Paper

# Cassiopeia – Towards a Distributed and Composable Crawling Platform

Leszek Siwik, Robert Marcjan, and Kamil Włodarczyk

*AGH University of Science and Technology, Department of Computer Science, Kraków, Poland*

**Abstract—When it comes to designing and implementing crawling systems or Internet robots, it is of the utmost importance to first address efficiency and scalability issues (from a technical and architectural point of view), due to the enormous size and unimaginable structural complexity of the World Wide Web. There are, however, a significant number of users for whom flexibility and ease of execution are as important as efficiency. Running, defining, and composing Internet robots and crawlers according to dynamically-changing requirements and use-cases in the easiest possible way (e.g. in a graphical, drag & drop manner) is necessary especially for criminal analysts. The goal of this paper is to present the idea, design, crucial architectural elements, Proof-of-Concept (PoC) implementation, and preliminary experimental assessment of Cassiopeia framework, i.e. an all-in-one studio addressing both of the above-mentioned aspects.**

***Keywords—composable software, distributed Web crawling framework, event-driven architecture, event-driven processing, SEDA, Web crawler.***

## 1. Introduction

Nowadays, Internet robots, crawlers, and spiders are arguably the most-popular and most-commonly-used computer programs worldwide. In fact, www.user-agents.org claims that there were 2461 such agents in use during 2010 alone. Despite the ubiquity of such systems, identifying the best one is nearly impossible due to the specific requirements necessary for each individual use-case.

One may ask if it is possible to develop an Internet robot that can provide a framework for defining and composing robots from the ground up. A framework that can function in an efficient and scalable runtime environment by providing new building blocks to fit the needs of each user. One that is also able to adapt to varying dynamic situations while allowing the user to track task realization as well as results.

If such a toolkit were to be developed with a simple drag & drop interface, it would be a godsend for a great number of users who deal with unique and specialized search tasks. Analysts in the marketing, financial, and criminal sectors, for example, would be able to spend more time concentrating on their work and less time dealing with licensing, compatibility, and all of the other issues plaguing the solutions that are currently available.

The sheer scope and complexity of the World Wide Web [1]–[4] make the development of Internet robots and, more importantly, a framework which supports the composition of graphical robots, a herculean task. To be suitable and truly effective, the architecture of such solutions needs to be top-notch [5]–[10].

When designing a crawling system, applying the appropriate concurrency model is crucial. Each of the two classical models (thread-based and event-driven) has important shortcomings - so the question is this: are there reasonable alternatives that are able to improve crawling effectiveness while simultaneously addressing assumed flexibility and ability for composing crawlers from the building-blocks provided? In this context, Staged Event Driven Architecture (SEDA) seems to be a promising answer.

The goal of this paper is to present the idea, architecture, proof of concept implementation, and preliminary experimental assessment of the Cassiopeia framework. The authors believe this is an easy to use, all-in-one studio for (re)defining, (re)composing, and ultimately executing Internet robots in an efficient, distributed, agent-based crawling environment with the advanced concurrency model applied.

This paper is organized as follows. In Section 2 the most important top level functional and nonfunctional requirements regarding the Cassiopeia framework are defined. In Section 3, its top-level architecture as well as particular elements are presented. In Section 4 Cassiopeia agents, i.e., the most important architectural components are described. In Section 5 a typical concurrency model is discussed. Staged Event Driven Architecture, as well as its adjustment and implementation for Cassiopeia purposes are presented in Section 6. In Section 7 results of a preliminary experimental assessment of the Cassiopeia framework itself (especially, SEDA implementation) and Cassiopeia Web Crawler (CWC) are presented. Finally, in Section 8 short conclusions and future work are discussed.

## 2. Top Level Requirements

The goal of the Cassiopeia project is to design and develop a flexible and open framework for composing, defining, instantiating, launching, running, monitoring, and managing distributed crawlers as well as storing and analyzing the gathered results. Among the most important, top-level functional and non-functional requirements and assumptions, the following should be enumerated:

- it should be possible to (re)compose (also in runtime) Internet robots from available building blocks, i.e. small functionalities available on the Cassiopeia platform;

- it should be possible to redefine the composed crawlers (also in runtime) without recompiling or even restarting;
- it should be possible to extend the Cassiopeia framework by providing new building blocks not available on the platform thus far (it should be an open, not closed, framework);
- since crawling tasks (especially specialized tasks, such as those found in criminal cases) can be really demanding and long-lasting, an efficient and effective concurrency model should be applied. What is important, concurrency should be self-manageable and transparent since the end user wants to focus on logical task definitions and result analysis, not on implementation and execution details;
- taking complexity of crawling tasks into account:
  - framework should be easy-to-scale – so distributed architecture is assumed. Obviously, it should be easy to add new logical and physical computational units while redistribute running tasks among them only when needed. What is important, it shouldn't affect the effectiveness or the efficiency of the framework itself;
  - the architecture shouldn't assume any constraints regarding geographical deployment of computational units. Task execution units should be independent, and the effective model and channels of communication among them should be assured;
- the framework should be fault tolerant, so:
  - any single points of failure should be reduced or eliminated at all;
  - if some of execution units fail – realization of their tasks should be taken over by the rest of computational units. It should be done automatically without interrupting task execution;
  - running the (parts of the) framework and task execution should be possible on many different (if not all) popular hardware and software configurations.

## 3. Cassiopeia Platform Architecture

Assuming the top level functional and nonfunctional requirements defined in the previous section (and many other aspects), the following architecture of the Cassiopeia platform – presented in Fig. 1 – has been proposed. In several of the following subsections, its crucial elements are briefly discussed.

### *3.1. Communication Layer*

Providing both an effective communication channel as well as a common communication interface becomes far more necessary when the assumed distributed character of the Cassiopeia framework is taken into consideration. To address this, a dedicated communication layer has been distinguished. It provides two communication models on the framework, i.e., a point-to-point and a publish-subscribe model. The point-to-point communication model is realized between one single sender and one single recipient. On the Cassiopeia framework, a p2p communication model is used for communication:

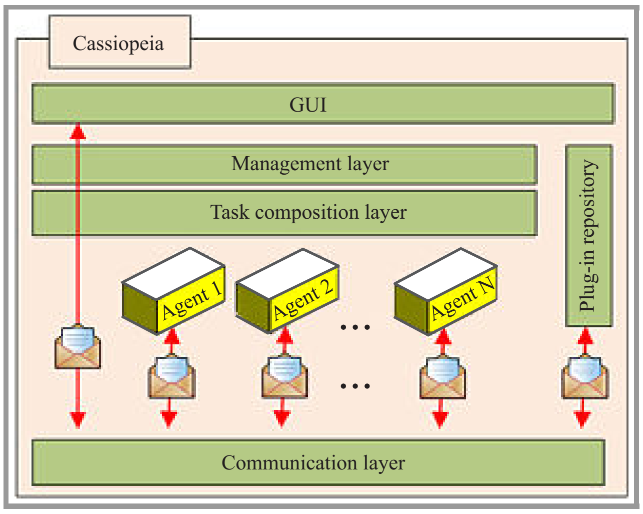


***Fig. 1.*** Cassiopeia top level architecture.

- among agents – e.g., for requesting a job to be completed;
- between GUI and agent – e.g., for stopping or pausing agent activity;
- between an agent and a plug-in repository – e.g., for downloading additional functionalities – i.e. plug-ins from the repository;
- between GUI and plug-in repository – e.g., for downloading information about available plug-ins or for submitting new plug-ins to the repository.

On the other hand, a publish-subscribe communication model is realized between one single sender and many recipients. Messages are published by the sender in the communication channel and, next, are provided to all recipients subscribed for receiving messages from this channel. On Cassiopeia, there are two communication channels of this kind:

- a general communication channel among agents and between agents and the GUI. Any agent as well as the GUI is able to publish messages on this channel as well as subscribe to receive messages from this channel;
- a heartbeat channel – described more precisely later.

From a technical point of view (as one may see in Fig. 2), the above-mentioned communication channels are implemented with the use of Java Message Service (JMS) and RESTful Web Services technologies. JMS is a part of Java Platform Enterprise Edition (J2EE), a technology which makes it possible to communicate with the use of messages. This has been chosen since is is pretty simple to realize both point-to-point as well as publish-subscribe models with the use of this technology. What is more, it provides both synchronous and asynchronous communication models and different levels of QoS. The communication infrastructure in JMS consists of several elements – most importantly, *the message*. Communication is realized between JMS clients, but not directly. The communication service provider plays the role of the mediator, providing the required level of QoS and separating particular JMS clients. The provider is responsible for implementing the JMS specification. Implementation used in the Cassiopeia framework is Apache ActiveMQ[1]. In point-to-point communication in JMS, the message sender is called *a producer* and the JMS client receiving the message is called *a consumer*. The producer puts its messages on the JMS queue with unique identifier, and the consumer takes messages from the appropriate queue whenever it wants or needs to. In the publish-subscribe model, the JMS client sending the message (this time is called a JMS publisher) puts the message on the so-called JMS topic. The main difference between the JMS queue and JMS topic is that the message put on the topic will be provided to all JMS clients registered to receive messages from this topic (they are called JMS subscribers). One example of such communication and messaging in the Cassiopeia framework is the so-called heartbeat message. Heartbeat messages are sent periodically by agents to inform other agents as well as the GUI that they are still alive. If an agent doesn't send such a messages for a period of time, it is assumed to be dead. There is a defined dedicated topic for such messages to avoid any delays and mess while providing heartbeats messages. Only agents are able to publish messages on this channel, whereas both the agents and the GUI are able to subscribe to receive messages from it (presented schematically in Fig. 3).

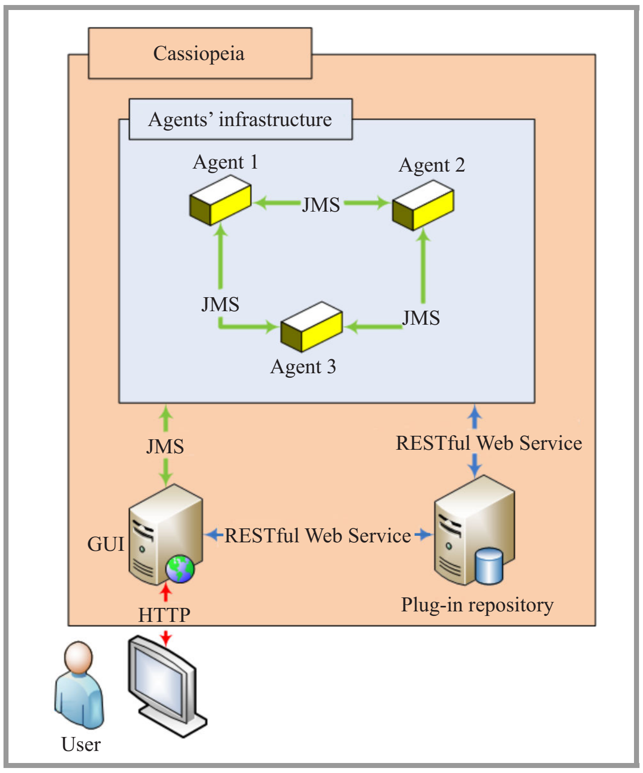


*Fig. 2.* Management layer architecture.

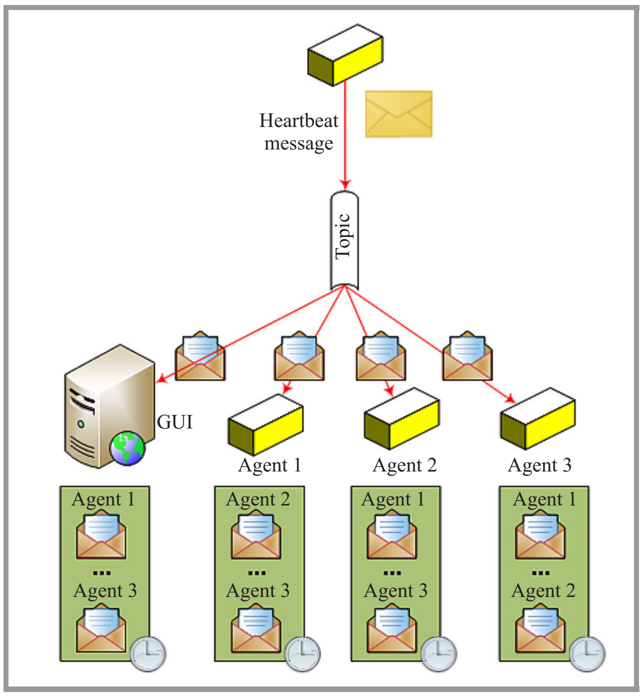


*Fig. 3.* Heartbeat messages.

In contrast to the communication realized with the use of JMS technology, communication with the plug-in repository, i.e., between agents and plug-in repository as well as between GUI and plug-in repository is realized with RESTful Web Services technology (presented in Fig. 2).

### 3.2. Task Composition Layer

Since the required functionalities and behavior depends on particular use-cases and contexts, there is no one, "ideal" crawler. This is why one of the main top level requirements regarding the Cassiopeia framework is to provide the ability for composing crawlers from predefined building blocks. This requirement is addressed by a task-composition layer consisting of two main parts responsible for task and plug-in definitions, respectively.

In the Cassiopeia framework, composing a crawling task (i.e., the crawler) consists in selecting appropriate implementation of functionalities represented by plug-ins and defining the structure of connections among them. Plug-ins

[1]http://activemq.apache.org

"react" to events appearing in their inputs, perform actions according to their definition and – if necessary – generate and send events to another plug-ins to which they are connected. This way, i.e., by selecting appropriate plug-ins and defining the structure of connections among them, almost any (crawling) task can be defined. Simple task definition consisting of three plug-ins with sample connections among them is presented in Fig. 4.

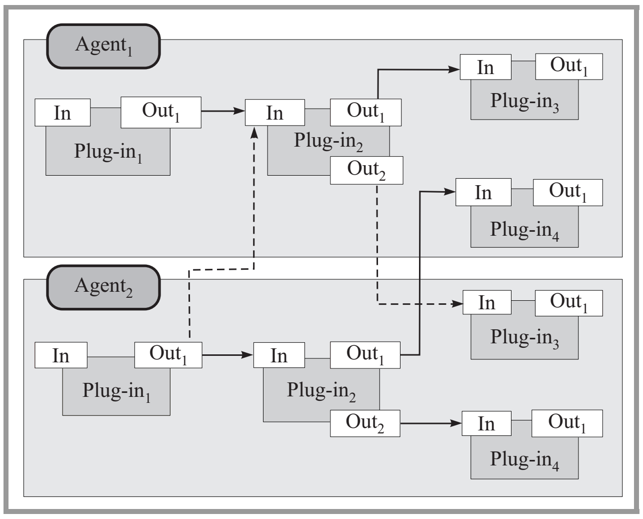


*Fig. 4.* Sample task definition and events distribution in Cassiopeia.

On the Cassiopeia platform, there exists two kinds of plug-ins:

- Processors – i.e., event-driven plug-ins. Their work consists in processing and – if necessary – generating and sending events to successive plug-ins. In this type of plug-in, there is exactly one single input where events that should be processed are passed on. With one single input, many outputs of preceding plug-ins can be linked, and this way, they can provide events that should be processed.
- Data providers – plug-ins of this type are executed exactly once during task execution. Since their role is to generate events on the basis of their starting configuration, they don't have defined inputs.

Both Data providers and Processors can define any number of outputs where generated events appear. Decisions regarding how many events should be produced and where they should be passed on depends on the plug-in implementation only. Each plug-in – to be validated as a proper Cassiopeia plug-in – has to define a plug-in descriptor allowing for its successful installation. Such a descriptor has to define at least:

– a plug-in identifier,

– information about its author and a short description,

– plug-in entry point, i.e., the fully-qualified main class' name,

– a definition and – finally – a description of configuration parameters as well as a definition and description of the plug-in's outputs.

From a technical point of view, a plug-in is a Java class compiled into JAR file and implementing defined interfaces. For instance, Data Providers have to implement *void provideData()* method whereas processors have to provide implementation of *void process(Event event)* method. To make the Cassiopeia framework "pluginable", the mechanism responsible for dynamic loading of plug-ins while the agent is working has to be provided. One considered approach was implementing the Open Services Gateway (OSGi)[2] specification. Finally, it was rejected as "too heavy" and not flexible enough, and our own implementation of a light class loader has been provided.

### 3.3. Platform Management Layer

Mentioning only the most important functionalities, the management layer allows users to:

– (re)create (crawling) tasks by selecting appropriate plug-ins and (re)defining connections among them,

– distribute tasks among agents for their execution,

– monitor agents,

– monitor repository service;

– submit new plug-ins to the plug-in repository.

From a technical point of view, it is designed and implemented as a Web application with HTML, JavaScript,

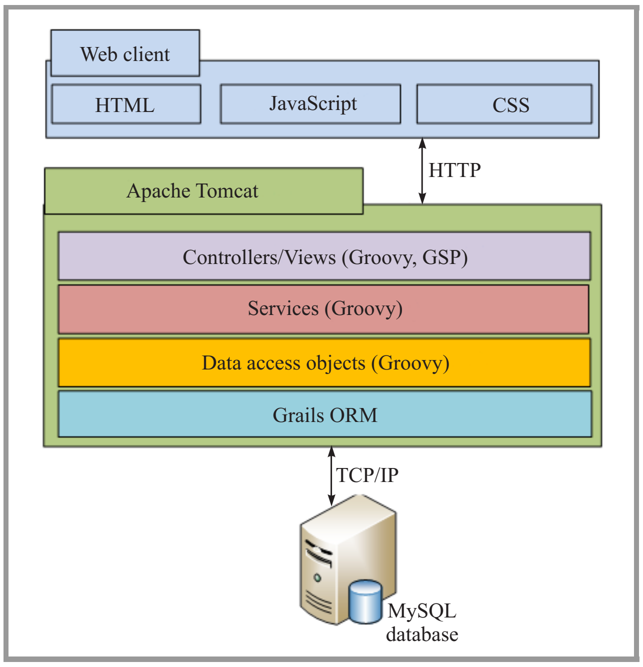
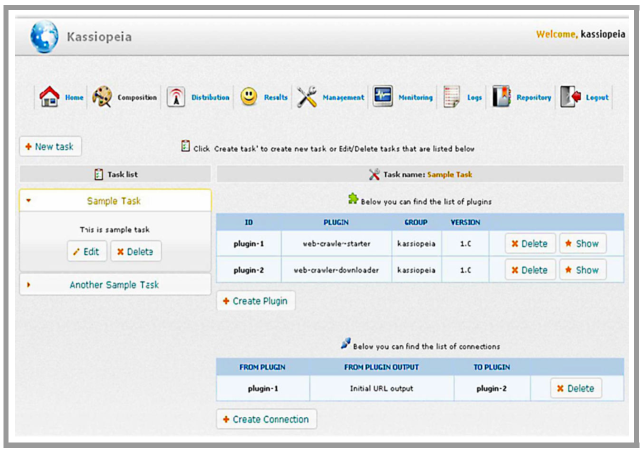


*Fig. 5.* Management layer architecture.

[2]www.osgi.org/Specifications/HomePage

and CSS on the client side, and Groovy, Groovy Server Pages (GSP)[3], and Grails application framework[4] running on Tomcat application server and mySQL as a database engine on the server side (presented in Fig. 5).
A sample GUI for task definition is presented in Fig. 6 [11]. It will be redesigned, improved, and extended in the future (with drag&drop features, for instance). At this stage, how-



Click Create task' to create new task or Edit/Delete tasks that are listed below

This is sample task

Task name: Sample Task

Below you can find the list of plugins

| ID | PLUGIN | GROUP | VERSION | |
|---|---|---|---|---|
| plugin-1 | web-cravle-starter | kassiopeia | 1.C | Delete Show |
| plugin-2 | web-crawler-downloader | kassiopeia | 1.C | Delete Show |

Below you can find the list of connections

| FROM PLUGIN | FROM PLUGIN OUTPUT | TO PLUGIN | |
|---|---|---|---|
| plugin-1 | Initial URL output | plugin-2 | Delete |

*Fig. 6.* Agent's composition screen.

ever, the idea, architectural design, and applied concurrency model, as well as a practical verification of architectural decisions, are more important than graphical design and user experience.

## 4. Cassiopeia Agents

Cassiopeia agents, as the most important and, simultaneously, the most complex elements of the Cassiopeia framework, are discussed here in a separate section. The top level agent's architecture is presented in Fig. 7. Agents are implemented as stand-alone Java applications. The components of all agents are implemented as beans, created and managed within the Spring framework with the use of the IoC container, JMX, JMS, and batch jobs mechanisms [12].
The task manager is a component responsible for task processing and execution only when it is received by the Communication Layer Adapter. Among other things, it is responsible for:

- task deserialization – task definition is saved and transmitted in the XML format, so the task manager starts its activity with task deserialization and then converts it into the graph of Java objects. It is performed with the use of XStream2 library;

- task validation – after deserialization, task manager validates plug-in connections and configurations;

[3]http://groovy.codehaus.org/
[4]http://grails.org/

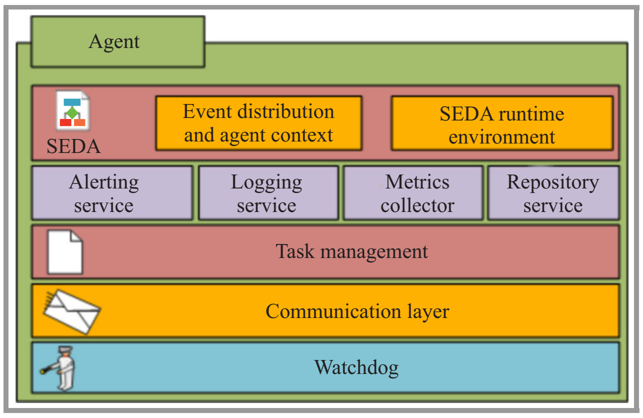


*Fig. 7.* Agent's architecture.

- task graph creation – when all instances of required plug-ins are created – a graph defining a particular task is created. Such graph is passed on to the SEDA runtime environment component.

The plug-in manager is responsible for downloading plug-ins from the plug-in repository as necessary, creating their instances and passing them on to the task manager.
To create a plug-in instance, a JAR file with plug-in implementation has to be localized, and then all required classes have to be loaded. A JAR file with plug-in implementation can be loaded from the local repository or if it cannot be found there downloaded from the remote plug-in repository. The local repository in current Cassiopeia implementation is realized as a regular directory located in the agent's directory tree with a convenient API provided by the Repository service.
SEDA runtime environment is Cassiopeia's proprietary implementation of the SEDA specification [13], [14], and it's responsible for task execution in high-concurrency conditions. SEDA, as an architecture specification for high-concurrency systems, mixes thread-based and event-driven concurrency models. Since it is a crucial part of the Cassiopeia architecture, it is discussed in further detail in Section 6.
To ensure an even and a dynamic workload distribution – events appearing on plug-in's outputs are handled by the Event Distribution Service (EDS). Each event handled by EDS is processed with the use of a Consistent Hashing Algorithm (CHA) [15], [16], i.e. a task- and event-distribution algorithm. On the basis of the value of hashing function returned by CHA, EDS makes a decision regarding distribution of events among active agents. What is important, CHA and Cassiopeia architecture make it possible to identify the agent responsible for handling a particular event locally, i.e. without any additional communication among agents and without any central or global decision makers. This is possible, since each agent keeps and updates (on the basis of received heartbeat messages) a set of known alive and active agents. What is more, CHA and Cassiopeia micro and macro architecture as well, allows, this way, for tasks and events to be redistributed when

some agents "die" and are no longer available. And again, this can be done locally without any additional communication among agents and without any central decision-making components. Sample event distribution between two agents working on the same task is presented in Fig. 4. When plug-in$_1$ of agent Agent$_2$ sends the event on its OUT$_1$ output, it can be passed on both – the input of plug-in$_2$ of the same agent or the input of plug-in$_2$ of another agent. Which situation, i.e., to which agent in fact this event as well as its processing will be distributed depends on the result of applying of the distribution algorithm (CHA).

Logging service makes it possible to save agent logs in a local file system and send them to the framework management layer (GUI) if necessary. This way, the end user is able to follow and track the activity and behavior of all agents from one single place with the use of a user-friendly and convenient GUI. From a technical point of view, Logging service is a far extension of the Log4j library.

The task and the responsibility of the Alerting service component is to report critical failures. In current implementation, a simple email notification is sent to the framework's administrator in such case.

The main responsibility of the Metrics collector component is to collect some statistics and parameters about an agent's work and activity, which can be used for monitoring and diagnosing the Cassiopeia environment. Additionally, it can be used for calculating some automatic measures and metrics, providing synthetic information about Cassiopeia's actual state and efficiency. The component is designed and implemented in such way that adding new parameters, statistics, or measures that should be calculated is easy and straightforward.

A Watchdog component is responsible for broadcasting heartbeat messages. As previously mentioned, such messages are sent by agents periodically to inform other agents and the framework itself (GUI) that it is still alive. If an agent doesn't send such messages for some time, it is assumed to be dead. The decision if such message should be sent or not, which means that the agent is working normally and is performing its own tasks or, conversely, that something has failed is made by the Watchdog component. Heartbeat messages indirectly inform other agents and the framework itself about the actual state of the Cassiopeia infrastructure. In fact, they include such information as agent identifier, agent JVM state (at the moment the message was sent), the identifier of a task on which the agent is actually working, etc. The watchdog component is also responsible for handling heartbeat messages coming from other agents. On the basis of received messages, watchdog keeps and updates information about the set of known agents that are alive and active. This information is used by the EDS in CHA while a decision about event distribution is being made.

As previously mentioned, a Communication Layer is distinguished on a macro (i.e. framework) level. It is responsible for providing communication channels among parts and components from different frameworks. On a micro (i.e., agent) level, Communication Layer Adapter (CLA) makes it possible to access the communication services provided by the framework's communication layer. CLA is responsible for any aspects of the agent's communication, i.e. for communication with other agents and within the framework itself (with GUI in particular) as well.

The agent's code is instrumented with the use of JMX technology. It allows not only for a convenient monitoring of agent activity and state, but also makes it possible to change the agent's configuration parameters in runtime. With nearly every component distinguished in the agent's microarchitecture, there is an appropriate Managed Java object (MBean) associated so it is possible to change its configuration parameters to influence agent behavior.

## 5. Concurrency Models

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

### *5.1. 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 the "one request – one thread/process" model. Such model is supported by both contemporary operating systems as well as programming languages and environments. In such an approach, the operating system switches the processor among threads/processes evenly – what is very convenient from a developer and architect point of view. The efficiency of such system significantly falls, however, when the number of threads/processes increases. To avoid such situation, some systems define a limit regarding the number of threads/processes that can be simultaneously created and processed. When the top limit is reached, new requests are simply not accepted. Such an approach allows to avoid the efficiency problem. However, it increases latency (also undesirable, of course). It is a pretty popular approach, and it is implemented, for instance, by Apache Web server or Tomcat application server. However, it is not appropriate for systems with massive concurrency, such as crawling systems.

### *5.2. Event-Driven Concurrency*

Limitations and problems with allocating an uncontrolled pool of threads are reasons why developers and architects give up such an approach and use an event-driven concurrency [14]. In such model, there is a relative 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 consecutive stages are triggered by events. In contrast to the previously-described approach, the application itself is able to control how the given task/request is being processed.

Applying such model of concurrency allows to avoid the problems discussed earlier by reducing the system efficiency when the number of threads increases. Now, when the number of requests grows, application throughput increases too – until the top limit is reached. In such case, any further requests are scheduled in the input queue and are processed only when the required resources become available again. The application throughput stays constant even during workload peaks, and latency grows linearly – not exponentially, as in the previous case. There is, however, a strong assumption that non-blocking implementation of the event processing unit is provided what is usually difficult to achieve and has to be ensured by the application itself.

One of the issues related to the event-driven concurrency model is that the application itself has to care for event scheduling and queuing. The application has to make decisions, for instance, on when to start processing incoming events and how they should be scheduled and ordered. What is more, it has to keep the service level balanced and minimize latency. That is why scheduling, dispatching, and prioritizing algorithms are crucial for system efficiency. It is usually implemented and adjusted individually for a given application and its use-cases. A few problems result, including extending the application with new functionalities, since dispatching algorithm and concurrency management mechanisms likely have to be replaced. Flash Web Server, with its Asymmetric Multi-Process Event Driven (AMPED) architecture, is an example of a server based on such model of concurrency [17].

None of the typical concurrency management models noted above are ideal approaches. That is why research on alternative models is still needed to propose efficient and convenient architecture for concurrent and distributed applications. One of the most-promising models is a Staged Event-Driven Architecture – SEDA [13], mixing to some extent both approaches previously discussed as well as providing some additional, interesting, and important (for crawling and the crawler composition platform) benefits, such as splitting the application into separate stages connected by event queues.

## 6. SEDA Implementation for Cassiopeia Purposes

SEDA was proposed in 2000 at the University of California by Matt Welsh *et al.* [13]. SEDA mixes both approaches discussed in the previous section, i.e., event-driven and thread-based concurrency. It provides task-scheduling mechanisms and makes it possible to manage task execution parameters during the runtime. This also makes it possible to reconfigure the application automatically depending on its workload. SEDA consists of a network of nodes called stages. With each stage, there is one associated input-event queue. Each stage is an independent module managed individually, depending on input queue parameters. The possibility of monitoring its input queue by each node/stage makes it possible to filter and prioritize events, and to resize and manage the pool of threads it uses appropriately to the actual situation, number of events to be processed, and the general workload. In the consequence, the SEDA-based application becomes very flexible since, on the one hand, it is workload-resistant and, on the other, doesn't consume resources if it is unnecessary [13].

There are, of course, some limitations regarding SEDA-based application efficiency [18], and even the author of this specification has some remarks and thoughts – both positive and negative – about this architecture [19].

Introducing stages with the structure of connections makes it really easy and natural to decompose the application into separate and easy-to-replace modules. Although there are some open-source and enterprise SEDA implementations for Cassiopeia-platform purposes, it has to meet some additional needs and requirements. That is why the proposed implementation of the SEDA specification has been developed [20]. Mentioned in Fig. 7, the SEDA run-time environment is an implementation of SEDA architecture working within the one, single JVM. Besides crucial SEDA elements such as stages and queues, some additional components have also been implemented, such as monitoring, notification, and events-distribution mechanisms.

Generally speaking, the SEDA runtime environment is responsible for configuring a given plug-in, allocating all required resources, launching and running the application, monitoring all application runtime parameters, releasing unnecessary resources, and ultimately stopping the application.

The task configuration layer is responsible for providing task configuration (read from an XML file) as well as creating task stages along with their controllers and connections. After that, it returns the instance of a *ConfiguredTask* class.

There is a dedicated component responsible for calculating and collecting statistics regarding SEDA runtime. The implementation presented has been equipped also with the event-notification mechanism.

Stage is a separate application module which meets the SEDA stage specification. Each stage developed for Cassiopeia purposes consists of a plug-in, a managing module, a input queue, a controller, and a thread pool. Plug-in is an event handler defined by SEDA specification, and it defines business logic realized by the particular stage. The thread pool is responsible for executing business logic defined in the plug-in, and the controller monitors the size of the input queue and resizes the pool of threads as necessary.

Input queue is the event (input data) source for stage as well as the means of communication between stages.
Above, only a glance at the SEDA design and implementation for Cassiopeia purposes is given since discussing it in detail is outside the scope of this paper. More detailed discussion is presented in [21].

## 7. Preliminary Experimental Verification

To assess the correctness and usability of the Cassiopeia framework, a simple Cassiopeia Web Crawler (CWC) has been defined, composed, and run.
CWC structure consists of the following plug-ins: Seed, URL normalizer, Seen URL filter, Domain URL filter, Downloader, Content filter, Store, and URL extractor.
Seed is a plug-in of DataProvider type, so SEDA Runtime Environment launches it only once, at the beginning of the task's execution. As a configuration parameter, it takes the initial (starting) URL address. As an output, it returns URL addresses that should be processed. URL normalizer takes the URL address which appears in its input and returns its normalized version in its output. Thanks to the introduction of this plug-in into the CWC definition, such URL identifiers as http://example.com http://example.com/ and http://example.com:80 are recognized as the same, single URL. During crawling task execution, it is possible that the same URL identifier will be found many times, and consequently, it would be many-times analyzed, downloaded, etc. To avoid such a situation, the CWC definition consists – among others – of a 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 URL's if they don't belong to the allowed domains. Each URL belonging to the allowed domains, defined as a configuration parameters of this plug-in is simply passed on to its output.
Downloader plug-in is responsible for downloading Web resources from URL's which appear in its input. When the resource is being downloaded, the plug-in monitors its size and download 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 to further processing units – but this time decision is made on the basis of Web resource content analysis. In current implementation, it is made just on the basis of the MIME resource header, and for further processing, only documents of HTML, XHTML and XML types are passed on.
Store plug-in is responsible for defining the database structure and for storing crawling results as well. In the described implementation, data such as textual content of downloaded web resources, their size, MIME type, and saving time-stamp is stored. To store additional information about downloaded Web resources, or to store it in a different way, i.e. in a file system, it is enough to prepare an alternative implementation of the Store plug-in and put it into the task graph. Link extractor analyzes all resources appearing in its input (according to Content filter plug-in specification, only HTML, XHTML or XML documents should appear), extracts all URLs from them, wraps them into events, and sends to its output.
The part of XML file with Cassiopeia Web Crawler definition is presented in Listing 1.

Listing 1. The part of Cassiopeia Web Crawler XML definition

```
<?xml version="1.0" encoding="utf-8"?>
<task>
   <name>Cassiopeia Web Crawler</name>
   <description>
      This task is a simple web crawler
      implementation for Cassiopeia platform
   </description>

   <!--PLUGINS-->
   <plugin>
    <instanceId>url-seed</instanceId>
     <coordinates>
       <pluginId>url-seed</pluginId>
       <groupId>Cassiopeia</groupId>
       <version>1.0</version>
     </coordinates>
     <parameter>
       <name>initial-url</name>
       <value>http://www.agh.edu.pl</value>
     </parameter>
   </plugin>
   ...
   <!--CONNECTIONS-->
   <connection>
     <fromPlugin>url-seed</fromPlugin>
     <fromPluginOutput>out</fromPluginOutput>
     <toPlugin>url-normalizer</toPlugin>
   </connection>
   ...
</task>
```

The most important parameters of three simple crawling experiments performed with the use of Cassiopeia Web Crawler are as follows:

- Experiment 1:
  - Domain: www.agh.edu.pl,
  - Number of Cassiopeia agents: 3,
  - Max. number of requests per agent per minute: 10,
  - Max. number of threads in the stage pool: 5,
  - Experiment duration: 24 hrs.
- Experiment 2:
  - Domain: www.interia.pl,
  - Number of Cassiopeia agents: 2,
  - Max. number of requests per agent per minute: 10,
  - Max. number of threads in the stage pool: 5,
  - Experiment duration: 24 hrs.

- Experiment 3:
  - Domain: www.interia.pl,
  - Number of Cassiopeia agents: 1,
  - Max. number of requests per agent per minute: ∞,
  - Max. number of threads in the stage pool: 5,
  - Experiment duration: 5 min.

All experiments have been repeated 5 times and in appropriate figures and tables the average values from obtained results are presented.
Taking the top level requirements and main architectural assumptions and decisions into account, preliminary experimental verification of Cassiopeia framework should assess its two crucial aspects, i.e. SEDA implementation and concurrency model applied as well as the ability to compose crawlers from provided building blocks.

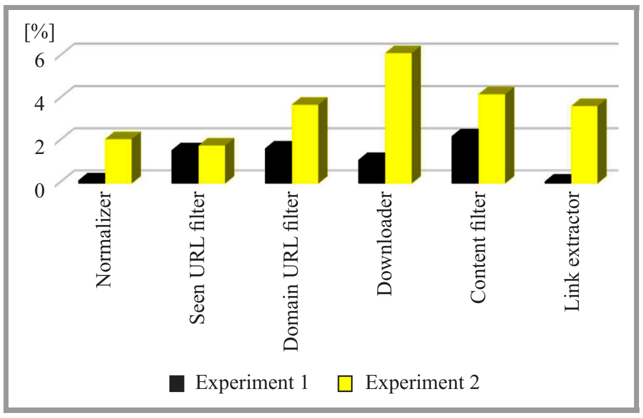


*Fig. 8.* Differences in the number of events performed by agents working on the same task.

Looking for the answer if SEDA implementation work properly and is applied concurrency model a proper one, a normalized, average difference in the number of events processed by a particular agents' plug-ins during the experiments is presented in Fig. 8. As one may see, the proposed architecture and SEDA implementation seem to work properly and efficiently, since pretty-even event distribution among all agents working on a particular task can be observed. Generally speaking, the measured difference in the number of events processed by agents working on the same task was not higher than 6% during the performed experiments. It is a really good result in the first implementation. This proves beyond a doubt that the architectural decision to apply SEDA as a concurrency model, as well as its implementation was an absolutely proper and appropriate decision. A sample event distribution, in the case of plug-in processing of almost or slightly more than a million events, is presented in Fig. 9.

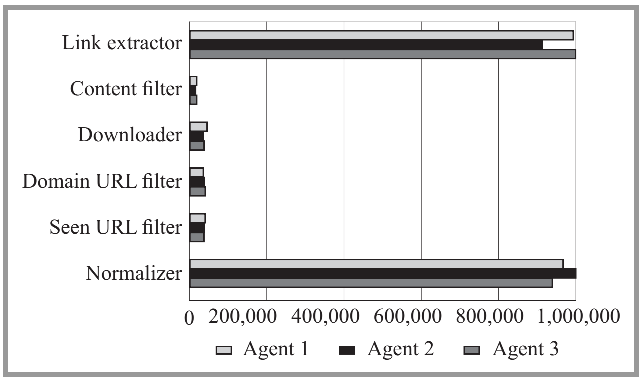


*Fig. 9.* Distribution of events among agents working on the same task during experiment 1.

In Fig. 10, the average number of threads allocated by each plug-in during experiments 1 and 3 respectively is presented. As one may see, even event distribution was not occupied with extensive allocation of system resources since the average number of threads during both experiments oscillates around 2. What is really promising, CWC uses the maximum number of threads for Downloader stage when it is necessary. Since download speed was not very high, Downloader plug-in is generating a pretty low number of events for the next stage (Content filter). So, the downloading process is a classical bottle neck and Cassiopeia tries to improve its efficiency by assigning the maximum number of available resources. This proves once again that self-management mechanisms work properly. It shows also one of the important advantage of Cassiopeia over the other crawlers, i.e. the ability to optimize performance on the level of every single task and task stage (resource downloading).

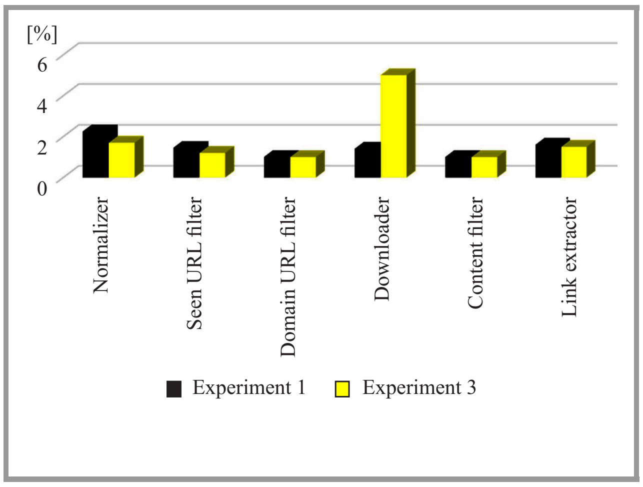


*Fig. 10.* The average number of threads allocated by particular plug-ins.

During preliminary assessment of CWC and the Cassiopeia framework itself as a framework for running crawlers, some simple comparative experiments against other crawlers were also performed. The results of one of these experiments are presented in Table 1. During this experiment, three crawlers, i.e., CWC described previously, as well as simple single-threaded [22] and multi-threaded Crawler4J crawlers were working for ten minutes on pages in www.agh.edu.pl domain with the same strategy and in the same hardware and software environments. The maximum number of threads in the stage pool for CWC had been set to 5 as previously, the number of threads in Crawler4J was managed by JVM itself according to its specification.

Table 1
Results of the simple comparative study

| | No. of resources | Size of data [MB] |
|---|---|---|
| CWC | ∼3050 | ∼135 |
| Simple crawler | ∼2050 | ∼100 |
| Crawler4J | ∼1500 | ∼80 |

As previously, 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 the table. As one may see, the results obtained are pretty promising, since CWC was able to process the highest number of Web resources (more than 3000) and to download the most amount of data (more than 135 MB) in the allotted time. It is interesting that multi-threaded Crawler4J turned out to be worse than a simple single-threaded crawler, likely due to non-optimal thread management.

Performed experiments (especially comparative ones) have been 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 efficient and yet self-manageable concurrency model based on SEDA specification;
- the results obtained justify and encourage further research and work on the Cassiopeia framework.

## 8. Conclusions and Future Work

Today, there are many crawlers and crawling systems available to Internet users – unfortunately, the majority of them are closed solutions limited to performing specific tasks. These limitations affect many individuals who must perform very tough and specialized crawling, searching, and analyzing tasks as a part of their work. Marketing, criminal, and governmental analysts are among those who would benefit greatly from an easy-to-use, all-in-one studio – one dedicated to composing crawlers that fit each individual's specific needs. A studio which can run, monitor, and analyze search results in one integrated package that doesn't require a lot of time-consuming maintenance. In an attempt to fill this void, the Cassiopeia project has been initiated.

This paper presents the idea, assumptions, top-level requirements, architectural design, and proof-of-concept implementation of the Cassiopeia project. During the experiments presented in the previous section, findings confirmed that it is possible to design and implement a framework for composing crawlers in a graphical way and, subsequently, run such crawlers in a fully-distributed manner. It was also confirmed that all of Cassiopeia's elements work together in harmony. In particular, it was shown that all of the task and event distribution mechanisms function properly and effectively, as demonstrated by the fairly-equal distribution among the working agents. And thanks to the SEDA architecture, it is possible to obtain truly-effective concurrency realization and resource management that significantly boosts the effectiveness of the whole solution.

In the future, further experiments will be performed to prove that more specific and complicated crawling tasks can be defined and run on the Cassiopeia platform. Among other things, more-sophisticated plug-ins will be introduced and implemented in an attempt to further examine Cassiopeia's effectiveness. So, the next step will be to prepare a release-candidate version of this platform, which will include advanced plug-ins intended to execute real-life crawling tasks.

## References

[1] F. Maghoul *et al.*, "Graph structure in the Web", in *Proc. 9th Int. World Wide Web Conf.*, Amsterdam, The Netherlands, 2000, pp. 309–320.

[2] H. Garcia-Molina, A. Paepcke, A. Arasu, J. Cho, and S. Raghavan, "Searching the Web", *ACM Trans. Internet Technol.*, vol. 1, no. 1, pp. 2–43, 2001.

[3] A. Gulli and A. Signorini, "The indexable Web is more than 11.5 billion page", in *Proc. 14th Int. World Wide Web Conf.*, Chiba, Japan, 2005, pp. 902–903.

[4] K. Bharat and A. Broder, "A technique for measuring the relative size and overlap of public search engines", in*Proc. 7th Int. World Wide Web Conf.*, Brisbane, Australia, 1998, pp. 379–388.

[5] A. Singh, M. Srivatsa, L. Liu, and T. Miller, "Apoidea: A decentralized peer-to-peer architecture for crawling the World-Wide-Web", in *Proc. SIGIR Worksh. Distrib. Inform. Retrieval*, Toronto, Canada, 2003.

[6] J. Cho and H. Garcia-Molina, "Parallel crawlers", in *Proc. 11th Int. World Wide Web Conf.*, Honolulu, Hawaii, 2002, pp. 124–135.

[7] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler", *Software: Practice and Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[8] V. Shkapenyuk and T. Suel, "Design and implementation of a high performance distributed Web crawler", in *Proc. 18th IEEE Int. Conf. Data Engin.*, San Jose, CA, USA, 2002.

[9] M. Santini, P. Boldi, B. Codenotti, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler", in *Proc. 8th Australian World Wide Web Conf.*, Sushine Coast, Queensland, Australia, 2002.

[10] F. Menczer, G. Pant, P. Srinivasan, and M. E. Ruiz, "Evaluating topic-driven Web-crawlers", in *Proc. 24th Ann. Int. Conf. Res. Develop. Inform. Retriev.*, New York, USA, 2001, pp. 241–249.

[11] K. Wlodarczyk, "Kassiopeia – distributed and pluginnable crawling system", Master thesis, Department of Computer Science, University of Science and Technology, Kraków, 2011.

[12] K. Donald, C. Sampaleanu, R. Johnson, and J. Hoeller, "Spring framework reference documentation" [Online]. Available: http://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/

[13] M. Welsh, D. Culler, E. Brewer, and E. Gribble, "SEDA: An architecture for Well-Conditioned scalable internet services", Harvard University, 2001 [Online]. Available: http://www.eecs.harvard.edu/∼mdw/papers/seda-sosp01.pdf

[14] B. M. Michelson, "Event-Driven Architecture Overview", Patricia Seybold Group, Boston, USA, 2006 [Online]. Available: http://www.omg.org/soa/Uploaded%20Docs/EDA/bda2-2-06cc.pdf

[15] D. Lewin, D. Karger, T. Leighton, and A. Sherman, "Web caching with consistent hashing", in *Proc. 8th Int. World Wide Web Conf.*, Toronto, Canada, 1999.

[16] D. Lewin, M. Lehman, D. Karger, T. Leighton, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web", in *Proc. 8th Int. World Wide Web Conf.*, Toronto, Canada, 1999.

[17] V. S. Pai, P. Druschel, and W. Zwaenpoel, "Flash: An Efficient and Portable Web Server", Ann. Tech. Conf., Monterey, CA, USA, 1999 [Online]. Available: http://static.usenix.org/event/usenix99/full_papers/pai/pai.pdf

[18] D. Pariag *et al.*, "Comparing the performance of Web server architectures", in *Proc. 2nd ACM SIGOPS/EuroSys European Conf. Comp. Sys.*, Lisbon, Portugal, 2007, pp. 231–243.

[19] M. Welsh, "A Retrospective on SEDA", 2010 [Online]. Available: http://matt-welsh.blogspot.com/2010/07/retrospective-on-seda.html

[20] M. Kluczny, "SEDA as an architecture for efficient, distributed and concurrent systems for Web crawling purposes", Master thesis, Department of Computer Science, University of Science and Technology, Kraków, 2012.

[21] L. Siwik, K. Wlodarczyk, and M. Kluczny, "Staged event-driven architecture as a micro-architecture of distributed and pluginable crawling platform", *Comp. Science*, vol. 14, no. 4, pp. 645–665, 2013.

[22] Cassiopeia Web Crawler [Online]. Available: http://home.agh.edu.pl/siwik/crawler/

**Leszek Siwik** has graduated with honors from Computer Science at the AGH-UST University of Science and Technology in 2002, next he has graduated from Department of Management at the AGH-UST in 2004. He works as an Assistant Professor at the Department of Computer Science of AGH-UST where in 2009 he obtained his Ph.D. with honors in Computer Science in artificial intelligence area. His research focuses on multi-agent systems in multi-objective optimization, security and cryptography and mobile systems.
E-mail: siwik@agh.edu.pl
AGH University of Science and Technology
Department of Computer Science
Mickiewicza Av. 30
30-059 Kraków, Poland

**Robert Marcjan** obtained his M.Sc. in Computer Science in 1990 at the AGH University of Science and Technology in Kraków. He works as an Assistant Professor at the Department of Computer Science, AGH-UST where in 2000 he obtained his Ph.D. with honors in Computer Science in the area of artificial intelligence and multi-agent systems. His research focuses on AI, multi-agent systems, expert systems and databases.
E-mail: marcjan@agh.edu.pl
AGH University of Science and Technology
Department of Computer Science
Mickiewicza Av. 30
30-059 Kraków, Poland

**Kamil Włodarczyk** obtained his M.Sc. in Computer Science in 2011 at AGH University of Science and Technology in Kraków. He now works on high performance trading platform used at the heart of one of the world's leading investment banks. His areas of interests include distributed systems and concurrent programming.
E-mail: kawlodar@gmail.com