Computer Science • 14 (4) 2013 http://dx.doi.org/10.7494/csci.2013.14.4.645

Leszek Siwik
Kamil Włodarczyk
Mateusz Kluczny

STAGED EVENT-DRIVEN ARCHITECTURE AS A MICRO-ARCHITECTURE OF DISTRIBUTED AND PLUGINABLE CRAWLING PLATFORM

Abstract *Crawling systems available on the market are usually closed solutions dedicated to performing a particular kind of task. There is a meaningful group of users, however, which require an all–in–one studio, not only for executing and running Internet robots, but also for (graphical) (re)defining and (re)composing crawlers according to dynamically changing requirements and use–cases. The Cassiopeia framework addresses the above idea. The crucial aspect regarding its efficiency and scalability is concurrency model applied. One of the promising models is staged event–driven architecture providing some useful benefits, such as splitting an application into separate stages connected by events' queues– which is interesting, taking into account Cassiopeia's assumptions regarding crawler (re)composition. The goal of this paper is to present the idea and PoC implementation of the Cassiopeia framework, with special attention paid to its crucial architectural element; i.e., design, implementation, and application of staged event–driven architecture.*

Keywords web crawler, composable software, distributed web crawling framework, event–driven architecture, event-driven processing, SEDA

1. Introduction

Nowadays, Internet robots, crawlers, or spiders are probably the most popular and the most-frequently developed computer programs and systems worldwide. It is enough to mention that, according to the *www.user-agents.org* service, there were 2461 agents classified as robots, crawlers, or spiders introducing themselves with their own user agent in 2010.

Simultaneously, although there are so many implementations, it would be really difficult to point out the best one, since assessment of: provided functionality, architecture, efficiency, effectiveness (i.e., –the quality–) depends on a particular use case and requirements.

One may ask, however, if it is possible to develop yet another Internet robot which provides a (closed) set of available functionalities and realizes its tasks in a fixed, predefined way, but rather a framework for defining and composing robots from provided building blocks – i.e., small (pieces of) functionalities – and next, setting their parameters up and just running on provided, efficient and scalable runtime environment. Additionally, such a framework should enable providing new building blocks and use them to (re)define and to adjust (also in the runtime) crawlers and robots to varying and dynamic situations and requirements. Equally important are the tracking and monitoring (stages of) the tasks' realization and analyzing gathered results as well.

If all of this could be done in as easy as only possible way (i.e., in a graphical drag&drop way), for a huge number of analysts (for instance marketing, financial, or criminal) who perform very tough, specialized, and varying crawling, searching, or analyzing tasks – such a toolkit would be a great improvement in their daily work. Currently, instead of focusing on doing their jobs, they have to deal often with a whole set of different crawling and searching systems and tools (one per group of particular kind of tasks) with all the negative consequences of such a situation (licensing, compatibility, maintenance, know–how, migration, etc.).

One has to remember that enormous size and unimaginable structural complexity of the WWW network [4, 1, 7] are reasons why, from a technical and architectural point of view, developing effective Internet robots – and the more so – developing a framework for supporting graphical robots composition – becomes a really challenging task. To ensure appropriate and suitable effectiveness, such solutions have to be developed according to really effective architecture [5, 15] calling here Apoidea [16], UbiCrawler [3] or topic–driven crawlers [11], just to mention a few interesting solutions.

Due to the same reason, the crucial aspect when a crawling system is designed, is applying the appropriate concurrency model. There are two classical concurrency models (thread–based and event–driven ones), but both have important shortcomings; and the question is if there are reasonable alternatives being able to significantly improve crawling effectiveness and addressing simultaneously (pre)assumed flexibility and ability for composing crawlers from provided building blocks. In this context,

staged event driven architecture (SEDA) [18] seems to be a very interesting and promising specification.

The goal of this paper is to present the idea, the architecture, and the proof–of–concept implementation of the Cassiopeia framework; i.e., easy to use, all–in–one studio for (re)defining, (re)composing, and then executing and running Internet robots and crawlers and simultaneously advanced, effective and efficient, distributed, agent–based crawling environment with advanced concurrency model applied.

The paper is organized as follows: in section 2, the most important top-level functional and nonfunctional assumptions are defined; in section 3, top-level architecture and its particular elements are briefly discussed; in section 4, a typical concurrency model is discussed; in section 5, staged event–driven architecture is presented; in section 6, some details of SEDA implementation for Cassiopeia purposes are presented; in section 7, preliminary results of simple experiment(s) made on Cassiopeia PoC implementation with the use of experimental Cassiopeia Web Crawler (CWC) along with preliminary results of SEDA implementation assessment are presented; and finally, in section 8, short conclusions and future work are discussed.

2. Top level assumptions

The goal of the Cassiopeia project is to design and develop an open and flexible framework for composing, defining, instantiating, launching, running, monitoring, and managing distributed crawlers – or widely – Internet robots, as well as for storing and analyzing gathered results. Below, the most important, top level functional and non–functional assumptions are listed and briefly commented on:

- Cassiopeia is a framework which enables (graphical) (re)composition of Internet robots from available building blocks; i.e., small(er) functionalities available on this platform.
- It enables a redefinition of composed crawlers, also in the runtime, without their recompilation or even restarting.
- Cassiopeia is not limited only to (pre)defined building blocks, it enables the addition of new building blocks not available on the platform so far.
- Since crawling tasks can be really demanding and long–lasting, an efficient and effective concurrency model is implemented and applied. What is important, it is self–manageable and transparent since the end user wants to focus on logical task's definition and analyzing its results–not on its implementation and execution details.
- Again, taking complexity of (some) crawling tasks into account:
  - framework is easy–to–scale and distributed architecture is (pre)assumed. If needed, it is easy to add new logical and physical computational units and redistribute running tasks among them. What is important, it doesn't affect the effectiveness and efficiency of the framework itself;

  - the architecture doesn't assume any constraints regarding geographical deployment of computational units. Task execution units are independent and effective models, and channels of communication among them are applied.
- It is fault tolerant so:
  - single points of failure are reduced (if not eliminated at all);
  - if some of execution units fail, realization of their tasks is taken over by the rest of computational units. It is done automatically, without restarting task execution;
- Running (parts of) the framework and task execution is possible on many different popular hardware and software configurations.

3. Cassiopeia platform architecture at a glance

Taking the above (and of course many others aspects and requirements) into consideration, the following architecture of the Cassiopeia platform has been proposed (see Fig. 1).

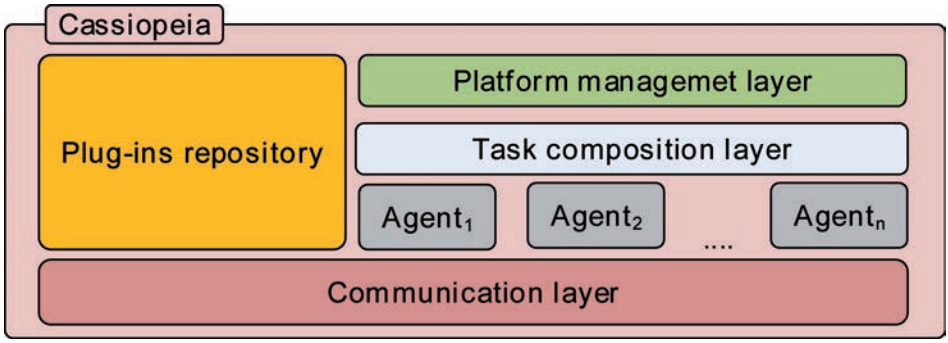

Figure 1. Cassiopeia top level architecture.

The most important computational and processing units are Cassiopeia agents[1]. Their architecture and efficiency decide mainly about general Cassiopeia efficiency.

All agents' components are implemented as beans created and managed within the Spring Framework container. During their implementation, such mechanisms as IoC container as well as spring support for JMX, JMS, and batch jobs have been used.

Particular elements of Cassiopeia agent's architecture are presented in Figure 2. Its two most important architectural components are: events' distribution service and SEDA runtime environment, being the SEDA implementation developed for Cassiopeia purposes. SEDA specification, as well as its proprietary Cassiopeia implementation,

[1]We are not talking about FIPA compliant agents but just about logical computational units designed according to given below architecture

are discussed in next sections. But first, in section 4, typical concurrency models and their advantages and shortcomings are discussed to explain why SEDA is such a promising and interesting architecture in the context of the Cassiopeia framework.

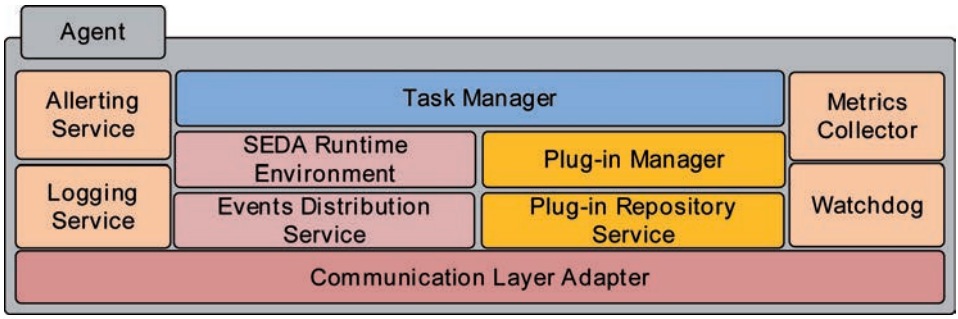

Figure 2. Cassiopeia agent's architecture.

4. Concurrency models

In the context of any concurrent systems (and crawling systems in particular), the crucial element is the model of concurrency applied. Choosing the appropriate strategy of managing threads and processes, as well as tasks scheduling, can significantly improve crawling efficiency and effectiveness. On the other hand, one has to deal with threats connected with inappropriate application or implementation of the chosen model. Below, two of the most important concurrency models; i.e., concurrency based on the pool of threads and event–driven concurrency, are summarized.

4.1. Concurrency based on the pool of threads and processes

The most popular model of concurrency (especially in the case of processing requests by servers) is „one request–one thread/process" model. Such a model is supported by both modern operating systems and programming languages and environments as well. In such a model, the operating system cares about switching processor(s) among threads/processes evenly. What is very convenient from developer's/architect's point of view, however, is the fact that the efficiency of such a system can be significantly reduced when the number of threads/processes increases.

To avoid such a situation, some systems limit the number of threads/processes that can be simultaneously created and processed; and if the top limit is reached, new requests are simply not accepted. Such an approach allows us to avoid the problem with efficiency, however increases latency what is also of course not desirable. It is a pretty popular approach and is implemented, for instance, by the Apache web server or Tomcat application server, but is not appropriate for systems with massive concurrency, such as crawling systems.

4.2. Event–driven concurrency

Limitations and problems with using an uncontrolled pool of threads cause that developers and architects give up such an approach and use event-driven concurrency [12]. In such a model, there is a relatively-small number of threads (usually one per CPU) working in infinite loops and processing different events provided by input queues. Such an approach implements task processing as a finite state machine where transitions between following stages are triggered by events. In contrast to the previous approach, the application itself is able to control how a given task/request is being processed.

Applying such a model of concurrency allows for avoiding problems discussed earlier with reducing system efficiency when the number of threads increases. In this case, when the number of requests increases, application throughput increases either – until the top limit is reached. In such a case, each following request is scheduled in the input queue and is processed as only required resources become available again. In such a model, application throughput stays constant even in workload peaks and latency grows linearly–not exponentially as in the previous case.

One of the issues related to the event–driven concurrency model is that the application itself has to care about events' scheduling and queuing. It has to make a decision, for instance, when to start processing incoming events and how they should be scheduled and ordered. What is more, it has to be done keeping the service level balanced and minimizing the latency. That is why scheduling, dispatching, and prioritizing algorithms are crucial for system efficiency and usually are implemented and adjusted individually for given application and its use cases what results, among the others, in problems with extending the application with new functionalities. In such a case, dispatching algorithm and concurrency management mechanisms also have to be likely replaced. Flash Web Server, with its AMPED (Asymmetric Multi-Process Event Driven) architecture, is an example of a server based on such a model of concurrency [13].

None of the typical, above-mentioned concurrency management models are ideal approaches. That is why research on alternative models is still conducted to propose efficient and convenient architecture for concurrent and distributed applications. One promising model is staged event driven architecture–SEDA [18], to some extent mixing both approaches discussed above and providing some additional interesting (and important from crawling and crawlers composition platform's point of view) benefits such as splitting an application into separate stages connected by events queues.

5. Staged Event Driven Architecture

Staged Event Driven Architecture–SEDA was proposed at the University of California in 2000 by: Matt Welsh, Steven D. Gribble, Eric A. Brewer and David Culler [18]. SEDA mixes both discussed in the previous section approaches; i.e., concurrency based on the pool of threads and the event–driven concurrency. It provides task scheduling

mechanisms and makes it possible to manage task execution parameters in runtime. It also makes it possible to reconfigure the application itself depending on its workload.

SEDA consists of a network of nodes called stages. With each stage, there is an associated input events' queue. Each stage is the independent module which is managed individually depending on input queue parameters. The possibility of monitoring by each node's/stage's input queue makes it possible to filter and prioritize events, and to resize and manage the pool of threads used by itself appropriately to the actual situation, the number of events to be processed, and the general workload.

Thanks to this, the SEDA–based application becomes really flexible since it is workload–resistant on the one hand and doesn't consume resources if it is not necessary on the other [18]. There are, of course, some limitations regarding the efficiency of SEDA–based applications [14], and even the author of this specification has some–both positive and negative–remarks and thoughts about this architecture [17]. But what is important and useful when the platform for software composition is being designed, introducing stages and connections among them makes it really easy and natural to decompose the application into separate, easy–to–replace modules.

5.1. SEDA stage

According to the SEDA specification, stage is a fundamental processing unit in this architecture. It's a one single application component consisting of: events' processing unit, events' input queue, and the pool of threads. Each stage is also equipped with a controller responsible for scheduling and thread allocation (see Fig. 3). Threads, in the pool of stage, handle events coming from the input queue, process them in the events' processing unit, and produce zero or more events on the input queue of the next stages.

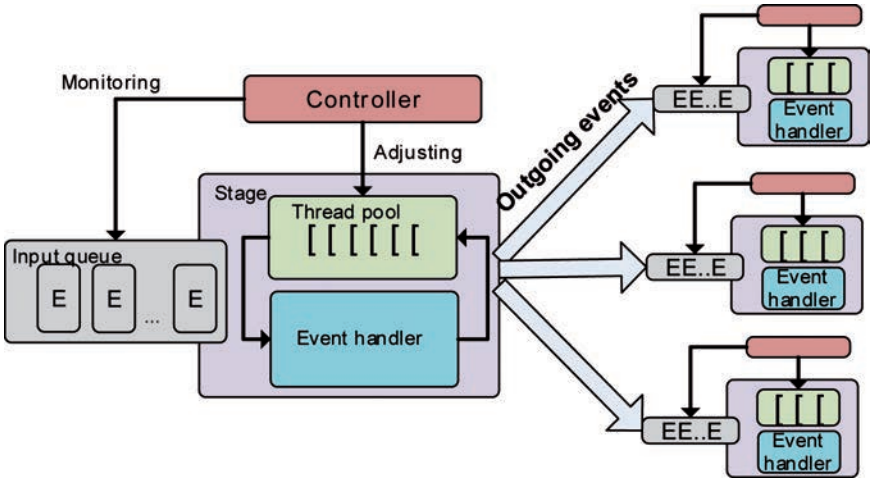

Figure 3. The structure of: the stage and the application in SEDA [18].

Application based on SEDA is defined by the set of connected stages creating directed graph, where stages are connected by events' queues.

5.1.1. Stage controllers

One of the main assumptions of the SEDA approach is to exempt developers from the problem of resource-assigning management. To manage each stage separately, SEDA introduces so–called controllers (see Fig. 3). They are responsible for automatic adjustment of allocated resources depending on the actual efficiency and demand for resources. Controllers constantly monitor the runtime environment and adjust their parameters to achieve assumed goals–i.e., the maximum stage's throughput usually. Authors of SEDA specification proposed two solutions regarding resource management by controllers.

The first one is applying a thread pool controller responsible for adjusting the number of threads working in a given stage depending on the size of the input queue. Its goal is to adjust the number of allocated resources according to the assumed efficiency and the service level, avoiding simultaneously the situation when too many resources are allocated (and wasted, in fact). In such an approach, controller periodically checks the size of the input queue and, if only its size exceeds the given threshold, it allocates an additional thread to the pool, releasing (some) threads if only idle time appears.

The second solution proposed by the SEDA authors is the so–called group of controllers responsible for adjusting the number of events processed simultaneously. It is true that processing more events improves the throughput; but on the other hand, too many events processed simultaneously can increase latency. Group controller monitors the frequency of appearing events on its output and reduces the number of processed events until its throughput decreases. In such a case, it increases the number of simultaneously-processed events.

Besides the above-mentioned default controllers predefined by SEDA authors–more sophisticated solutions, such as taking into account events' priority, can be applied of course. Just to mention some examples of successful application of the SEDA approach are such solutions as Sandstorm (service framework), Haboob (`httpserverbasedonSandstorm`), or Gnutella (p–2–p file exchanger) can be mentioned.

To recapitulate, enormous Internet size and complexity [1, 4, 7] requires efficient architectural solutions, especially when web crawlers are designed and implemented – it is enough to mention such interesting solutions as Apoidea [16], UbiCrawler [3], or topic–driven crawlers [11]. Traditional concurrency (i.e., concurrency based on the pool of threads/processes) is often not a proper solution. On the other hand, event–driven concurrency seems to be a more-effective approach but has the opinion to be more complicated and complex, especially for developers and architects since many aspects controlled by operating system in traditional model have to be handled by the application itself in this case.

SEDA tries to link both of these approaches as a really promising alternative and not forcing developers to deal with difficult low–level concurrency management issues, allowing additionally for natural application decomposition. That is why this very approach has been chosen and applied as a concurrency management model and architecture in the Cassiopeia framework. In the next section, selected details of SEDA implementation for Cassiopeia purposes as well as its incorporation and adjustment to Cassiopeia architecture are discussed.

6. SEDA implementation for Cassiopeia purposes

Although there are some open–source and enterprise SEDA implementations[2,3,4] since for Cassiopeia platform purposes it has to meet some additional needs and requirements, our own implementation of SEDA specification has been developed [10].

Mentioned in Figure 2 SEDA runtime environment is an implementation of SEDA architecture working within one–single JVM. Its top level elements are presented in Figure 4. As one may see, besides crucial SEDA elements such as stages and queues, also some additional–coming from Cassiopeia specification and needs–components such as: monitoring, notification and events distribution mechanisms have been implemented.

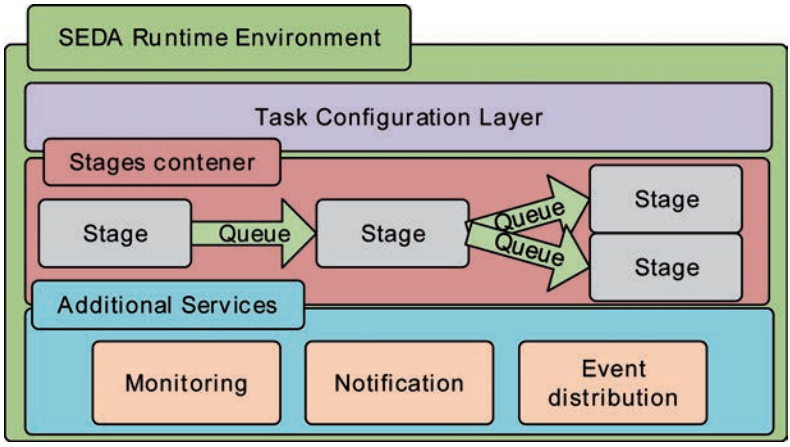

Figure 4. SEDA runtime environment design.

[2] http://jcyclone.sourceforge.net
[3] http://www.mulesoft.com/mule-esb-open-source-esb
[4] http://sourceforge.net/projects/seda/

6.1. SEDA Runtime Environment implementation

SEDA runtime environment is responsible for configuration, allocation of all required resources, launching and running application, monitoring all application runtime parameters, releasing unnecessary resources, and stopping the application.

Implementation of the SEDA runtime environment is provided by SEDA class. Its two most important methods are *start()* and *stop()*. The *Start()* method is responsible for launching task provided as its argument. Initially, the *start()* method checks if all conditions required for starting a task are fulfilled; i.e., if the provided task is a proper one, and if SEDA environment itself is in a *STOPPED* state. If so, the provided task is being configured and launched; after launching all stages–SEDA environment changes its state to *RUNNING* one.

Stop() method is responsible for stopping all task's stages and switching the SEDA environment back to *TERMINATING* and STOPPED states.

The task configuration layer is realized by *TaskDescriptor* class, which is responsible for providing task configuration read from the XML file. The main method in this class is the *configure()* method, which is responsible for creating task stages along with their controllers, connections among them, and after that, returning the instance of the *ConfiguredTask* class being configured and ready for launching task.

6.1.1. Stage implementation

The stage is a separate module of the application, fulfilling the SEDA stage's specification. Each stage developed for Cassiopeia purposes consists of a plugin, a managing module, an input queue, a controller, and a thread pool, as shown in Figure 5.

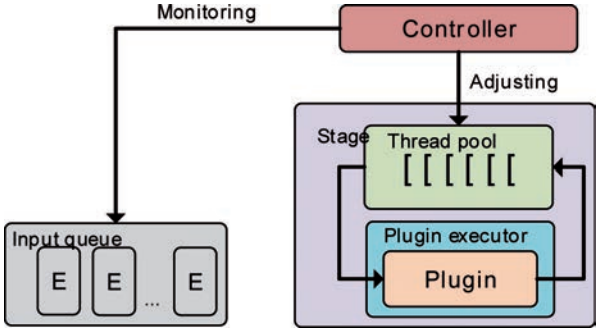

Figure 5. The architecture of the stage.

Plugin represents the event's handler presented in Figure 3, and it defines business logic realized by the particular stage. Because it is designed as a stage's element, it naturally meets the SEDA specification. And because it is defined as a plugin managed by the plugin manager known from Figure 2, it can be easily replaced – what supports perfectly the general assumption that Cassiopeia should not be yet another crawler, but rather a pluginable framework for crawlers' composition.

The thread pool is responsible for executing business logic defined in the plugin, and the controller monitors the size of the input queue and resizes the pool of threads, if necessary.

Plugins are provided as implementation of one of two abstract classes; i.e., *DataProviderPlugin* or *ProcessorPlugin* class. *DataProviderPlugin* class implements one single method; i.e., *provideData()*. This method is executed by the SEDA runtime environment exactly once and is primarily responsible for providing the configuration and application starting parameters, such as the initial URL of web crawler. All classes inherited from *ProcessorPlugin* abstract class have to provide implementation of *process()* method, which is responsible for realization of plugin's business logic consisting in processing events appearing on stage's input queue. This method is realized cyclically in the thread pool. The input queue is an event (input data) source for the stage as well as the means of communication between stages. Its implementation extends *LinkedBlockingQueue* class from *java.util.concurrent* package and adds *putEvent()* method, which is responsible for adding events to the input queues and handling exceptions. The threads' pool is provided by *StagePool* class. It decorates *ThreadPoolExecutor* class, from *java.util.concurrent* package. The *StagePool* class is responsible, among others, for providing default threads' pool configuration and for its resizing, if necessary. The controller's implementation is divided into the two main parts. The first one is a controller class run as a separate thread and periodically launching mechanisms responsible for thread pool management. The second one is implementation of the *PoolSizeStrategy* interface, defining a set of methods responsible for thread pool management.

For Cassiopeia platform purposes, a single strategy of thread pool management has been implemented; i.e., when the size of the input queue exceeds a predefined threshold, additional threads are allocated until the top limit provided as a configuration parameter is reached. If only the size of the input queue falls below the given value, the size of the threads' pool is also decreased. As one may notice, if only a different thread pool management strategy is necessary, it is enough to provide different implementation of *PoolSizeStrategy* interface.

The elements discussed above come directly from the SEDA specification; but for Cassiopeia purposes, some additional elements have been implemented in the SEDA runtime environment. First of all, in the presented SEDA implementation, communication among stages is realized with the use of the events' router being the part of the SEDA runtime environment. Thanks to providing the appropriate addressing strategy and a lower–level communication layer, it is possible to realize communication among the stages working on separate JVMs, which is schematically presented in Figure 6.

In the presented SEDA implementation, there is also a dedicated component responsible for calculating and collecting statistics describing on–line the parameters of the SEDA runtime environment. Actually, such information as stage activity and workload, the size and the workload of input queues, and the size of the threads' pool is collected. Thanks to this, any efficiency problems as well as deadlocks and bottlenecks

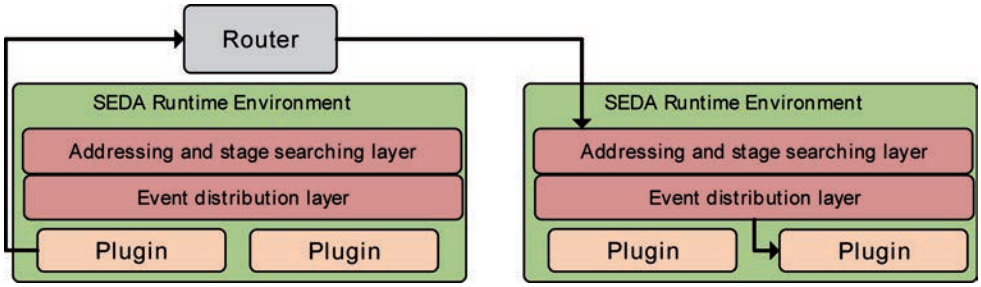

Figure 6. The schema of communication between stages running on different JVMs.

can be identified and notified. The presented implementation has been equipped also with an event-notification mechanism. The appropriate module is available both for the maintenance and for the development team to inform about the system resource allocation. Actually, notifications are realized as email messages, and such information as sender address, smtp connection parameters, and recipients address list are provided as configuration parameters.

7. Preliminary experimental verification

Taking the top-level requirements and the main architectural assumptions and decisions into account, it seems that preliminary experimental verification of Cassiopeia PoC implementation should assess its two crucial aspects; i.e., SEDA implementation and concurrency model applied as well as ability for composing crawler(s) from provided (experimental) building blocks.

7.1. Preliminary performance tests of SEDA implementation for Cassiopeia purposes

After implementation of SEDA for Cassiopeia framework purposes, preliminary tests regarding, among others, assessing application throughput depending on increasing workload, have been performed. To perform such tests, two plug-ins have been implemented; i.e., *EventGeneratorPlugin* and MockPlugin. *EventGeneratorPlugin* being a *DataProviderPlugin* implementation has been responsible for generating the given number of events per second (EPS) on input queue of the next stage, whereas *MockPlugin's* (implementation of *ProcessorPlugin*) responsibility has been reduced to handling events from its input queue, waiting for a while (configurable parameter) and then forwarding this event on to the input queue of the next stage (if only such exists). During the experiment, *EventGeneratorPlugin* generated from 1 to 40 EPSs. The wait time for *MockPlugin* was set to 300ms, and the maximum number of threads in its pool was set to 10.

As presented in Figure 7a, along with the increasing number of events, the application throughput also increases until the maximum value is reached. For initial

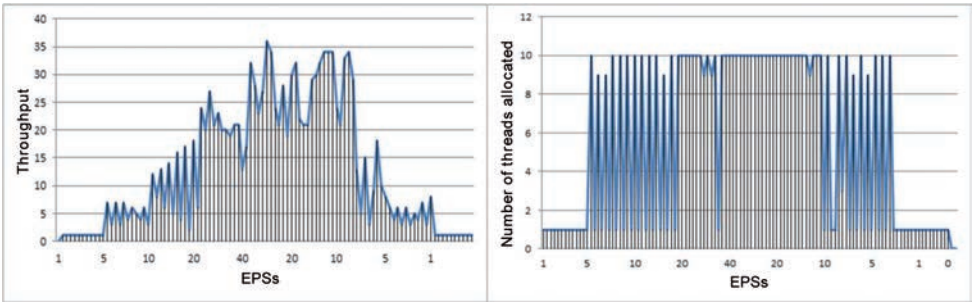

Figure 7. The throughput (a) and the number of allocated threads (b) as a function of generated workload [10].

1, 5, 10, 20 EPSs, the average application throughput is almost equal to the generated workload. When 40 EPSs workload has been generated, all possible resources (10 threads per stage) have been used, and the maximum possible throughput (c.a. 30 EPSs in considered configuration) has been reached.

Because more events than the maximum throughput were generated, they were queued in the stage's input queue. Consequently, even when the workload has been decreased, the maximum application throughput has been kept for a while (i.e. until the queued events have been processed and the size of the input queue was decreased). This confirms that presented SEDA implementation proved its flexibility and self–management ability regarding adjustment of its configuration to the actual workload.

For comparison, in Figure 7b, there is the number of threads allocated by the application during the same test presented. For 1–5 EPSs, one thread in the stage pool was enough to process all incoming events. When the number of EPSs increased, the number of threads started to oscillate between 1 and 9–10. It is not a desirable effect, and results from the fact that *ThreadPoolExecutor* class possesses its own thread management mechanism and its own task queue, and observed behavior suggests that probably actual used thread pool management based on classes from java.util.concurrent should be replaced with its own implementation.

When the number of EPSs reached 40, the thread pool was used entirely; and such a state has been kept until all events queued in input queue have been processed. After that, the number of threads in the pool was reduced to its minimum value.

On the basis of the performed tests, it can be said that extending the original SEDA specification and adjusting it to Cassiopeia's needs and requirements doesn't affect its efficiency, providing simultaneously a natural support for crawlers composition and distribution. It seems, thus, that the proposed SEDA implementation can be successfully applied for distributed, high concurrent, and composable framework such as Cassiopeia. So the next step is composing and running simple crawling tasks to prove that it works properly all together, and that the defined requirements have been fulfilled. Obtained results are presented in next subsection.

7.2. Preliminary functional verification–Cassiopeia Web Crawler

To assess the correctness and the usability of a designed Cassiopeia framework as a toolkit for composing and running web crawlers, a simple Cassiopeia Web Crawler (CWC) has been defined, composed, and run. According to taken assumptions, it was designed as a Cassiopeia task. It realizes breadth first algorithm and visits pages belonging to defined (sub)domain only.

7.2.1. Cassiopeia Web Crawler definition

CWC structure i.e. its set of plug–ins and the structure of connections among them is presented in Figure 8.

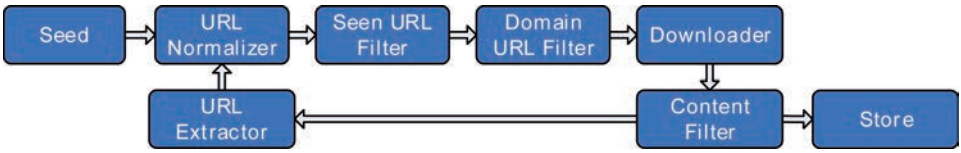

Figure 8. Definition of Cassiopeia Web Crawler (CWC).

Seed is a plug–in of *DataProvider* type, so SEDA Runtime Environment launches it only once at the beginning of task execution, and it has no inputs. As a configuration parameter, it takes an initial URL address. As an output, it returns URL address(es) that should be processed.

URL Normalizer takes the URL address appearing on its input and returns its normalized version on its output. Consequently, such URL's identifiers as *http://example.com http://example.com/* and *http://example.com:80* are recognized as one single URL address. During crawling task execution, it is (more than) possible that the same URL identifier will be found many times and, consequently, would be many times analyzed, downloaded, etc. To avoid such a situation, the CWC definition consists–among others– of Seen URL Filter plug–in which is responsible for analyzing found URLs and eliminating previously seen ones.

Domain URL Filter is responsible for rejecting extracted URLs if they don't belong to the allowed domain(s). Each URL belonging to allowed domain(s) (defined as a configuration parameters of this plug-in) is simply passed on to its output.

Downloader plug–it is responsible for downloading web resources which URLs appears on its input. When the resource is being downloaded, the plug–in monitors its size and downloading time as well. If they exceed limits defined in the plug–in configuration, the downloading process is terminated.

Content Filter's responsibility is making a decision about passing a given web resource on for further processing. However, this time decision is being made on the basis of the web resource content analysis. In the PoC implementation, it is made just on the basis of MIME resource header, and for further processing, there are passed on only documents of HTML, XHTML and XML type.

Store plug-in is responsible for defining a database structure and for storing crawling results as well. In the described implementation, such data as textual content of downloaded web resource, its size, MIME type, and saving timestamp is stored. To store additional (type of) information about downloaded web resource, or to store it in a different way (for instance, in file), it is enough to prepare an alternative implementation of Store plug–in and to put it into the task graph.

Link Extractor analyzes all resources appearing on its input (according to Content Filter plug–in specification there should appear only HTML, XHTML or XML documents), extracts all URLs from them, wraps them into events, and sends to its output.

7.2.2. Experiment details and obtained results

Using Cassiopeia framework and Cassiopeia Web Crawler described above, two experimental crawling tasks have been created, as shown in Table 1.

Table 1

Parameters of two experimental tasks performed with the use of CWC.

Parameter	**Experiment 1**	**Experiment 2**
The number of Cassiopeia agents	3	2
Domain for crawling	`www.agh.edu.pl`	`www.interia.pl`
Max. number of http requests generated by each agent per minute	10	10
Max. number of threads in the stage pool	5	5
Experiment duration	c.a. 24h	c.a. 24h

Both experiments have been repeated 5 times, and in Figure 9, Figure 10 and Table 2, average values from the obtained results are presented.

As one may see in Figure 9 and Figure 10, events produced during performing experimental tasks by all plug-ins defining CWC have been evenly distributed among all agents working on those tasks. The majority of events were generated by URL Normalizer and URL Extractor. Applying Seen, Domain, and Content Filter plug-ins significantly limited the number of events sent to the Downloader plug-in.

Additionally, in Table 2, the exact number of events generated by particular plug–ins during experiment 2, as well as the average number of threads in the stage (plug–in) pool, are presented just to show that the mechanisms responsible for thread–pool self–management work properly.

Just for comparison, experiment 2 has been repeated; but this time, limitation for the number of http requests per minute has been removed. The parameters of this experiment are presented in Table 3.

As before, experiment 3 was repeated 5 times, and the average from obtained results are presented in Table 4. As one may see, CWC uses the maximum number of threads for the Downloader stage. Since download speed is not very high, Downloader

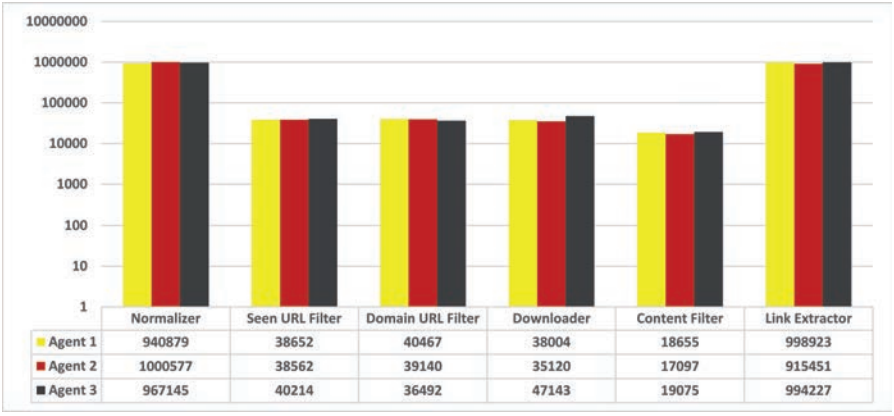

Figure 9. The number of events generated by plug-ins and agents during experiment 1 [19].

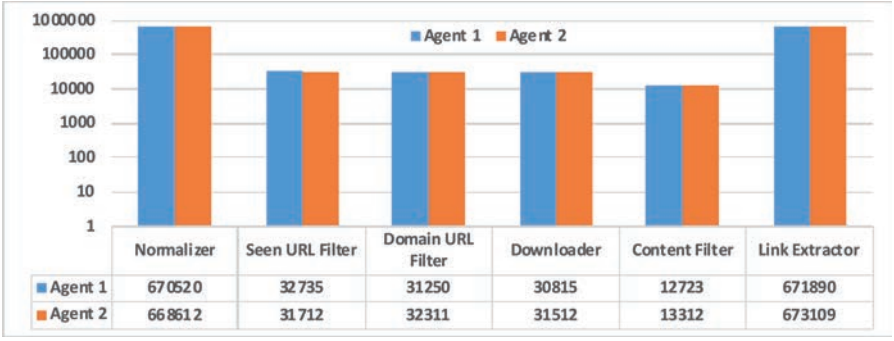

Figure 10. The number of events generated by plug-ins and agents during experiment 2 [10].

Table 2

The number of generated events and the average number of threads in stage pool in experiment 2.

Plug–in	The No. of generated events		The avg. No. of threads in the stage pool	
	Agent 1	Agent 2	Agent 1	Agent 2
Url Normalizer	670 520	668 612	2.3	2.2
Seen URL filter	32 735	31 712	1.5	1.4
Domain URL filter	31 250	32 313	1	1
Downloader	30 815	31 512	1.3	1.5
Content filter	12 723	13 312	1	1
Link extractor	671 890	673 109	1.6	1.6

Table 3
Parameters of experiment 3.

The number of Cassiopeia agents	1
Domain for crawling	www.interia.pl
Max. number of threads in stage pool	5
Experiment duration	5 minutes
Acceptable types of resources	HTML, XHTML, XML

generates a pretty low number of events for the next stage (Content Filter). So, the downloading process is a classic bottleneck, and Cassiopeia tries to improve its efficiency by assigning the maximum number of available resources, which proves once again that the resources' self–management mechanisms work properly.

It also shows one of the important advantages of the Cassiopeia framework over other crawlers; i.e., the ability for performance optimization on the level of every single task and task's stage (for instance, resource downloading). During the preliminary assessment of CWC and the Cassiopeia framework itself, also some simple comparative experiments have been performed. Results of one of them are presented in Figure 11.

During this experiment, three crawlers (i.e. CWC) described previously, as well as simple single-threaded[5] and multi-threaded Crawler4J[6] crawlers worked for ten minutes on pages in *www.agh.edu.pl* domain with the same strategy and in the same hardware and software environment. The maximum number of threads in the stage pool of CWC has been set to 5 as previously, the number of threads in Crawler4J has been managed according to its specification by JVM itself.

Table 4
The number of generated events and the average number of threads in experiment 3.

	The number of generated events	The average number of threads in stage pool
URL Normalizer	421	1,7
Seen URL Filter	240	1,2
Domain URL Filter	180	1
Downloader	178	5
Content Filter	122	1
Link Extractor	430	1,5

As before, the experiment was repeated five times, and the average number of downloaded resources, as well as the average size of downloaded data, are presented in Figure 11.

[5] `home.agh.edu.pl/~siwik/crawler/`
[6] `https://code.google.com/p/crawler4j/`

As one may see, the obtained results are pretty promising since in the given time, CWC was able to process the highest number of web resources (more than 3000) and download the most amount of data (more than 135 MBs)[7].

It is interesting that multi–threaded Crawler4J turned out to be worse than the simple single threaded crawler, which results probably from non-optimal thread management.

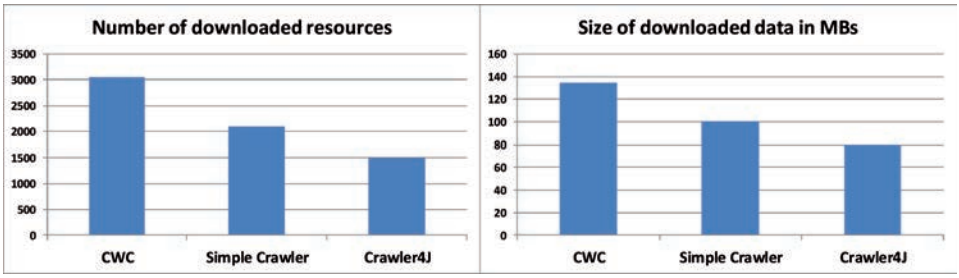

Figure 11. Results of simple comparative experiment.

The performed experiments (especially the comparative ones) were absolutely too simple to draw any far–reaching conclusions, and a lot of real–life experiments and comparisons still have to be performed. It can be said, however, on the basis of results presented in this section that:

- first of all, it is possible to design and implement a distributed, efficient, yet easy-to-use pluginable platform for (re)composing crawlers according to actual needs;
- it is possible to adjust and apply to such a platform an interesting and promising, efficient and yet self-manageable, concurrency model based on SEDA;
- the obtained results justify and encourage us to research and work on the Cassiopeia framework further.

8. Conclusions and the future work

There are many crawlers and crawling systems available on the market. Unfortunately, the majority of them are closed solutions dedicated to performing specific (kind of) tasks. At the same time, a meaningful number of analysts (such as marketing, criminal, or governmental ones) have to perform tough, vane, and very specialized crawling, searching, and analyzing tasks, and they don't need more and more dedicated crawlers, but rather an easy-to-use, all–in–one studio for composing crawlers according to actual needs, and then running, monitoring and analyzing gathered results from one single

[7]The absolute number and the absolute amount of downloaded data are not impressive since the experiments were performed on typical PCs with reduced Internet connection throughput to not be banned but more important here are relative values since all compared crawlers worked in the same environments and conditions.

place. They need such a solution, since they have to focus on doing their jobs, not dealing with tools, systems, programs, etc.

To address (among others) the above requirements, the Cassiopeia project has been started. In this paper, the idea, assumptions, top level requirements, architectural design, and finally PoC implementation are presented.

During the experiments presented in the previous section, it was confirmed that, first of all, it is possible to design and implement a framework for composing crawlers in a graphical way in accordance to specific and dynamic requirements, and then to run such crawlers in fully distributed way.

It was also confirmed that all Cassiopeia elements work properly. In particular, it was confirmed that all mechanisms responsible for tasks and events distribution work properly and effectively, since pretty even distribution was observed.

It was also confirmed that, thanks to using the SEDA architecture, it is possible to obtain really effective concurrency realization and resources' (self)management, which significantly boosts the effectiveness of the whole solution.

In the future, further experiments are going to be performed to prove that more specific and more complicated crawling tasks can be defined and run on the Cassiopeia platform which needs, among others, implementing and providing more sophisticated plug–ins.

So the next steps will be preparing an RC version of this platform, including implementation of advanced plug–ins for executing real–life crawling tasks.

References

[1] Arasu A., Cho J., Garcia-Molina H., Paepcke A., Raghavan S.: Searching the Web. *ACM Trans. Internet Technol.*, vol. 1(1), pp. 2–43, 2001. ISSN 1533-5399. http://dx.doi.org/10.1145/383034.383035.

[2] Bharat K., Broder A.: A technique for measuring the relative size and overlap of public Web search engines. *Comput. Netw. ISDN Syst.*, vol. 30(1-7), pp. 379–388, 1998. ISSN 0169-7552. http://dx.doi.org/10.1016/S0169-7552(98)00127-5.

[3] Boldi P., Codenotti B., Santini M., Vigna S.: UbiCrawler: a scalable fully distributed web crawler. *Softw. Pract. Exper.*, vol. 34(8), pp. 711–726, 2004. ISSN 0038-0644. http://dx.doi.org/10.1002/spe.587.

[4] Broder A., Kumar R., Maghoul F., Raghavan P., Rajagopalan S., Stata R., Tomkins A., Wiener J.: Graph structure in the Web. *Computer Networks*, vol. 33(1ż6), pp. 309–320, 2000. ISSN 1389-1286. http://dx.doi.org/http://dx.doi.org/10.1016/S1389-1286(00)00083-9.

[5] Cho J., Garcia-Molina H.: Parallel crawlers. In: *In Proceedings of the 11th international conference on World Wide Web*, pp. 124–135. ACM Press, 2002.

[6] Donald K., Sampaleanu C., Johnson R., Hoeller J.: *Spring framework reference documentation*.

[7] Gulli A., Signorini A.: The indexable web is more than 11.5 billion pages. In: *Special interest tracks and posters of the 14th international conference on World Wide Web*, WWW '05, pp. 902–903. ACM, New York, NY, USA, 2005. ISBN 1-59593-051-5. http://dx.doi.org/10.1145/1062745.1062789.

[8] Karger D., Lehman E., Leighton T., Panigrahy R., Levine M., Lewin D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pp. 654–663. ACM, New York, NY, USA, 1997. ISBN 0-89791-888-6. http://dx.doi.org/10.1145/258533.258660.

[9] Karger D., Sherman A., Berkheimer A., Bogstad B., Dhanidina R., Iwamoto K., Kim B., Matkins L., Yerushalmi Y.: *Web caching with consistent hashing. Computer Networks*, vol. 31(11ż16), pp. 1203–1213, 1999. ISSN 1389-1286. http://dx.doi.org/http://dx.doi.org/10.1016/S1389-1286(99)00055-9.

[10] Kluczny M.: *SEDA as an architecture for efficient, distributed and concurrent systems for web crawling purposes.* Master's thesis, Department of Computer Science, University of Science and Technology, 2012.

[11] Menczer F., Pant G., Srinivasan P., Ruiz M. E.: Evaluating topic-driven web crawlers. In: *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 241–249. ACM, 2001.

[12] Michelson B. M.: *Event-driven architecture overview. Patricia Seybold Group*, vol. 2, 2006.

[13] Pai V. S., Druschel P., Zwaenepoel W.: Flash: an efficient and portable web server. In: *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '99, pp. 15–15. USENIX Association, Berkeley, CA, USA, 1999. http://dl.acm.org/citation.cfm?id=1268708.1268723.

[14] Pariag D., Brecht T., Harji A., Buhr P., Shukla A., Cheriton D. R.: Comparing the performance of web server architectures. *ACM SIGOPS Operating Systems Review*, vol. 41(3), pp. 231–243, 2007.

[15] Shkapenyuk V., Suel T.: Design and implementation of a high-performance distributed Web crawler. In: *Data Engineering, 2002. Proceedings. 18th International Conference on*, pp. 357–368. 2002. ISSN 1063-6382. http://dx.doi.org/10.1109/ICDE.2002.994750.

[16] Singh A., Srivatsa M., Liu L., Miller T.: Apoidea: A Decentralized Peer-to-Peer Architecture for Crawling the World Wide Web. In: J. Callan, F. Crestani, M. Sanderson, eds., *Distributed Multimedia Information Retrieval*, *Lecture Notes in Computer Science*, vol. 2924, pp. 126–142. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-20875-4. http://dx.doi.org/10.1007/978-3-540-24610-7_10.

[17] Welsh M.: *A Retrospective on SEDA.* 2010.

[18] Welsh M., Culler D., Brewer E.: SEDA: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, vol. 35(5), pp. 230–243,

2001. ISSN 0163-5980. http://dx.doi.org/10.1145/502059.502057.

[19] Wlodarczyk L.: *Kassiopeia – distributed and pluginnable crawling system.* Master's thesis, Department of Computer Science, University of Science and Technology, 2011.

Affiliations

Leszek Siwik
AGH University of Science and Technology, Krakow, Poland,
http://home.agh.edu.pl/siwik, siwik@agh.edu.pl

Kamil Włodarczyk
AGH University of Science and Technology, Krakow, Poland

Mateusz Kluczny
AGH University of Science and Technology, Krakow, Poland

Received: 5.06.2013
Revised: 10.11.2013
Accepted: 15.11.2013