J Supercomput (2015) 71:4663–4679
DOI 10.1007/s11227-014-1110-4

A stepwise approach to developing staged applications

Tiago Salmito · Ana Lúcia de Moura · Noemi Rodriguez

Published online: 31 January 2014

Abstract The *staged event-driven architecture* (SEDA) can be seen as a milestone as regards integration of threads and events in a single model. By decomposing applications into sets of multi-threaded stages connected by event queues, SEDA allows for the use of each concurrency model where most appropriate. Inside each SEDA stage, the number and scheduling policy of threads can be adjusted to enhance performance. SEDA lends itself to parallelization on multi-cores and is well suited for many high-volume data stream processing systems and highly concurrent event processing systems. In this paper, we propose an extension to the staged model that decouples application design from specific execution environments, encouraging a stepwise approach for designing concurrent applications, similar to Foster's PCAM methodology. We also present Leda, a platform that implements this extended model. In Leda, stages are defined purely by their role in application logic, with no concern for locality of execution, and are bound together through asynchronous communication channels, called connectors, to form a directed graph representing the flow of events inside the application. Decisions about the configuration of the application at execution time are delayed to later phases of the implementation process. Stages in the application graph

This work is partially supported by the Carlos Chagas Filho Foundation for Research Support of the State of Rio de Janeiro (FAPERJ) under Grant no. E-26/100.519/2011 and the National Scientific and Technological Research Council (CNPq).

T. Salmito (✉) · A. L. de Moura · N. Rodriguez
Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Rio de Janeiro, RJ 22451-900, Brazil
e-mail: tiago@salmito.com; tsalmito@inf.puc-rio.br

A. L. de Moura
e-mail: amoura@inf.puc-rio.br

N. Rodriguez
e-mail: noemi@inf.puc-rio.br

can then be grouped to form clusters, and each cluster is mapped to an exclusive OS process, running on an arbitrary host. Finally, we discuss two example applications which we developed to evaluate the Leda platform.

Keywords Concurrency · Threads · Event-driven · Staged events

1 Introduction

Over the last years, concurrency models based on a combination of threads and events, also called *hybrid models* [1,7,12], have been the focus of some attention. In previous works [15,16], we discussed how, although all proposals of hybrid models try to benefit from the advantages of both concurrent abstractions, it is possible to group them into categories according to their bias towards events or threads. *Event-driven* hybrid models extend one-threaded event-driven systems to take advantage of multiple cores by running independent event loops in separate threads [3,22]. *Thread-based* hybrid models, on the other hand, provide thread-like facilities on top of event-based systems that maintain local information by implicitly creating closures that encapsulate state [2,8,12].

A third and relevant category is the *staged* model, which exposes both abstractions to the programmer. This model was inspired by SEDA (staged event-driven architecture) [20] and adopted in a number of works [7,9,19]. In a staged application, the processing of tasks is carried out by a series of modules (*stages*) connected by event queues. Inside each stage, concurrency is typically provided by preemptive multi-threading. Task scheduling policies are local to each stage, permitting a flexible tuning of an application. The staged event-driven concurrency model is well suited for many high-volume data stream processing systems and highly concurrent event processing systems.

The staged architecture strives to decouple the execution of different stages, allowing concurrency to be treated as a local problem. However, connected stages are still dependent on each other, because each stage must explicitly define the destination stage for each event it emits. This requirement does not facilitate reusing a stage when composing different applications. The structure of a staged application, i.e., the granularity of the tasks associated to each of its stages, is also strongly influenced by its execution environment. In the original SEDA design, for instance, each stage is a self-contained module, naturally associated to its own scheduling parameters and pool of resources. This ties the granularity of the logic of the application to the granularity of runtime resource decisions, which may not be the same.

In this paper, we propose an extension to the staged event-driven architecture that aims to take its flexibility one step further, decoupling the application design from specific execution configurations. In our proposed extension, the programmer defines the structure of a concurrent application through a graph whose nodes are independently designed stages and whose edges are asynchronous communication channels called *connectors*. Later, to configure the application execution environment, the application graph is partitioned into *clusters* that are mapped, at execution time, to OS processes that may be running on different hosts. Different, and configurable, scheduling policies—such as the number of allocated threads and priority mechanisms—can be applied to each individual cluster.

The separation of the definition of the application structure from the partition of the application graph into independently managed clusters allows for a wider set of execution scenarios to be explored without impacting the application structure and the specification of its stages.

The rest of this paper is organized as follows. Section 2 describes the staged concurrency model. Section 3 describes our proposed extension to the staged event-driven model, and Sect. 4 presents the implementation of Leda, a concurrent platform based on our proposed architecture. Section 5 illustrates the development of a Leda application. In Sect. 6, we evaluate the Leda platform through two example applications. Finally, Sect. 7 contains some final remarks.

2 The staged event-driven architecture

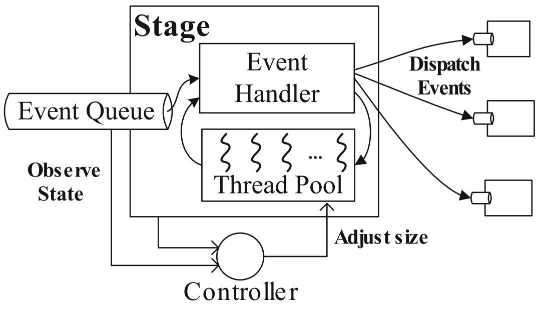
The staged event-driven architecture (SEDA) [20] introduced a modular approach for the design of concurrent systems combining events and threads. In this approach, the processing of tasks is divided into a series of self-contained modules (*stages*), connected by event queues.

A SEDA stage, illustrated in Fig. 1, is composed by an event queue, an event handler and a thread pool. The threads in the pool constantly remove events from the stage's input queue, and execute parallel instances of the event handler, which provides the core logic of the stage. Typically, the processing of an event generates one or more events that are dispatched to other stages' event queue.

SEDA's design decouples event handling from thread allocation and scheduling. An event handler is not responsible for managing its threads; instead, an underlying stage controller, provided by the runtime system, adapts the stage's resource usage dynamically. Controllers can observe a number of factors, such as the stage's input queue length, response time and throughput, and adjust scheduling parameters according to some predefined policy. Because different scheduling policies can be configured, and are local to each stage, systems can control the level of concurrency at a finer grain and employ appropriate scheduling policies for the specific characteristics of each stage.

A SEDA application can be viewed as a graph with nodes representing stages and edges representing the event flow through these stages. Figure 2 illustrates the structure

Fig. 1 A SEDA stage



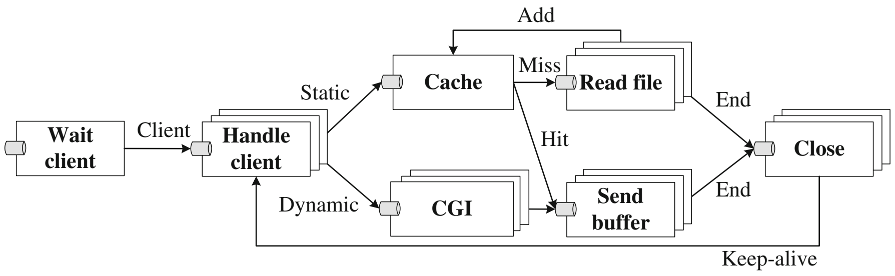


Fig. 2 A staged web server

of a SEDA application with an example of a web server implementation. The stages of this server fall into three general categories: client connection handling (*Wait client*, *Close*), request processing (*Cache*, *CGI*) and I/O (*Read file*, *Send buffer*).

In the original SEDA implementation, all stages are executed inside a single OS process and a shared memory queue implements the connections between them. The implications of these choices are twofold. On one hand, they make the design unsuitable for distributed memory environments. On the other, there is no system-enforced memory protection among stages, and inside a single stage it is up to the programmer to control accesses to shared space. Even the implementation of different schedulers for different stages is complicated by the fact that all schedulers must execute inside the same addressing space. Some proposals of staged concurrency based on SEDA overcome the first limitation using different implementations of event queues that can connect stages executing in different OS processes or in different hosts [9,19]. As for the problem of shared memory, existing staged systems do not, in general, make specific provisions for avoiding race conditions, with the exception of Aspen [19].

Another issue that deserves attention in the original SEDA implementation is the tight coupling between stages. Because each stage explicitly specifies the destination of the events it triggers, the logic of each stage is intermingled with the role of this stage in the specific application at hand, reducing opportunities for reuse.

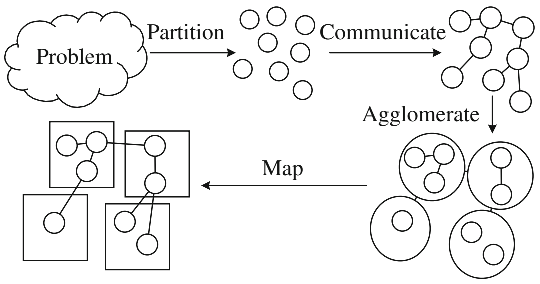
3 Extending the staged model

The SEDA architecture uses stages to create a concurrency model which provides modularity and exposes both threads and events as programming abstractions available to the programmer. Our goal is to evaluate whether this model can be extended to allow further degrees of decoupling. By providing a general well-defined communication interface for stages, enforcing a shared-nothing environment, and allowing the same application to be mapped to different execution contexts, we believe we can reduce the complexity of programming and tuning concurrent applications.

Stepwise application development

Our extension supports the idea that the design of a concurrent application should be carried out in several steps, much in line with the methodology for developing parallel

Fig. 3 Foster's design methodology



applications described by Foster [4]. Foster proposed *PCAM*, a design methodology for parallel programs based on a task/channel abstract model of computation. This methodology structures the design process as four distinct phases: partitioning, communication, agglomeration and mapping. The *partitioning* phase exposes opportunities for parallel execution, creating as many tasks as possible. In the *communication* phase, the information flow among tasks is specified. The third phase, *agglomeration*, moves towards concrete implementations, agglomerating the tasks identified in the partitioning phase so as to obtain a reasonable number and granularity of tasks. The fourth and final phase in PCAM is the *mapping* of tasks to physical processors. Figure 3 contains a conceptual representation of the task/channel design process showing these phases. PCAM encourages the development of scalable and reusable programs by delaying machine-dependent considerations until the latest possible moments. In our extension, the developer is naturally led to a PCAM-like design process.

In our extended model, the design of an application begins by decomposing its functionality into a set of stages. Analog to Foster's *partitioning* step, this decomposition captures the application's potential modularity and concurrency opportunities, producing the maximum possible number of tasks. One important difference, however, is that Foster's partitioning can use either *domain* or *function* decomposition. As in SEDA, each stage defines a different event handler which may on its turn generate other events to be handled by other stages. So, partitioning is always based on functionality. At runtime, several concurrent *instances* of a same stage may execute in parallel, providing domain partitioning. Differently from SEDA, in our model stages are oblivious of the stage that will consume these events. Instead of sending events to specific destinations, stages emit events through *output ports* bound, in a later step, to asynchronous channels (*connectors*) that connect communicating stages. By decoupling the design of communicating stages, our model allows for the reuse of stages in different applications.

In the next step, the general structure of the application is defined by binding the stages' input and output ports through *connectors*—objects that represent event queues. Binding stages through connectors is similar to binding components in architecture description languages (ADLs) [6]. The result of this binding process is a graph representing the application structure, with nodes representing stages and edges representing connectors. In Fig. 4, part (a) depicts an example of such a graph with four stages (*A*, *B*, *C* and *D*), their output ports, and four connectors.

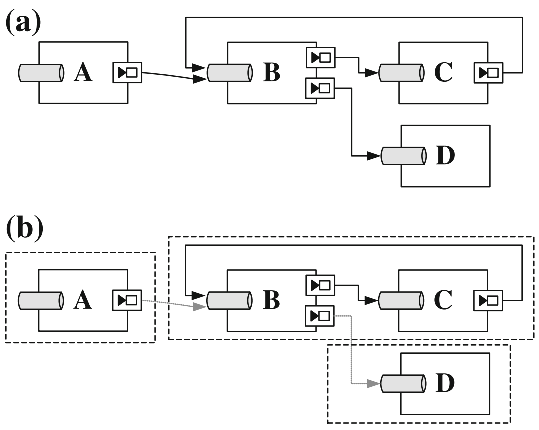


Fig. 4 A stage graph of an example application

After binding stages through connectors, the application developer must verify the communication patterns between stages and determine whether the corresponding connectors may carry information that cannot be exchanged between different OS processes (such as file or socket descriptors). Such connectors are then marked as *local*.

In the third step of the application design, analog to Foster's *agglomeration*, the stages in the application graph are partitioned into *clusters*, which will be mapped to independent execution units. This partition must comply with the communication patterns defined before, i.e., stages communicating through local connectors must reside in the same cluster. This requirement allows the programmer to rely on the guarantee that the communicating stages may freely exchange process-related information.

Clusters represent the granularity at which the programmer will be able to define different scheduling policies, and their size may range from a single stage to the whole application. Part (b) of Fig. 4 depicts a possible partition of the graph in part (a). Because the agglomeration process is fully separated from the previous steps in the design of an application, the developer is free to experiment with different partitions in order to achieve a good balance between fine-grained scheduling policies and computing/communication granularity.

The fourth and final step, *mapping*, occurs at the moment of running the application, when each cluster is mapped to an OS process running on an arbitrary host. The fact that this mapping is delayed to this step allows the developer to run the same partitioned graph over different distribution scenarios.

4 The Leda platform

Leda supports the stepwise development of applications discussed in Sect. 3. All of the involved steps, except for the first one, are typical configuration tasks. A scripting, interpreted, language such as Lua [10,11] is a natural choice for these tasks. The ease of interfacing Lua with C also makes Lua adequate for the coding of event handlers, for the programmer can handle computing-intensive events with compiled code as needed. Lua also provides important facilities for the implementation of Leda itself.

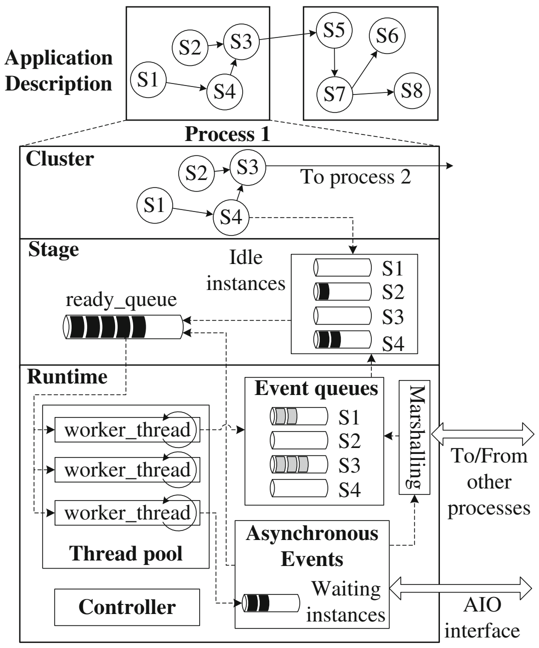


Fig. 5 Leda's runtime architecture

Leda enforces memory isolation both between different stages and between different instances of a single stage. Event handlers for stages are typically stateless, maintaining no persistent state between different instantiations. When strictly necessary, the programmer may mark a stage as *stateful*, and Leda will guarantee that the different executions of its event handler will be serialized.

Because Lua is designed to have its code invoked from a C host program, it is easy to tweak the execution environment for event handlers so as to satisfy the requirements established in the Leda concurrency model. To enforce memory isolation, event-handler instances are executed in user-level threads with exclusive addressing spaces provided by *Lua states* [10]. A Lua state represents an independent interpreter state and keeps track of functions and global variables, among other information. Lua's C API allows us to divorce Lua states from kernel threads: the programmer can either execute each Lua state inside a different kernel thread or share a limited number of kernel threads among an arbitrary number of Lua states [17].

Figure 5 illustrates the architecture of our implementation. The *Application Description* component maintains a high-level representation of the application graph, in which each of the configured clusters has been mapped to a different OS process. Each process starts by reading a description of its cluster and loading the event-handler code of its stages (Lua binary bytecode format). Inside each process, a *Controller* component defines scheduling parameters, such as the size of the thread and Lua state pools, the maximum size of all event queues, etc.

The runtime implementation relies on a pool of worker (*kernel*) threads and a pool of Lua states dedicated to the execution of event handler *instances*. The number of worker threads should be roughly related to the number of available cores, while the number of instances can be much higher. At any given time, an event-handler instance can be in one of four execution states: *IDLE* (waiting for events), *READY* (bound to an event), *RUNNING* (processing an event), and *WAITING* (waiting for the result of an asynchronous operation). Because event-handler instances are user-level threads, it is possible to attach or detach them from worker threads as needed. Each worker thread repeatedly fetches an event-handler instance from the *ready_queue* and runs it until it either completes or requests an asynchronous operation. In the case of stateful stages, only one event-handler instance is created, and events emitted to a stateful stage are processed sequentially by this single instance.

The *Asynchronous Event* runtime component deals with asynchronous I/O operations. It is composed by an event queue and an event loop and runs on an exclusive kernel thread. This component intercepts I/O calls from event-handler instances and resumes their execution only when the requested operations are ready, by putting them back on the ready queue. Our implementation supports two high-performance, asynchronous event-driven I/O interfaces: *epoll* and *AIO* in Linux. *Epoll* provides readiness notification of sockets descriptors and *AIO* allows disk accesses to overlap in the background.

The *Asynchronous Event* component also implements interprocess communication when events are emitted to stages in other clusters. The *Marshaling* component serializes and deserializes Lua data structures when destination stages are located on other processes.

5 Designing staged applications

In this section, we illustrate the development of a Leda application using the skeleton of an application that we developed for the Computer Graphics Laboratory at PUC-Rio. This lab runs a project with the Brazilian Oil Company to study riser breakage; one of the project's goals is to support visualization of breakage simulation. The input file for this visualization contains the initial geometry of the points composing the oil riser and the position of these points for each simulation step. This information is handled by a method of finite elements, in this case cylinders approximated by prisms with rectangular sides. The higher the number of sides, the better the quality of visualization. The output of the finite element method is then rendered on the computer screen. The whole process is usually repeated several times for best results.

The project team wanted to write a program to handle files with large numbers of steps. For small data sets, after the file is read and the finite element method applied, the resulting data are kept in memory and rendered over and over again, and the time taken to initially generate the data to be rendered is not very relevant. However, for large numbers of steps, it is not feasible to keep all the visualization data in memory.

Leda was used to create a solution for this problem as follows. An initial stage (*reader*) reads the input file and forwards the data to the stage that implements the finite prism method (*extrusion*). This stage can benefit from domain decomposition (Leda instances), as different prisms can be computed independently. Extrusion results are fed to a last stateful stage (*renderer*), which works as a reducer, collecting all the data for renderization. This last state may receive data related to different steps in

Fig. 6 A simplified example of a stage graph definition

```
1: local reader = leda.stage{
        handler = do_read,
        init = init_read,
        autostart = true
   }
2: local extrusion = leda.stage{
        handler = do_extrusion,
        init = init_extrusion
   }
3: local renderer = leda.stage{
        handler = do_render,
        init = init_render,
        stateful = true
   }
4: graph = leda.graph{
        reader:connect("segment", extrusion),
        extrusion:connect("prism", renderer)
   }
5: -- This is an example of graph partitioning
6: graph:part(graph:all() − extrusion, extrusion)
7: graph:map("host1 : port", "host2 : port")
8: graph:run()
```

intermingled events. It keeps a Lua table containing an entry for each step. When it receives all the data for the next step, this step is marked as ready for exhibition. A step is rendered for a minimum time on the screen for comfortable viewing. In the original version, as we mentioned, renderization data for all the steps were kept in memory. In the Leda version, as soon as the rendering stage moves to a new step, data from the last one is thrown away. This is trivial in Leda because of Lua's support for garbage collection. Figure 6 illustrates the configuration of the application, which reflects our proposed design methodology, based on PCAM.

Lines 1 to 3 of Fig. 6 create the three stages in the application. This corresponds to the *partitioning* step, in which opportunities for functional decomposition are explored. The **`leda.stage`** API function creates new stages. It receives a Lua table defining a stage handler (in this case, functions `do_read`, `do_extrusion` and `do_render`), and optionally an initialization function (respectively, `init_read`, `init_extrusion`, and `init_render`), and configuration parameters. The implementation of the event handlers is shown in Fig. 7 for illustration. Function **`leda.stage`** returns a reference to the newly created stage.

The configuration parameter **`autostart`**, used when creating stage **`reader`**, indicates that an instance for this stage should be automatically put in the *ready_queue*

Fig. 7 Examples of Leda event handlers (code of auxiliary functions omitted for brevity)

```
local $input\_file$ = assert($arg[1]$)
local $sides$ = assert($arg[2]$)
function DO_READ( )
    local $fd = io.open(input\_file, "r")$
    for $line\_segment$ in $read\_segment(fd)$ do
        leda.send(“$segment$”, $line\_segment$)
    end
    $fd : close()$
end
function DO_EXTRUSION($segment$)
    local $line\_element = calc\_prism(segment, sides)$
    leda.send(“$prism$”, $line\_element$)
end
function DO_RENDER($element$)
    if not $lines[element.line]$ then
        $lines[element.line] = \{\}$
    end
    $table.insert(lines[element.line], element)$
    if $complete(current\_line)$ then
        ... -- render current line (Code omitted)
        $lines[current\_line] = nil$
        $current\_line = (current\_line+1)\ \%\ maxlines$
    end
end
```

upon its creation. Parameter **`stateful`**, used for stage *renderer*, indicates that it must have a single instance with persistent state. Stage *extrusion* is stateless and the system can maintain several instances of its event handlers running concurrently (the exact number of instances to be created can be defined as command-line parameters).

Line 4 of Fig. 6 invokes function **`leda.graph`** to configure the application. This corresponds to the *communication* step. It is at this point that the programmer binds output ports to destination stages, creating connectors between them. At runtime, when an instance of a stage emits an event to an output port, the corresponding connector will direct it to the appropriate (local or remote) event queue depending on this configuration.

Function **`part`**, called in line 6, partitions the application into clusters of stages, performing the *agglomeration* step. It receives a list of stage sets, each of which will form a cluster. Auxiliary function **`all`** returns a set with all stages in the application. Sets of stages can be manipulated using the following operators: union (+), difference (−) or intersection (*). In our example, the application is partitioned into two clusters: one containing a single stage (*extrusion*) and the other with the rest of the application graph.

Next, line 7 maps the graph of clusters to OS processes using function **`map`**, which binds each application cluster to a running process listening on the indicated port. This corresponds to the last step, *mapping*.

Finally, in line 8, the invocation of **`run`** starts the execution of the application. Although here the application configuration script is shown as a single chunk of Lua code, the four PCAM steps can be described in separate files, facilitating the decoupling between application logic and execution.

Event handlers are invoked with parameters that are transparently extracted from a dequeued event. Leda offers a single **`leda.send`** primitive for handlers to emit events to other stages. Events are chunks of data that are delivered to the connector bound to a provided output port (in our example, ports *segment* and *prism*).

The Computer Graphics Lab is currently running this application on different machines. Depending on the number of available processors and prism sides that are required for each finite element, the user can sometimes observe a slowdown, but in most configurations this is not the case. The result is considered satisfactory by users that can now work on real-world, very large, data sets.

6 Evaluation

In this section, we describe two experiments that evaluate different aspects of the Leda platform. In the first one, we explore the decoupling between logic and execution environment that is the goal of our extension. In the second experiment, we seek to evaluate Leda's overhead in a typical concurrent application, comparing a Leda web server with well-established implementations.

All tests presented in this section were conducted on a multi-core CPU (Intel Xeon X5650 2.67GHz processor) with 24 cores and 24 GB of RAM running the Linux operating system. The chosen Linux distribution was Debian, with kernel version 2.6.32 x86_64. The performance results that are reported represent average values obtained from three executions.

Exploring Leda's flexibility

This experiment explores the flexibility of our model, showing how different execution configurations can be used to tune the performance of an event-stream processing application.

Complex event processing systems are an emerging class of applications that process data from various sources, often distributed, that generate large volumes of events [21]. Events can be filtered and correlated to identify complex patterns. Applications also often apply windows to sequences of events received from arbitrary sources to detect relevant patterns. In contrast to the detection of simple events, this kind of processing may result in heavy loads.

Figure 8 shows the graph of our test application, made up of seven types of stages. The *Source* stage extracts events from a web server log. Each *Source* runs in an exclusive Leda cluster, which is mapped to a process that resides in the same host as the web server. The *Window* stage implements a spatial window, sending out arrays

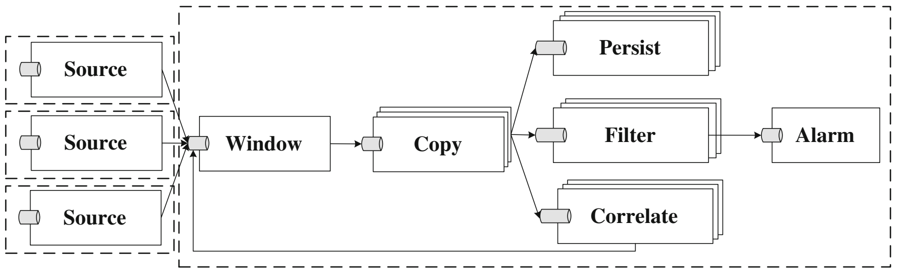


Fig. 8 Synthetic complex event processing application graph

of events. The *Copy* stage generates different copies of these arrays o be processed in parallel by *Persist*—which writes them to disk—, *Correlate*—which correlates events in a window—, and *Filter*—which filters events of interest. Finally, the *Alarm* stage executes an action if an event of interest is detected.

Given this application graph, there are several different possible execution configurations. The whole application can be executed in a single cluster, sharing a common pool of operating system threads, or it can also be partitioned in more than one cluster. Besides, Leda can create different numbers of stage instances, allowing for different degrees of domain parallelism. Finally, the pools of operating system threads can have different sizes.

We report the results of running the application using two different configurations: (1) a single cluster grouping all processing stages, excluding sources (represented by dashed rectangles in Fig. 8), and (2) two clusters, one of them exclusively for running the most CPU-intensive stage (*Correlate*). The number of Leda instances is, at the moment, described by command-line arguments, as is the size of the thread pool of both processes.

Table 1 shows total execution times for a set of different scenarios. Tests were performed by handling pre-stored log files containing a total of one million entries, and the size of the spatial window was set to 1,000 elements. Each configuration was executed with 24 (one thread per core) and 48 (two threads per core) operating system

Table 1 Execution time of test scenarios for the two graph configurations

Configuration	Threads	Number of instances			Execution time (s)
		Persist	Filter	Correlate	
1	24	6	6	12	107.836
2	24	6	6	12	98.720
1	24	12	12	12	104.015
2	24	12	12	12	102.200
1	48	12	12	24	112.116
2	48	12	12	24	102.329
1	48	24	24	24	111.629
2	48	24	24	24	111.616

threads. In the dual-process case, OS threads were split up equally between the two processes. Different choices for the number of instances of each stage were also tested.

The dual-process configuration in general presented better results than the single-process case. The cause for this is probably the fact that in the dual-process configuration, part of the execution resources (OS threads) are dedicated only to the computing-intensive stage.

All in all, results in Table 1 show that variations in runtime parameters lead to differences of more than 10 % in execution times, and thus testify to the importance of allowing the programmer to easily experiment with different execution scenarios.

Evaluating Leda's performance

As a second experiment, we apply Leda to a classic web server scenario to evaluate the overhead incurred by its infrastructure. Our web server implementation was derived from Xavante [14].

The Xavante web server is implemented entirely in Lua, using coroutines to serve both static and dynamic content for various simultaneous clients. These coroutines are responsible for handling client requests and dispatch requested files or sending the results of the execution of a dynamic script. We adapted the Xavante architecture to the staged model by organizing the code executed by the server coroutines into stages. The resulting application graph is shown in Fig. 2.

Stage *Wait client* executes a loop that waits for TCP connections from clients in a predefined port, and passes the client socket through its `client` port for further processing. Because the Leda implementation provides asynchronous interfaces for sockets, this stage does not block any threads while it waits for new connections.

The *Handle client* stage reads and processes requests, creating a Lua table describing the requested HTTP headers. After processing the request, this stage selects the type of the content according to the requested file (static or dynamic), and routes the request, along with its corresponding client socket, to the appropriate output port.

If the request is for static content, it goes to a stateful *Cache* stage that caches previously read files. The *Read file* stage handles events from the `miss` port of the *Cache* stage and uses the asynchronous I/O interfaces to read files from disk and send the response headers back to the client. After *Read file* finishes reading the whole file, it send a completion event back to the *Cache* stage to insert the new data into the cache. The cache uses a simple LRU (Least Recently Used) policy to store temporary content and has a configurable maximum memory size.

Requests for dynamic content are handled by the *CGI* stage, which executes the requested script and forwards the resulting buffer. The *Send data* stage sends contents in memory to clients from both *Cache* and *CGI* stages. If the client requested a persistent connection (through the keep-alive header), the client socket is put back on the event queue of the *Handle client* stage to process a new request. Otherwise, the client connection is closed.

With the exception of *Cache*, stages of the server do not require persistent state due to the stateless nature of the HTTP protocol; therefore, clients can be handled by parallel instances inside each stage. All connectors used in this application are local, because stages need to exchange socket descriptors. All stages of the application must thus run in a single operating system process.

We compared the behavior of our staged web server with two implementations used in production environments: Apache (version 2.2.22) [5] and NginX (version 1.5.2) [13], both implemented in C.

The Apache server was configured to preinitialize 150 threads in the pool and scale up to 2,000 threads. The Leda and Nginx implementations use one thread (or process in the case of Nginx) for each available core.

We used the HTTPerf tool [18] to generate requests to a set of files from 10 hosts, with a total number of concurrent clients varying from 100 to 5,000 for each tested server. Test cases handled three situations: serving static content of small files, serving static content of large files and serving dynamic content.

Figure 9a shows the average response time of requests served and the response throughput in each server for files with size of 300 kB. In this scenario, the Apache web server achieved better results because its architecture with multiple threads is more adequate to maintain low latency in situations where the server is not overloaded. The fact that all requests were handled with success (no timeout errors) corroborates this conclusion.

Next, we increased the size of the requested files to overload the file system. The requests then started failing because of connection timeouts. Figure 9b shows the average response time and the response throughput for requests for 5 MB files. In this scenario, the average response throughput reaches its plateau between 200 and 300 concurrent clients. Apache shows a degradation when overloaded due to its use of large numbers of threads. The Leda and Nginx implementations maintained a near constant throughput, with Leda presenting slightly better results. This is expected behavior in event-driven systems, where the throughput of events remains constant when the application is overloaded and the excess work is accumulated in the event queues.

The last test scenario, serving dynamic content, consists in requesting a Lua script that executes a `for` loop of 30,000 iterations. This evaluates the behavior of the servers in a situation with large processing demand. Figure 9c shows the average response time and the response throughput for Apache and Leda for requests to dynamic contents. In these graphs, we can see the effect of the overhead of using a large number of threads when the request involves a large processing load. The latency of the Leda implementation shows a smoother growth, because the available processing resources on the server are used more efficiently with the use of asynchronous I/O interfaces for sockets.

These results show that concurrent applications written in Leda can present acceptable performance, and indicate that the cost of the provided flexibility is low.

7 Final remarks

Among the existing proposals for hybrid concurrency models, the staged event-driven architecture stands out for allowing the programmer to balance the use of threads and events, exposing both abstractions as coding tools. Besides, staged architectures decouple the execution of different stages, allowing scheduling decisions to be tai-

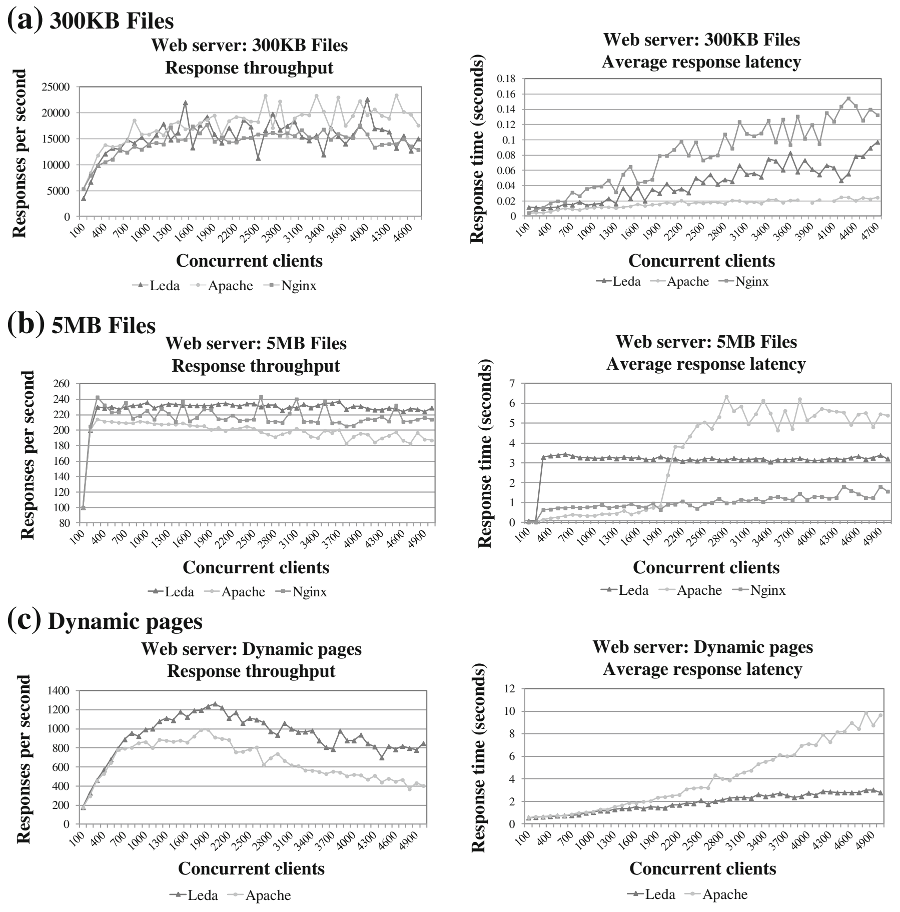


Fig. 9 Web server test results

lored to the characteristics of each stage. In this paper, we presented a model that extends the decoupling between application logic and runtime scheduling by establishing a stepwise procedure for developing concurrent applications. In our model, stages are initially defined purely by their role in application logic, with no concern for locality of execution, and are bound together through asynchronous communication channels, called connectors, to form a directed graph representing the flow of events inside the application. The resulting graph must later be partitioned in clusters, and these must be mapped to processes that may run either on a single host or in an arbitrary distributed configuration. Because the partitioning step is decoupled from the definition of the graph, different partitions can easily be applied to the same application, allowing the programmer to experiment with different granularities.

The combination of Lua states with operating system threads allows Leda to free the programmer from dealing with race conditions: stateless event handlers are executed in separate, disposable, environments, and stateful handlers are serialized to guarantee safe execution.

The Leda architecture allows for seamless extension of execution scenarios to distributed-memory machines, as the mapping phase makes no distinction between local and remote hosts. Many stream processing applications that operate with large amounts of data, such as weather forecasting, log processing and stock market analysis, inherently process and compose data from different autonomous domains. Distribution is a key requirement in these situations.

A number of extensions have been proposed to the original SEDA architecture [7, 9, 19]. However, the novel feature of Leda is the focus on the degree of decoupling between application logic and runtime scheduling and configuration decisions. SEDA already makes it possible to control the number of threads dedicated to each stage. In Leda, clusters allow the granularity of scheduling to be defined by the programmer, decoupling thread pools from application logic. Besides, the reification of instances as an execution parameter allows for further experimentation and tuning. These facilities are specially important when a single application is intended for execution in different platforms: different hardware support and operating system configurations will result in different parameter choices for best execution results.

References

1. Adya A, Howell J, Theimer M, Bolosky WJ, Douceur JR (2002) Cooperative task management without manual stack management. In: Proceedings of the general track of USENIX annual technical conference, Monterey, CA, USA, pp 289–302
2. Behren RV, Condit J, Zhou F, Necula GC, Brewer E (2003) Capriccio: scalable threads for internet services. In: Proceedings of the 19th ACM symposium on operating systems principles, Bolton Landing, NY, USA, pp 268–281
3. Dabek F, Zeldovich N, Kaashoek F, Mazières D, Morris R (2002) Event-driven programming for robust software. In: Proceedings of the 10th workshop on ACM SIGOPS European workshop, EW 10, New York, NY, USA, pp 186–189
4. Foster I (1995) Designing and building parallel programs, chap 2. Addison-Wesley, Reading
5. Foundation AS (2013) The apache web server. http://www.apache.org. Accessed July 2013
6. Garlan D, Monroe R, Wile D (2000) Acme: architectural description of component-based systems. In: Leavens GT, Sitaraman M (eds) Foundations of component-based systems. Cambridge University Press, London, pp 47–67
7. Gordon ME (2010) Stage scheduling for CPU-intensive servers. Ph.D. thesis, University of Cambridge, Computer Laboratory
8. Haller P, Odersky M (2007) Actors that unify threads and events. In: Proceedings of the 9th international conference on coordination models and languages, Paphos, Cyprus, pp 171–190
9. Han B, Luan Z, Zhu D, Ren Y, Chen T, Wang Y, Wu Z (2009) An improved staged event driven architecture for master-worker network computing. In: Cyber-enabled distributed computing and knowledge discovery, CyberC '09, Zhangjiajie, China, pp 184–190
10. Ierusalimschy R (2006) Programming in Lua, 2nd edn. Chap 24, Lua. Org
11. Ierusalimschy R, de Figueiredo L, Celes W (2011) Passing a language through the eye of a needle. Commun ACM 54(7):38–43
12. Li P, Zdancewic S (2007) Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In: Proceedings of the 2007 ACM SIGPLAN conference on programming language design and implementation, PLDI '07, New York, NY, USA, pp 189–199
13. (2013) Nginx: the nginx webserver. http://nginx.org. Accessed July 2013
14. Project K (2004) Xavante webserver. http://www.keplerproject.org/xavante. Accessed July 2013

15. Salmito T, de Moura AL, Rodriguez N (2011) Understanding hybrid concurrency models. Revista Brasileira de Redes de Computadores e Sistemas Distribuidos 4(2):33–41
16. Salmito T, de Moura AL, Rodriguez N (2013) Flexible approach to staged events. In: Sixth international workshop on parallel programming models and systems software for high-end computing (P2S2)
17. Skyrme A, Rodriguez N, Ierusalimschy R (2008) Exploring lua for concurrent programming. J Univers Comput Sci 14(21):3556–3572
18. Software H (2013) Httperf. http://www.hpl.hp.com/research/linux/httperf. Accessed July 2013
19. Upadhyaya G, Pai VS, Midkiff SP (2007) Expressing and exploiting concurrency in networked applications with Aspen. In: Proceedings of the 12th ACM SIGPLAN symposium on principles and practice of parallel programming, PPoPP '07, New York, NY, USA, pp 13–23
20. Welsh M, Culler D, Brewer E (2001) Seda: an architecture for well-conditioned, scalable internet services. SIGOPS Oper Syst Rev 35(5):230–243
21. Wu E, Diao Y, Rizvi S (2006) High-performance complex event processing over streams. In: Proceedings of the 2006 ACM SIGMOD international conference on management of data. ACM, pp 407–418
22. Yoo S, Lee H, Killian C, Kulkarni M (2011) Incontext: simple parallelism for distributed applications. In: Proceedings of the 20th international symposium on high performance distributed computing. ACM, pp 97–108