Michał Ślaski
Wojciech Turek
Arkadiusz Gil
Bartosz Szafran
Mateusz Paciorek
Aleksander Byrski

ANALYSIS OF DISTRIBUTED SYSTEMS DYNAMICS WITH ERLANG PERFORMANCE LAB

Abstract *Modern, highly concurrent, and large-scale systems require new methods for design, testing, and monitoring. Their dynamics and scale require real-time tools that provide a holistic view of the whole system and the ability to show a more detailed view when needed. Such tools can help identify the causes of unwanted states, which is hardly possible with a static analysis or metrics-based approach. In this paper, a new tool for the analysis of distributed systems in Erlang is presented. It provides the real-time monitoring of system dynamics on different levels of abstraction. The tool has been used for analyzing a large-scale urban traffic simulation system running on a cluster of 20 computing nodes.*

Keywords software engineering, distributed computing and simulation, distributed systems monitoring

Citation Computer Science 19(2) 2018: 139–155

1. Introduction

The fast development of Internet-scale applications that have been observed over the past few years brings new challenges for software engineers regarding issues that are difficult to solve using traditional methods. Massively concurrent systems executed in parallel using clusters of computers or cloud environments form new great challenges in terms of maintainability, robustness, and effectiveness [16]. The complexity and dynamics caused by parallel execution make it hard to analyze and understand all of the phenomena and interactions. This can result in the emergence of issues that are difficult to detect, reproduce, and eliminate.

Existing technologies and tools allow us to build and deploy systems that are able to handle millions of simultaneous users. However, the technologies and tools lack methodologies for guaranteeing that the created systems are robust and reliable, so the development process must consider these problems (cf. e.g., [6]). These missing methodologies are related to various steps in the system development process, starting at the design, continuing through the implementation, debugging, and testing, and ending at the monitoring and maintaining.

Well-recognized software engineering methods from recent decades (like the object-oriented approach) have developed standards for the analysis design, documentation, development, and testing of software. They even include design patterns for solving typical problems that have been common for several different technologies and programming languages. The existence of the methodologies have allowed for unification, simplifying the process of development, making the software quality controllable, and allowing the maintainability of existing systems [38].

However, the above-mentioned methodologies are hardly suitable for modern, highly-concurrent, large-scale systems. The domain needs new methodologies, which will probably emerge as a generalization of different approaches, tools, and technologies created now for solving particular problems [14]. One of the visible trends in the development of modern systems is the decomposition of processing into small asynchronous units. This allows us to create systems that possess strongly desired features: acceptance of partial failures and utilization of modern highly parallel hardware. This general approach is used by micro-service-based design and actor-based systems where loosely coupled fine-grained units cooperate, typically using asynchronous communication. Several attempts for defining more general methods of building such systems have already been proposed; for example, the supervision trees in Erlang/OTP [3] or Akka [1], which allow for explicitly defining the structure of the dependencies between the elements of a system.

The explicit modeling of the static structure is highly insufficient in ensuring the high quality of the software. A system composed of thousands of different loosely-coupled elements that operate for long periods of time can reach states that cannot be captured by static analysis. The domain of distributed system dynamics modeling and analysis seems to be one of the areas where basic methods have yet to be developed. In particular, this concerns the area of monitoring and visualizing system behavior

in order to detect performance problems, locate the cause of any anomalies, or just gain insight into how the system responds to particular stimuli. Traditional methods based on monitoring dashboards or the visualization of system traces do not scale well for massively concurrent systems and fail to provide the required understanding of how a system behaves. Consequently, new approaches in the area of system testing and monitoring are emerging, such as those pioneered in Netflix (dubbed Chaos Engineering [4]) and Intuition Engineering [31]. At the heart of these new methods is the premise that, in order to properly understand the operation of a massively concurrent system, we need a holistic view of its state available in real-time so that we can observe system dynamics in particular, how it responds to changing conditions such as failures. An offline analysis of system monitoring data or the real-time visualization of individual system parts are no longer sufficient.

In this paper, we present a visualization tool for the analysis of the dynamics of systems written in Erlang. This tool, called the Erlang Performance Lab (ErlangPL), allows for the real-time monitoring of a distributed system (application agnostic), integration of various metrics, and real-time presentation of the collected data using a set of modern visualization techniques. In its current state, the tool focuses on measuring the communication intensity between the processes and nodes. It certainly does not solve all of the problems related to understanding distributed system dynamics; however, we believe that it is an important step towards defining useful methods in this area.

The presented tool has been tested using a distributed simulation system that was deployed on a cluster of computers. We have a supercomputer available and ready to be tested; however, the tool requires an external connection to a web browser, which is impossible in the supercomputing environment we typically use (the Prometheus cluster of the Academic Computing Center Cyfronet AGH). Therefore, we had to limit ourselves to testing a cluster of 20 computers in order to present the tangible results in this paper.

In this paper, a distributed system monitoring tool is presented, applied to visualization of the events happening in a distributed traffic simulation system. In particular, efficiency-related issues are visualized, based on a test-bed of 20-computer cluster. Various conclusions from the experiments are presented in the paper, together with several new ideas for the further development of the tool that will be implemented in the future.

2. Monitoring of distributed systems

Performance monitoring and visualization is one of the main methods of gaining insight into the behavior of parallel and distributed systems [19]. Typical visualization techniques include the presentation of system event traces on Gantt charts, graphs (showing topology), or more advanced charts such as treemaps. These techniques become ineffective for large-scale systems, so data reduction techniques (e.g., aggregation or clustering) are applied in order to improve the scalability of the visualiza-

tion method. However, such data reduction may easily lead to information loss [22]. Schnorr et al. (2012) proposed an approach based on the hierarchical aggregation of monitoring data and treemap diagrams to achieve a scalable performance visualization of parallel applications consisting of thousands of processes [33]. This approach is, however, best-suited for visualizing the status of individual processes (but not the traffic between them).

In-memory computing platforms such as Hazelcast [15], Infinispan [18], or Apache Ignite [2] support advanced distributed computing capabilities. Hazelcast collects many standard monitoring metrics such as CPU and memory utilization as well as read/write throughputs for various objects of the system (nodes and data structures). The metrics can be visualized in real time, typically on time-series plots. Infinispan and Ignite expose the monitoring metrics through RMX, but no visualization dashboards are provided by default.

The actor-based systems implemented with frameworks like Akka or Erlang/OTP have a specific model of computation based on asynchronous messages and lock--free concurrency. Consequently, monitoring such systems must be based on specific actor-centric metrics focusing on actor utilization and communication between the actors [32]. The existing monitoring tools for Akka include Lightbend monitoring [24] and Kamon [20]. These tools focus on mailbox metrics (size and message waiting time) and message-processing time. However, actor utilization or message exchange metrics are not supported. A set of tools for the operations and maintenance of Erlang clusters is available in WombatOAM [39] (e.g., visualization of system topology, charts with various metrics, dashboard with notifications, and alarms).

Intuition Engineering [31], a term coined by Netflix, is an approach wherein a holistic state of a system is visualized in real-time. So far, Netflix has published the Vizceral tool [37] that visualizes traffic between system nodes at different levels of detail (global-, regional-, and service-level). The main idea of this approach is to first provide a tool enabling one to understand the "correct" system behavior and its anomalies, and second to quickly locate the potential sources of a problem in case a failure or performance issues occur. The latter feature is essential for quick problem diagnosis because, even if the system collects dozens of metrics, it gives no hint as to where to first look for the problem. Originally designed for Netflix, the approach has also been considered useful for smaller-scale systems [17].

Percept2 [23] and RefactorErl [35] are tools that approach analysis via the use of static data. Percept2 allows the user to visualize and profile parallel Erlang applications. It is based on Erlang's built-in ability to trace and save events from processes. Collected traces are analyzed offline, and the results are presented via a web-based interface. However, it is limited to profiling applications running on a single multicore machine rather than a cluster of machines (which is our main focus). RefactorErl was designed to model the relationships between Erlang processes. This tool is capable of detecting both explicit (e.g., process hierarchy and communication) and hidden relationships (e.g., via file operations or ETS tables), which allows developers to better

comprehend the code on which they are working. Unfortunately, RefactorErl is not able to detect possible bottlenecks or undesired system states, which often prove to be much harder to handle and have a disastrous impact on the application.

3. Erlang Performance Lab

The ErlangPL tool (`http://www.erlang.pl/`) aims at supporting software engineers working on systems running on top of a BEAM virtual machine (a.k.a. Erlang Runtime System [10]). The ErlangPL project was open-sourced in February of 2017, and its source code is available on the GitHub repository `https://github.com/erlanglab`. It is available as a command line tool, with a graphical user interface (GUI) rendered in any modern browser. The pre-compiled executable is compatible with the operating systems on which the BEAM is installed. Despite having "Erlang" in its name, it can support systems running on top of the BEAM virtual machine regardless of the programming language used for implementation (e.g., Erlang [8], Elixir [7], or LFE [25]).

In order to access the GUI, one needs to open a terminal and start `erlangpl` from the command line, providing the name of the node to be inspected. Once started, ErlangPL tries to connect over the Erlang Distributed protocol to the entire cluster to which the provided node name belongs. If successfully connected to a cluster, an HTTP service is started so that one can navigate a web browser through it. The browser renders a GUI implemented in React.js [30]. The GUI consists of several tabs, each visualizing a different aspect of the system. We describe views dedicated to the network traffic between the nodes and message passing between the processes. At the time of writing, there are also other views available (e.g,. rendering a supervision tree or displaying a dashboard with metrics and charts). Those other views are similar to already-existing tools like **observer** [28] and are not covered in this paper.

3.1. Network traffic visualizations

Traffic visualizations are divided into three levels of detail. The first-level view shows ingress and egress traffic between the inspected node and the cluster of the connected nodes. Nodes are represented as circles, and the centrally located one refers to the inspected node (see Figure 1). In the central circle, there is a counter with the number of messages processed every five-second period. The counter is updated every five seconds. To visually represent the traffic, animations of particles flowing from and to the inspected node are rendered. The direction in which the particles flow indicates ingress or egress. The number of animated particles depends on the value of the counter, so the more messages processed, the more particles produced. In the other circles, there is a percentage value that represents the number of messages processed by a node divided by the counter from the central circle.

This high-level overview can help us understand whether the processes running on the nodes send messages to the remote processes and how such traffic is distributed across the cluster.

Clicking on a circle opens the second-level traffic view. The view renders traffic between each pair of nodes within the cluster. A similar animation technique as the one described above is used, so it can help understand the traffic patterns. For example, a visualization of distributed database Mnesia [26] executing different types of operations can help understand which operations require more intensive communication between the nodes and how the traffic is balanced across the cluster [34]. In the scenarios where Mnesia asynchronously replicates the writes from the node executing operations to all other nodes, the animations show that the messages are only sent in one direction. On the other hand, in a visualization for scenarios where Mnesia executes transactions requiring a two-phase commit, there are a higher number of messages exchanged between the nodes, and the messages are sent in both directions (to and from the node on which the transaction was initiated). These differences in traffic patterns can be easily spotted when observing the animations and comparing the counters in the circles.

Clicking on any of the nodes in the second level transforms the view to the third level, in which all of the measurement-related details of that specific node are shown.

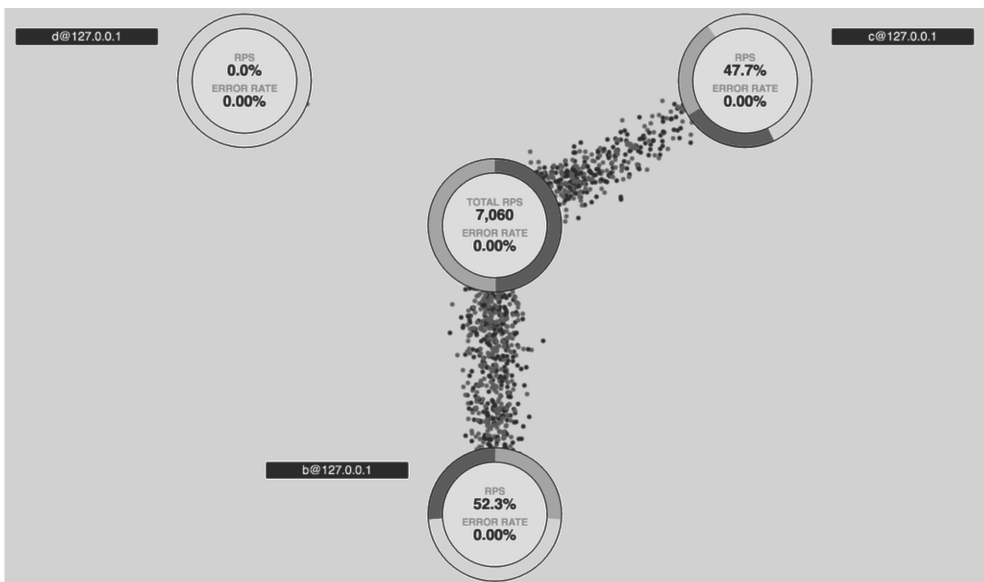

Figure 1. First-level cluster traffic view

3.2. Message-passing visualizations

The communication between the processes running on the BEAM is based on message passing. Each process has an unbound mailbox, and any other process can send it a message. This can lead to overload conditions when more messages are produced than consumed. A common technique to identify a potential bottleneck related to

mailbox overload is to look for processes with a long message queue. Such a technique can indeed help find overloaded processes, but it does not give a more holistic view of where the possible sources of such loads are.

Clicking on one of the nodes in the first-level traffic view described in the previous section opens the second-level view, where the messages passed between the Erlang ports and processes are visualized as shown in Figure 2. In this view, the circles represent the ports and processes running on the clicked node, and the animated particles represent messages. The direction in which the particles flow indicates whether a process is sending or receiving. This view can suggest which processes or ports are potential bottlenecks; e.g., if many processes pass messages to a single process or if one process is a source of messages passed to many other processes.

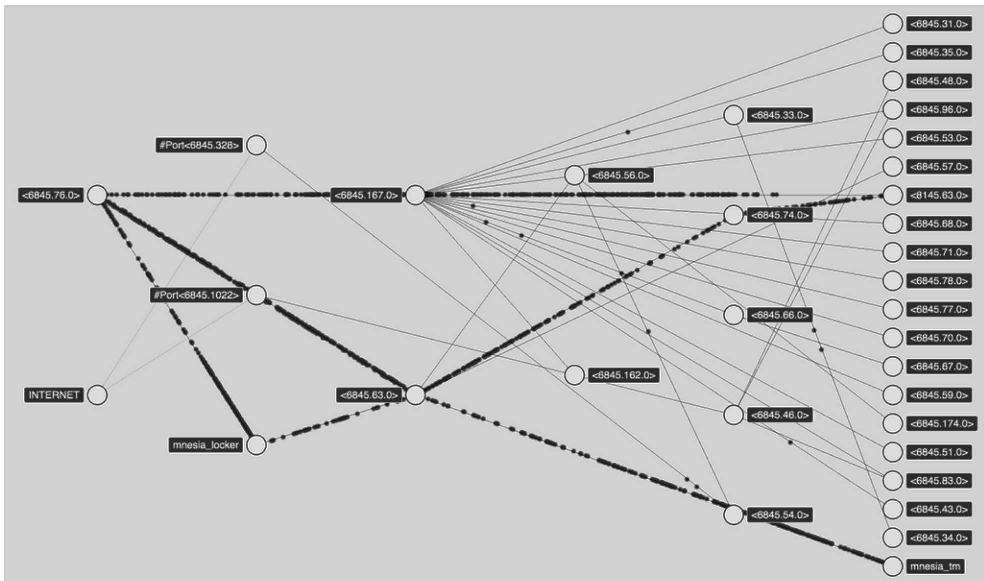

Figure 2. Second-level message-passing view

3.3. Implementation details

Command line tool `erlangpl` is implemented as a self-contained escript [13], meaning that all assets required to render the GUI and all non-standard .beam files required to execute the program are compressed and appended to the shell script. When executed, the script extracts its assets and calls the main function. Then, Erlang Distributed is started by calling the `net_kernel:start/1` function, and the node is registered in the local Erlang Port Mapper Daemon [9] as erlangpl@127.0.0.1. The default node name can be changed by passing the relevant argument to the script.

After the newly created node connects to the observed cluster, it spawns a remote process named epl_tracer on each of its nodes. The process is used to gather trace

events and serve as a proxy for remote function calls. The process is linked to the parent node, so it is killed as soon as the ErlangPL program is terminated.

The metrics needed to render the traffic visualizations described in Section 3.1 are gathered by polling the observed nodes every five seconds. Each node executes `net_kernel:nodes_info/0`, and the results are gathered on the erlangpl node, where they are processed and made available for the front-end program rendering the GUI.

The metrics needed to render the message visualizations described in Section 3.2 are gathered by tracing all processes with the trace flags of types 'send' and 'receive' (see Erlang code listing below).

```
TraceFlags = [send, 'receive', procs, timestamp],
erlang:trace(all, true, TraceFlags)
```

For each type of received trace event, a relevant counter is stored in a local ETS [11] table. Counters are polled every five seconds and processed further on the ErlangPL node, where they are made available for the GUI.

In systems where the processes communicate intensively with each other, the above tracer settings can cause many trace events to be generated. As a result, the overhead related to event processing and counter aggregating can overwhelm the epl_tracer process, and the counter's polling can timeout. In future versions of ErlangPL, this issue can be addressed by implementing the Erlang tracer behavior [12] as NIFs.

4. Distributed traffic simulation system

The distributed system analyzed in this paper is a microscopic urban traffic simulator that was presented in detail in [36]. This section summarizes only the crucial aspects that are interesting in the context of the presented tool.

Microscopic traffic simulation is a rapidly developing domain, used in various problems related to autonomous cars and urban traffic management. It requires high accuracy that imposes complex and detailed models of traffic as well as large-scale – a rapid and accurate simulation of whole cities is a desired possibility. These factors make the problem very computationally demanding; therefore, it is suitable for execution on highly-parallel hardware. However, the problem of distributing such a computation on several computing nodes is not straightforward.

The simulation model implemented in the system is an extended version of the well-known Nagel-Schreckenberg approach [27]. It uses discrete representation of all model parameters – space, time, and velocity. It introduces several improvements over the classic model, which make it more realistic in urban traffic modeling:

- multi-lane roads,
- lane change action,
- road crossings with common cells,
- traffic lights or right-of-way rules.

A visual representation of a fragment of the modeled road system is presented in Figure 3.

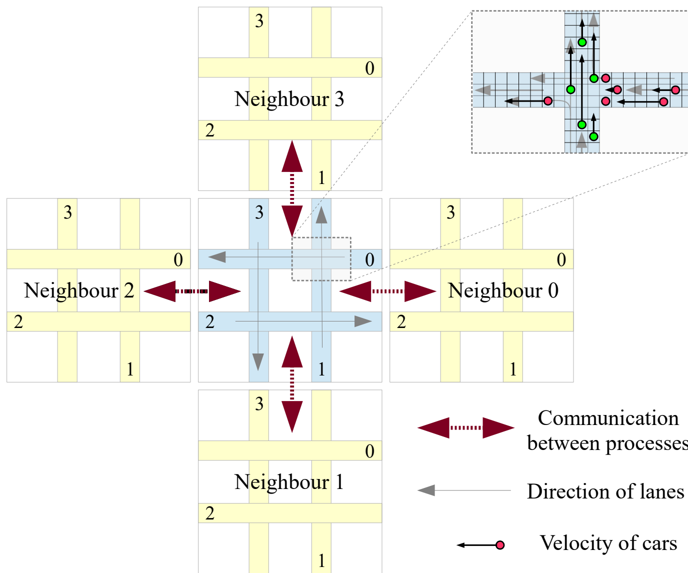

Figure 3. Visual representation of five crossroads in urban road system model. Central crossroad communicates with four neighboring crossroads only

Each lane is composed of a given number of cells that represent the possible locations of the cars. Each cell can be either occupied by one car or free. Each car is characterized by its current velocity, maximum acceleration, and probability of turning at a crossroad. The algorithm of updating the simulation state analyzes all cells and moves all cars forward (if possible) according to their characteristics. The sequential version of the algorithm is rather simple.

In order to parallelize the computations, the simulated space was split into subspaces covering single crossroads. Each crossroad is updated in parallel in a dedicated process, and the results of the update are sent to the processes responsible for the neighboring crossroads. The communication between the processes is limited to the four neighbors only. There is no need for a centralized synchronization of the computations, which makes the algorithm rather unique.

The time required by each of the processes to update the crossroad may vary significantly. This depends strongly on the number of cars present in the crossroad. The computations are synchronized using a simple mechanism of waiting for their neighbors to finish the previous step. This approach guarantees the correctness of the simulation without any centralized time control.

The simulation makes it possible to verify the influence of the controlled desynchronization of the computations on scalability. Desynchronization is defined as the

ability to compute a few further steps of a simulation without the states from the neighboring crossroads. With simple assumptions, this method also allows for the correct simulation.

The simulator is implemented in Erlang and designed for execution on a cluster of computers running connected Erlang virtual machines. It was tested on the Prometheus computer, which is a part of the PL-Grid infrastructure [29]. It offers 2232 computing nodes with 24 physical cores each. The simulation used up to 800 nodes, which is 19,200 cores working simultaneously on a single task. Although the scalability results are satisfactory (Fig. 4), we cannot claim that we fully understand the reasons for particular phenomena observed in the results.

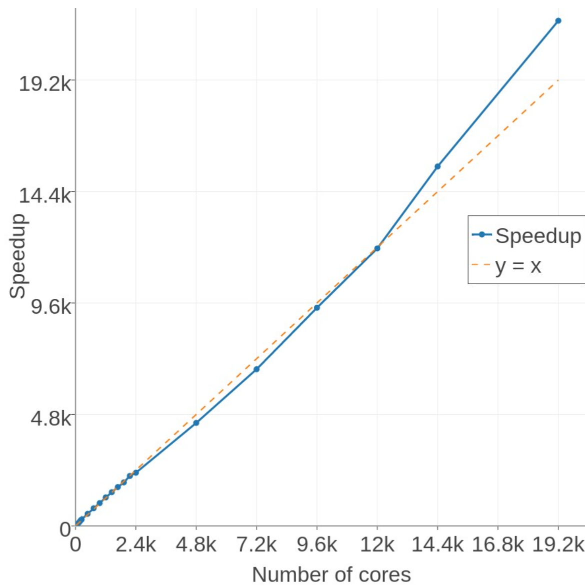

Figure 4. Scalability of simulation system running on supercomputer [36]

A very interesting aspect of the method is the distribution of the tasks between the available computing nodes. Obviously, the cost of communication between different nodes is higher than the cost of sending a message to the same node. Three methods were tested:

1. random assignment;
2. scatter assignment, where the grid of the crossroads was traversed systematically, and each crossroad was assigned to the next node in a cyclic list;
3. bulk assignment, where the previous method assigned a fixed number of consecutive crossroads to one node.

The results presented here were collected using the third task assignment method; the first two methods gave worse results.

5. Analysis of distributed system dynamics

The presented traffic simulation system was analyzed using the ErlangPL tool in several different configurations. The results presented in this section show that the tool is easy to use, does not affect the tested system, and makes the investigation of various aspects of its behavior possible.

The tests were carried out using a cluster of 20 modern physical computers connected to a fast local area network. Each computing node was equipped with four physical cores of an Intel i5 CPU. The computer was run under the control of Linux OS and used Erlang 19.3.

The integration of the ErlangPL with the traffic-simulation systems was straightforward. Once the system was started, the ErlangPL tool could be connected to any of the nodes, and the tracing would start automatically. The first results were available on the web interface after a few seconds – the interface started to show the communication intensity between the nodes in real-time.

The first set of tests involved three computing nodes and aimed at verifying the ability of detecting and presenting the differences in communication intensity between the computing nodes. A simulation of 20 crossroads was started on 3 nodes only, which resulted in an uneven assignment of tasks to the nodes. Figure 5 shows the visualization of communication between the nodes with three different task-distribution methods.

The visualization displays five Erlang virtual machines in the cluster, but only three of them are involved in the simulation. Two others represent the node used by the simulation to initialize the tasks and the node of ErlangPL. It is clearly visible that the intensity of communication between the three nodes taking part in the simulation is high and differs between the task-distribution methods, which explains the differences in the overall performance. The bulk method requires significantly fewer messages and gives the best performance; however, it seems that it is still not the optimal solution. Figure 5 reveals that one pair of nodes exchanges fewer messages than the other two pairs, which suggests that the tasks are not equally distributed in the system. This is an identified potential field for improvement.

In order to verify the possibility of visualizing larger numbers of nodes, a set of tests involving all 20 computers were carried out. The simulation task was far bigger than before – there were 320 crossroads in the model. ErlangPL managed to analyze and display the network of connections after a few seconds and fluently presented the simulation initialization and further processing. Selected frames are presented in Figure 6. Only the results of the bulk task-distribution method are presented. Visualization of all 20 nodes can be hard to read for an inexperienced user. Therefore, ErlangPL provides a very simple filtering (presented on the bottom of Figure 6) that is enabled by mouse movements. This view shows only the selected node and its communication with the other nodes, which allows us to identify cooperating nodes and estimate the communication intensity between them. In the case of our simulation, this intensity turned out to be rather irregular.

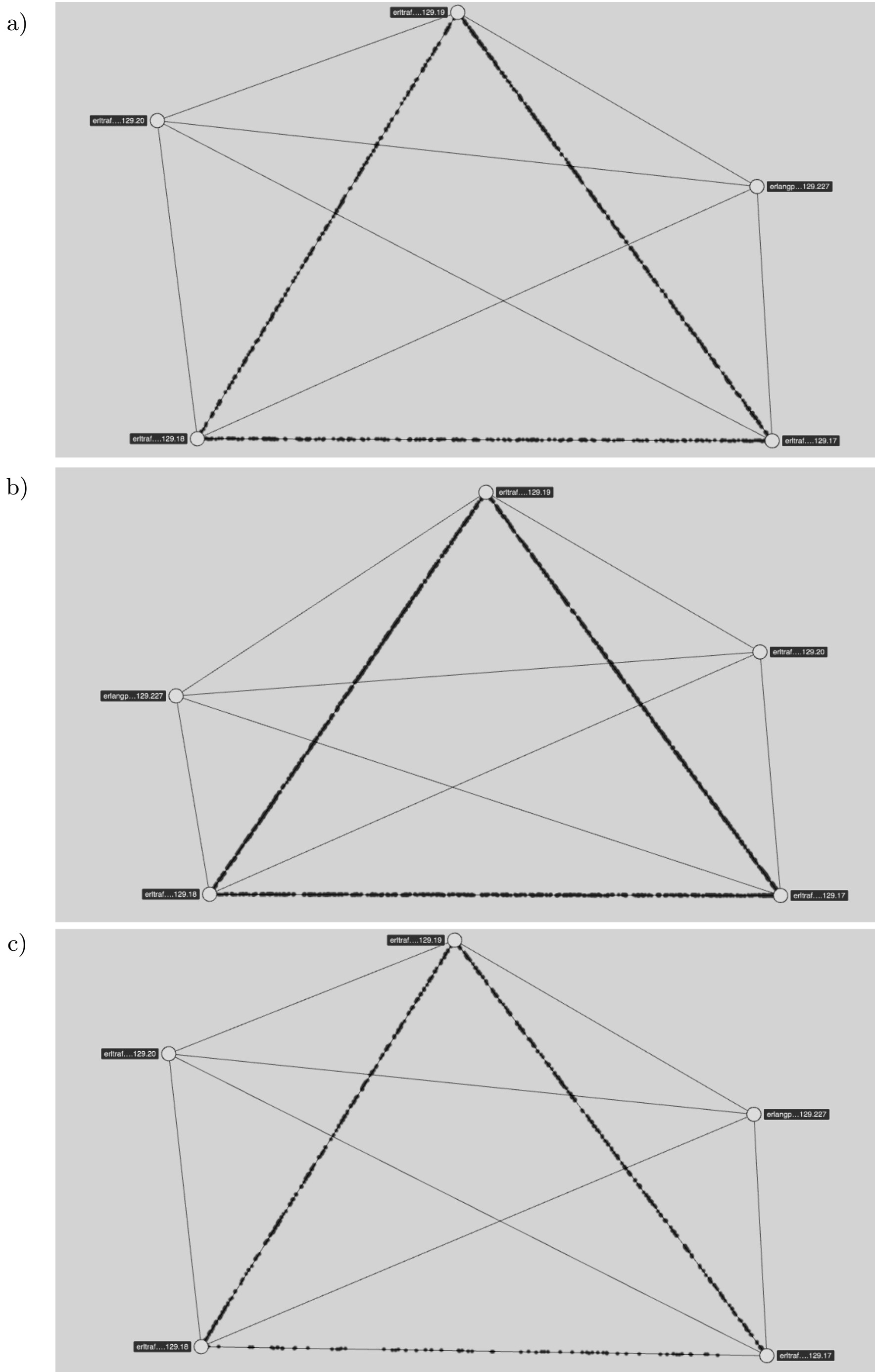

Figure 5. ErlangPL visualization of three nodes running 20 simulation tasks with different task-distribution methods: a) random; b) scatter; c) bulk

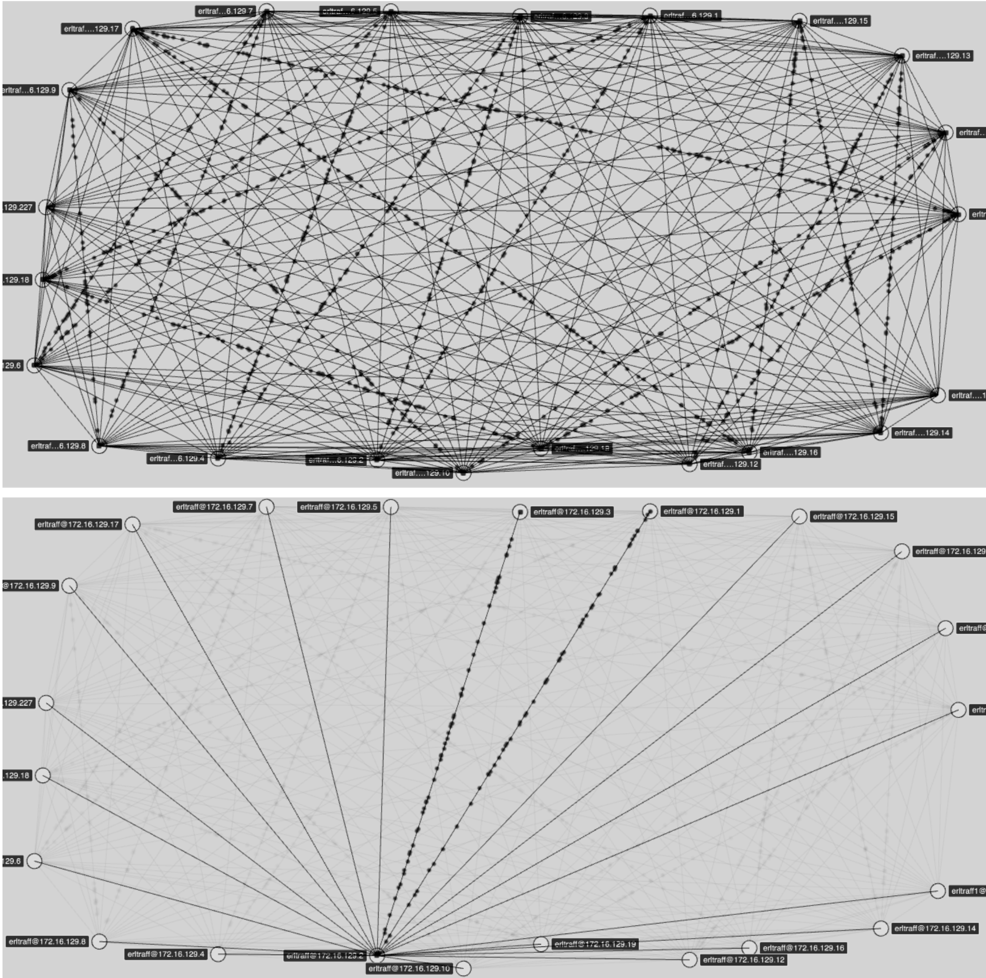

Figure 6. ErlangPL visualization of 20 nodes running common simulation tasks

Remarkably, there was no overhead in the monitoring system on the simulation performance. The mechanism that collects information about the communication between nodes is very lightweight, and it had no measurable impact on the computationally-intensive task.

The visualization of a cluster structure and communication between the nodes does not solve any issues automatically. However, it provides a significant amount of information in a way that makes it possible to be analyzed by a human being. The real-time presentation of the changes in communication intensity and visual representation makes it possible to draw conclusions about the reasons and effects, which can lead to identifying the problems in large-scale highly parallel systems.

The tests revealed a number of possible improvements that might be useful and could provide further information. This included CPU and memory utilization on the nodes, lengths of the message queues, or improved node layout creation.

6. Conclusions and further work

Modern, highly-concurrent, large-scale systems require new methodologies for solving issues caused by their complexity and high dynamics. New methods are needed to understand the phenomena taking place in such systems. It seems that the problem can be addressed with an approach based on providing a holistic view of a system to an expert. The proper real-time visualization of the changing states of a system can lead to the identification of potential problems.

The presented tool, Erlang Performance Lab, provides a set of visualizations on different levels of abstractions that can be quickly switched and analyzed in real time. The tool works almost transparently for the monitored system, and it is very simple to configure and use. We hope that it will become one of the basic tools used by distributed system developers.

The performed tests showed several interesting directions for the further development of the created tool. Enriching the existing visualizations with additional data seems very important. A more-sophisticated analysis of the collected data with user-defined functions would also be interesting.

The planned future work consists of applying the constructed monitoring tool for monitoring other simulation systems (e.g., [21]) or metaheuristic computing systems (e.g., [5]). The visualization provided will help in the diagnosis of technical communication-related issues, but it will also help in the observation of certain algorithmic features, such as the migration phenomenon in the parallel evolutionary algorithms as well as the relationship between them (e.g., influence of migration on the performance of the system).

Acknowledgements

The research presented in this paper was partially supported by AGH University of Science and Technology Statutory Project.

References

[1] Allen J.: *Effective Akka.* O'Reilly Media, 2013.

[2] Apache Ignite. `https://ignite.apache.org`.

[3] Armstrong J.: *Programming Erlang: Software for a Concurrent World.* The Pragmatic Programmers, LLC, North Carolina, USA, 2013.

[4] Basiri A., Behnam N., de Rooij R., Hochstein L., Kosewski L., Reynolds J., Rosenthal C.: Chaos engineering, *IEEE Software*, vol. 33(3), pp. 35–41, 2016.

[5] Byrski A., Kisiel-Dorohinicki M.: *Evolutionary Multi-Agent Systems: from inspirations to applications*. Studies in Computational Intelligence, vol. 680. Springer, 2017.

[6] Cesarini F., Vinoski S.: *Designing for Scalability with Erlang/OTP: Implement Robust, Fault-Tolerant Systems*, O'Reilly Media, 2016.

[7] Elixir. `https://elixir-lang.org/`.

[8] Erlang. `http://www.erlang.org/`.

[9] Erlang Port Mapper Daemon. `http://erlang.org/doc/man/epmd.html`.

[10] Erlang Runtime System. `http://erlang.org/doc/apps/erts/introduction.html`.

[11] Erlang Term Storage. `http://erlang.org/doc/man/ets.html`.

[12] Erlang tracer behavior. `http://erlang.org/doc/man/erl_tracer.html`.

[13] escript. `http://erlang.org/doc/man/escript.html`.

[14] Ford N.: *Functional Thinking: Paradigm Over Syntax*, 1st Edition, O'Reilly Media, 2014.

[15] Hazelcast. `https://hazelcast.com/`.

[16] Herlihy M., Shavit N.: *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

[17] Horanyi G.: Intuition Engineering with Docker. `https://medium.com/@ghoranyi/`, 2016.

[18] Infinispan. `http://infinispan.org`.

[19] Isaacs K.E., Giménez A., Jusufi I., Gamblin T., Bhatele A., Schulz M., Hamann B., Bremer P.T.: State of the art of performance visualization. In: *Proceedings of the 16th annual Eurographics Conference on Visualization (EuroVis 2014)*, 2014.

[20] Kamon. `http://kamon.io/documentation/kamon-akka/0.6.6/overview/`.

[21] Kazirod M., Korczynski W., Fernández E., Byrski A., Kisiel-Dorohinicki M., Topa P., Tyszka J., Komosinski M.: Agent-oriented Foraminifera Habitat Simulation. In: *Procedia Computer Science, International Conference on Computational Science, ICCS 2015*, vol. 51, pp. 1062–1071, 2015. `https://doi.org/10.1016/j.procs.2015.05.264`.

[22] Lamarche-Perrin R., Schnorr L.M., Vincent J.M., Demazeau Y.: Evaluating trace aggregation for performance visualization of large distributed systems. In: *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 139–140. IEEE, 2014.

[23] Li H., Thompson S.: Multicore profiling for Erlang programs using Percept2. In: *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, pp. 33–42. ACM, 2013.

[24] Lightbend intelligent Monitoring. `https://www.lightbend.com/platform/production/intelligent-monitoring`.

[25] Lisp Flavoured Erlang. http://lfe.io/.

[26] Mnesia. http://erlang.org/doc/man/mnesia.html.

[27] Nagel K., Schreckenberg M.: A cellular automaton model for freeway traffic, *Journal de Physique I*, vol. 2(12), pp. 2221–2229, 1992.

[28] Observer tool. http://erlang.org/doc/man/observer.html.

[29] PL-Grid. http://www.plgrid.pl/.

[30] React.js. https://facebook.github.io/react/.

[31] Reynolds J., Rosenthal C.: Vizceral Open Source. https://medium.com/netflix-techblog/vizceral-open-source-acc0c32113fe, 2016, Netflix Technology Blog.

[32] Rosà A., Chen L.Y., Binder W.: Profiling actor utilization and communication in Akka. In: *Proceedings of the 15th International Workshop on Erlang*, pp. 24–32. ACM, 2016.

[33] Schnorr L.M., Huard G., Navaux P.O.A.: A hierarchical aggregation model to achieve visualization scalability in the analysis of parallel applications, *Parallel Computing*, vol. 38(3), pp. 91–110, 2012.

[34] Slaski M.: Erlang Performance Lab. https://www.youtube.com/watch?v=ncedupb-Rqw, 2017. Slides available at https://speakerdeck.com/michalslaski/erlang-performance-lab.

[35] Tóth M., Bozó I.: Detecting and Visualising Process Relationships in Erlang. In: *Procedia Computer Science*, vol. 29, pp. 1525–1534, 2014.

[36] Turek W.: Erlang-based desynchronized urban traffic simulation for high-performance computing systems, *Future Generation Computer Systems*, vol. 79, pp. 645–652, 2018. http://dx.doi.org/https://doi.org/10.1016/j.future.2017.06.003.

[37] Vizceral. https://github.com/Netflix/vizceral.

[38] Weisfeld M.: *The Object-Oriented Thought Process*. Addison Wesley Professional, 2013.

[39] WombatOAM. https://www.erlang-solutions.com/products/wombat-oam.html.

Affiliations

Michał Ślaski
Erlang Solutions, ul. Batorego 25, 31-135 Krakow, Poland, michal.slaski@erlang-solutions.com

Wojciech Turek
AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Krakow, Poland, wojciech.turek@agh.edu.pl

Arkadiusz Gil
Erlang Solutions, ul. Batorego 25, 31-135 Krakow, Poland, arkadiusz.gil@erlang-solutions.com

Bartosz Szafran
Erlang Solutions, ul. Batorego 25, 31-135 Krakow, Poland,
bartosz.szafran@erlang-solutions.com

Mateusz Paciorek
AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Krakow, Poland,
mpaciorek@agh.edu.pl

Aleksander Byrski
AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Krakow, Poland,
olekb@agh.edu.pl

Received: 11.01.2018
Revised: 30.03.2018
Accepted: 02.04.2018