306.ibm.com/software/data/videocharger/`.

Johansson, E., Pettersson, M. and Sagonas, K. (2000) A high performance Erlang system. *Proceedings 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, pp. 32–43. ACM Press.

Lea, D. (1997) *Concurrent Programming in Java: Design principles and patterns*. Addison-Wesley.

Marlow, S. and Wadler, P. (1997) A practical subtyping system for Erlang. *Proceedings 1997 ACM SIGPLAN International Conference on Functional Programming*, pp. 136–149.

Microsoft. (2004) *Windows media 9*. `http://www.microsoft.com/windows/windowsmedia/technologies/overview.aspx`.

Milner, R., Parrow, J. and Walker, D. (1992) A calculus of mobile processes. *Information & Computation*, **100**(1), 1–77.

Okasaki, C. (1996) Functional data structures. *The Second International Summer School on Advanced Functional Programming Techniques*.

Wadler, P. (1998) Functional programming: An angry half dozen. *Sigplan Notices*, **33**(2), 25–30.

Wilkinson, P., DeSisto, M., Rother, M. and Wong, Y. (1999) *IBM videocharger*. IBM redbook edn. IBM International Technical Support Organization.