Computing and Informatics, Vol. 35, 2016, 870–889

USING AN ACTOR FRAMEWORK FOR SCIENTIFIC COMPUTING: OPPORTUNITIES AND CHALLENGES

Bartosz Balis, Krzysztof Borowski

AGH University of Science and Technology
Faculty of Computer Science, Electronics and Telecommunications
Department of Computer Science
Al. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: balis@agh.edu.pl

Abstract. We examine the challenges and advantages of using an actor framework for programming and execution of scientific workflows. The following specific topics are studied: implementing workflow semantics and typical workflow patterns in the actor model, parallel and distributed execution of workflow activities using actors, leveraging event sourcing as a novel approach for workflow state persistence and recovery, and applying supervision as a fault tolerance model for workflows. In order to practically validate our research, we have created Scaflow, an Akka-based programming library and workflow execution engine. We study an example workflow implemented in Scaflow, and present experimental measurements of workflow persistence overhead.

Keywords: Scientific workflows, actor model, workflow patterns, Akka framework

1 INTRODUCTION

With the growth of cheap computing power available on demand, scientific applications become increasingly more complex, and their development and execution – ever more challenging. Scientific workflow management systems [1] have been developed to help manage this complexity by providing programming environments and execution services for scientific applications.

The actor model of computation has an established position as a formalism for modeling concurrent systems [2], and there have been many implementations of actors in different programming languages [3]. However, only relatively recently actor

frameworks have became mature enough to be adopted as a mainstream technology for building complex distributed systems, an example being the Akka framework [4]. Compared to traditional distributed system architectures, actor frameworks present a considerably different approach to concurrency, state persistence and fault tolerance.

In this paper, we investigate the challenges and opportunities of using the actor model and framework for implementation and execution of scientific workflows. In particular, we focus on the following topics: implementing the workflow model of computation using actors, leveraging actors for parallel and distributed processing of workflow activities, a novel approach to workflow persistence and recovery based on event sourcing, and a new fault tolerance model for workflows based on the supervision approach. In order to evaluate our research, we have implemented Scaflow, a workflow management system based on the Akka framework.

This paper is organized as follows. Section 2 contains overview of related work. Section 3 presents the concept of implementing workflow semantics in the actor model. Section 4 describes advantages of using an actor-based framework for design and execution of scientific workflows. In Section 5, a practical evaluation of the proposed concepts is presented. Section 6 concludes the paper.

2 RELATED WORK

Scientific workflows are essentially distributed applications composed of application programs, typically executing on distributed nodes, and a workflow engine orchestrating them. Workflow systems need to provide a complex infrastructure to enable distributed execution of scientific workflows. Typically, this infrastructure includes agents that reside on remote nodes and execute the application programs, and a messaging system used for communication between the workflow engine and the remote agents. Example workflow systems that follow this architecture include Datafluo [5] and HyperFlow [6]. Some workflow systems rely on complex middleware services for deployment and execution of workflow tasks on distributed computing nodes. Examples include WS-PGRADE [7, 8] which utilizes the DCI-Bridge platform [9], and Pegasus using Condor [10]. The Taverna workflow system [11], in turn, relies on Web Services as a means of invoking distributed computations. In Kepler [12] distributed deployment and execution is supported by providing workflow components dedicated for particular computing infrastructures, for example Amazon EC2 or Grid middleware [13]. In this approach, business logic is intertwined with "infrastructure" logic in the workflow graph, i.e., special workflow activities are used to interact with a particular distributed computing infrastructure. Consequently, the workflow is tightly coupled to this infrastructure which is a major disadvantage of this approach.

In actor frameworks such as Akka, a distributed computing middleware is already in place. Remote actors are transparent as far as their creation and communication with them is concerned. Of course, the remote actor systems need to

be deployed on distributed nodes, but once this is done, the communication with them utilizes the same programming abstractions and does not require a dedicated infrastructure or middleware.

A failure in the middle of a long-running and resource-intensive workflow would result in a significant loss of time and money. Consequently, workflow systems provide the capability to persist the state of the workflow engine so that it can be recovered in order to resume the workflow execution after a failure. The technique most widely used for workflow persistence and recovery in existing systems is checkpointing (e.g. Pegasus [10] and Askalon [14]), i.e. periodic saving of a complete snapshot of the workflow's state. A full workflow checkpoint saves the information about the internal state of the workflow and its activities, as well as intermediate data required to resume the execution of the workflow.

Actor frameworks feature a different approach to persistence, based on *event sourcing*. In event sourcing, rather than persisting the most recent state, a complete history of *state changes* is saved in an *event log*. The full state of an actor system can be subsequently recovered by replaying the event log and applying the state changes again. Importantly, the actual operations that caused the state changes are not repeated, so that their side effects are not executed multiple times. Event sourcing applied to workflow persistence has the following advantages over checkpointing:

- Checkpoints are usually expensive and can only be done every so often. On the other hand, event sourcing is lightweight so that every state change is implicitly persisted as it happens. Because of the checkpoint interval, the progress of execution since the last checkpoint is lost after a failure, which is not the case with event sourcing.
- The event sourcing mechanism allows the restoration of the workflow state to any given point of execution, enabling so called "smart re-runs" [12], i.e. partial recovery of a historical workflow, in order to re-execute its remaining part, possibly with some changes introduced.
- Workflow execution typically needs to be suspended before performing checkpointing. Event sourcing is performed on the fly, without suspending the actors.

Addressing the disadvantages of checkpointing has been a subject of much research. An improved, decentralized workflow checkpointing approach is proposed in [15]. The checkpoints are captured independently and saved asynchronously by each workflow task. Consequently, there is no need to suspend the workflow while saving the checkpoint. However, the model is very complex and difficult to implement. For example, since all checkpoints are performed independently, consistent recovery of the workflow's global state is challenging. Event sourcing is similar in that it is decentralized (every actor logs its state changes independently). However, a consistent recovery of the global state is much easier because a complete history of state changes is available. In [16], a user-level approach to workflow checkpointing is presented which is similar to event sourcing in that it records the signature of a workflow execution. The workflow is recovered by repeating its execution from

this signature in order to rebuild the internal state of the workflow and workflow activities, but skipping the actual execution of activities that have already succeeded. However, this mechanism, rather than being implicitly implemented by the workflow system, relies on workflow-specific checkpointing activities that the workflow programmer needs to manually add to the workflow graph.

Another approach to workflow persistence, similar to event sourcing, is proposed by Köhler and others [17]. In this work, provenance records (which resemble event sourcing logs) are used to replay the workflow, just as in event sourcing. The biggest difference is that full re-execution of stateful activities is required in order to restore their state. For long-running computations this can be very expensive, so in such cases the approach falls back to checkpointing. Event sourcing was specifically designed to solve this problem by introducing the concept of domain events that enable restoration of internal state of actors without repeating their side effects.

There are efforts to implement flows on top of actors, notably Akka Streams.[1] The investigation of Akka Streams as a possible technology for building a scientific workflow system has led us to the conclusion that this technology has been created with quite different use cases in mind. Akka Streams is designed for *real-time streaming*, i.e. real-time processing of unbounded streams of small data elements. The central problem here is back pressure, i.e. controlling the intensity of the stream rates in order to avoid fast producers flooding slow consumers with data. Scientific workflows, on the other hand, are designed for resource-intensive computations, where the number of the input data sets is bounded, individual input data elements are often big, and tasks can be long-running. Consequently, the central problems revolve around persistence and recovery, fault tolerance, parallel processing, and leveraging distributed resources. In summary, even though Akka Streams has an elegant graph-oriented API for composing flows, the underlying technology does not satisfy the requirements of scientific workflows.

3 MAPPING WORKFLOWS TO ACTORS

3.1 Workflow Model

In this paper, we assume a workflow model similar to Process Networks [18] in which a workflow can be described as a directed *multigraph* where the nodes represent workflow activities, while the edges denote dependencies between them, i.e. data flow or control flow. A single node of a workflow, depicted in Figure 1, represents workflow activity `P` with multiple input and output "channels".

A workflow activity is *fired* (executed) as soon as its *firing conditions* are fulfilled. These conditions will vary depending on the activity type. For example, a firing condition may require an input element on all input channels (*synchronization pattern*), or on any one of them (*merge pattern*). Some examples of firing

[1] `http://doc.akka.io/docs/akka-stream-and-http-experimental/2.0.3/scala.html`

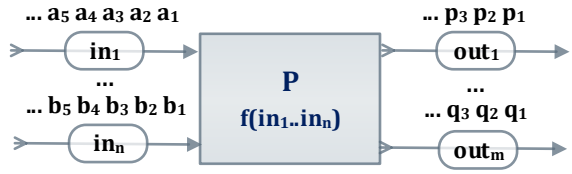


Figure 1. The adopted workflow model

conditions related to various workflow patterns [19] are shown in Table 1. Upon firing, the transformation function `f` is invoked and its outputs are produced to the output channels. Consequently, an activity can be viewed as a continuous function transforming input data sequences into output data sequences. More details about this workflow model can be found in [6].

Name	Semantics
Synchronization	Data elements have arrived on all inputs
Simple merge	Data element has arrived on any input
Discriminator	Data elements have arrived on any n-out-of-m inputs
Structured join	Data elements have arrived on n-out-of-m inputs, where n denotes the number of outputs activated in an earlier workflow activity.

Table 1. Examples of firing conditions (workflow patterns) and their semantics

3.2 Conceptual Difficulties

The most important characteristics of actors are as follows:

1. actors do not share state – they interact and exchange data only via asynchronous messages;
2. messages arriving at an actor are queued and processed sequentially;
3. actors update their internal state in response to arriving messages.

At a first glance an actor system easily maps to a workflow: workflow activities are simply actor instances, while data and control flow between activities are represented by messages exchanged between actors. However, this naive view quickly runs into serious difficulties, summarized in Table 2.

First, workflow activities have multiple input and output channels whose types are well-specified. While it does not necessarily imply static type enforcement (uncommon in workflow languages/engines, WOOL [20] being a rare exception), at a minimum we know well what a given input or output represents (e.g. "a CSV file with temperature measurements"). Actors, on the other hand, have a single input mailbox which consumes messages of *any* type. An actor may define multiple handlers that process messages based on their their structure or content. Second, as far

Aspect	Workflows	Actors
Input data	Activities have multiple (typed) input channels	One input message mailbox
Output data	Activities have multiple (typed) output channels	No output channels, no returned data types
Flow patterns	Complex patterns, e.g. synchronization of inputs from multiple sources	Simple "fire-and-forget" asynchronous messaging

Table 2. Mapping workflows to actors: conceptual difficulties

as output channels are concerned, actors have no concept of "outgoing messages" at all.

Finally, the conceptual problems become even more apparent if we consider flow patterns commonly found in workflows [19, 21]. Plain actors support only a simple interaction via "fire-and-forget" messaging. However, workflows need to support various complex patterns, such as synchronization of inputs from multiple sources, non-deterministic merge of inputs, etc. (compare firing conditions in Table 1).

Despite these conceptual difficulties, we argue that there are significant benefits to be obtained using actors for implementing and executing scientific workflows. The following subsection shows how workflow semantics can be implemented with actors, while Section 4 describes the advantages of using actors for workflow execution.

3.3 Design of Actor-Based Workflow

Figure 2 conceptually shows an implementation of a workflow node using the actor model.

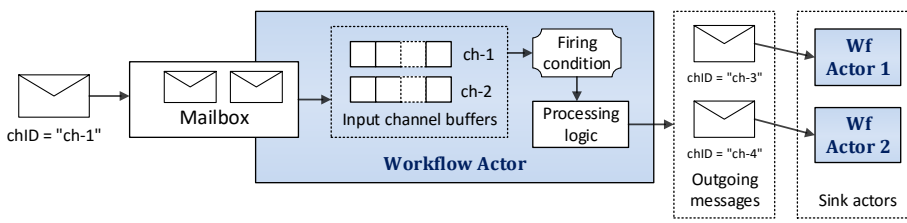


Figure 2. Conceptual implementation of a workflow component

Different communication channels in a workflow are distinguished by attaching a unique channel identifier to each message exchanged between actors. Workflow actors, in turn, have internal message queues for the individual input channels. A message arriving at an actor is first stashed in the proper queue. Next, a firing condition is checked (e.g. if there are messages on all input queues). In the case the condition is fulfilled, the appropriate messages are removed from the queues and passed to a processing logic subroutine which invokes the actual domain-specific

data processing. The results returned by the processing logic subroutine are immediately packed as messages and sent to the connected workflow nodes ("sink actors"). To this end, the workflow actor needs to have the following mapping: $(out_i \mapsto (actorRef, channelId))$, where out_i is the i^{th} output of the processing logic subroutine, while $(actorRef, channelId)$ identify the target sink actor and the input channel of this actor, to which out_i should be sent. Effectively, outgoing channels are implemented in this way.

4 USING ACTORS FOR WORKFLOW EXECUTION

This chapter presents the advantages of using an actor framework for execution of scientific workflows. The results of our research concern three areas: parallel and distributed processing in workflows, workflow persistence and recovery, and workflow fault tolerance.

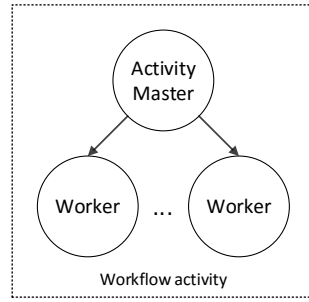
4.1 Parallel and Distributed Processing

In large-scale scientific workflows, applications invoked from workflow activities are routinely mapped onto resources of distributed computing infrastructures. An actor-based workflow engine can facilitate parallelization of computations within one node, and their distribution across many nodes. Table 3 highlights this advantage of actors by comparing three platforms used for building production-quality distributed systems. Without going into details of each platform, let us note that actors are a single abstraction used for implementing three crucial aspects in a complex distributed system: system structure, managing concurrency, and managing distribution. Actor-based APIs by default use solutions characteristic for distributed systems, such as location-transparent references (actors can be referred to by names) and interaction by message passing.

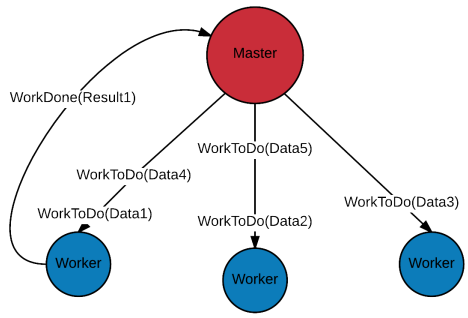
	Java	Node.js	Akka
Unit of program structure	Object	Function	Actor
Unit of concurrency	Thread	Async call	Actor
Unit of distribution	Process	Process	Actor

Table 3. Parallel and distributed programming: comparison of platforms

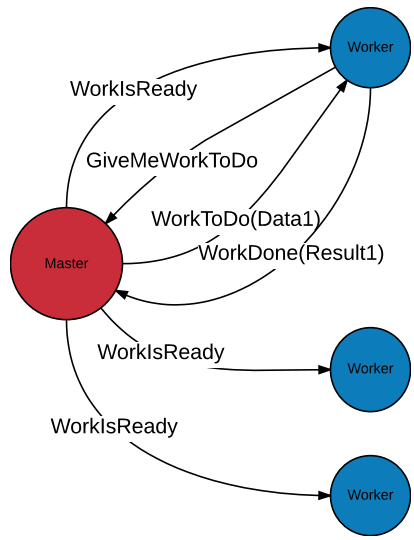
An architecture of a workflow activity supporting parallel and distributed processing is proposed in Figure 3 a). This architecture follows a widely used Master-Worker pattern. The Activity Master plays the role of message recipient, while the Workers perform the actual message processing. Such an architecture accomplishes several purposes: first, it isolates failure-prone tasks in separate actors (Workers), enabling effective fault-tolerance strategies (see Section 4.3); second, it allows parallel processing of multiple activity firings in separate Worker instances; finally, it enables distribution of work via remotely deployed Worker actors.



a)



b)



c)

Figure 3. Workflow activity architecture and interaction variants between Activity Master and Workers: a) Workflow activity architecture, b) Master-Worker interaction: "push" model, c) Master-Worker interaction: "pull" model

In typical actor systems where message processing is relatively lightweight, it is common and safe to create one actor per one request (message) to be processed. The underlying dispatchers will take care of scheduling message processing onto thread pools. However, workflow activities can be resource-intensive (in terms of CPU, memory, I/O bandwidth usage, etc.), so job scheduling and load balancing should be done by the workflow engine itself, rather than via standard mechanisms of an actor framework.

The first step towards this is to maintain a controlled pool of Workers. The Master can distribute work to them according to either of two strategies: "push" (Figure 3 b)) or "pull" (Figure 3 c)). In the "push" model, the Master decides which Worker should process the next job, while in the "pull" model Workers themselves ask the Master for jobs to be processed. The "pull" model naturally balances load among Workers and simplifies worker management. The "push" model, on the other hand, gives the Master more control over how jobs are assigned to workers which is important in the case where sophisticated job scheduling algorithms are used.

4.2 Workflow Persistence and Recovery

We propose a novel approach to workflow persistence and recovery based on event sourcing. The event sourcing model for workflow persistence boils down to defining the events that need to be logged in order to enable the full restoration of the workflow state. The following information should be persisted during workflow execution:

1. A workflow activity has sent/received a message.
2. A workflow activity has received message acknowledgment.
3. Results of performing data transformations.
4. Several additional events needed to save the state of Activity Master and Workers.

While the logging of these events ensures that the state of the workflow engine can be correctly restored, it will not reconstruct the internal state of stateful workflow activities (e.g. values accumulated across multiple firings). This can also be accomplished by adapting standard mechanisms of an actor framework. The developer needs to do the following in order to enable state recovery of a stateful workflow activity:

1. define domain-specific events and explicitly invoke a persist interface in the processing logic subroutine of that activity;
2. implement a recovery procedure that transforms domain events into updates of the activity's internal state.

4.3 Fault Tolerance

Actor frameworks typically adopt the *supervision* approach to fault tolerance, in which parent actors are the supervisors of their child actors. When an exception is thrown, the actor in which it occurred is suspended, while the exception is reported to its supervisor. The supervisor defines *fault tolerance strategies* that specify what actions should be taken for a given exception type. These actions are typically as follows: *Resume* the actor (preserving its state), *Restart* the actor (resetting its state), *Stop* the actor permanently, and *Escalate* the error to another (parent) supervisor.

This model of fault tolerance fits workflows very well, in particular scientific workflows which perform "dangerous" tasks that are prone to failure, such as resource-intensive processing, or invocations of external services. Figure 4 presents the supervision hierarchy proposed for a scientific workflow system. Every workflow activity has an *Activity Master* that is responsible for receiving messages and invoking the processing logic. However, the actual processing is performed not by the Activity Master, but it is delegated to a new Worker actor. Consequently, in the case of an error during processing only the Worker is affected. Multiple workers can be started at the same time, enabling parallel processing of input messages (see Section 4.1), or replication of tasks as an additional means of fault tolerance. Finally, all Workflow Activities are supervised by a Workflow Master actor.

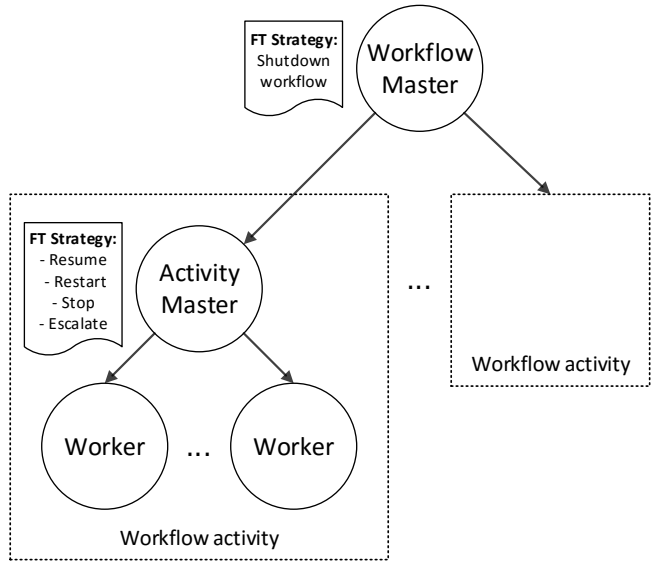


Figure 4. Supervision hierarchy for workflows

We propose the following semantics of fault tolerance strategies for scientific workflow activities:

Resume: continue execution of the worker, ignoring the error.

Restart: restart the worker and retry the operation, processing the same input messages again.

Stop: discard the messages that triggered the firing, restart the worker.

Escalate: report the error to the Workflow Master.

Table 4 presents examples of different failure categories and their respective fault tolerance strategies. Which strategy is the best will depend on the particular case and the nature of the failure. If the failure is transient and non-critical, the proper strategy might be simply to resume the worker. For example, a non-essential service might be temporarily unavailable, but the fundamental computations can still be performed without it, so there is no need to delay the processing. However, if the temporarily unavailable service is critical, the best strategy is to restart the worker in hope that the lost connection will be restored and the operation can be successfully completed.

Permanent failures, in contrast to transient ones, cannot usually be fixed simply by retrying the processing. Consequently, they require different fault tolerance strategies. If the failure is permanent but non-critical, we may decide to stop the worker and simply drop the messages that caused the error. For example, if the workflow processes a large collection of files and one of them is corrupt, we may decide to ignore it rather than fail the entire workflow because of one file. In the case of a critical permanent failure, on the other hand, the best strategy to follow might be to stop the workflow, fix the fault that caused the failure (e.g. free some disk space), and start the workflow again, recovering its state from before the failure.

Supervisor	Failure	Fault Tolerance Strategy
Activity Master	Transient non-critical failure (e.g. unavailability of a non-essential service)	**Resume** worker (ignore the error)
	Transient critical failure (e.g. lost connection)	**Restart** worker (retry the computation)
	Permanent non-critical failure (e.g. corrupt file)	**Stop** worker, drop input messages that caused the error
	Permanent critical failure (e.g. no disk space)	**Escalate** to Workflow Master
Workflow Master	Any	Shutdown workflow (perhaps for a later recovery after the failure has been fixed)

Table 4. Fault tolerance strategies for workflows

One of the biggest advantages of the presented approach is that fault tolerance strategies are **user-defined**, giving the workflow programmer unprecedented control over how failures are handled. Strategies can be specified not only for each workflow activity, but also for specific exceptions that may occur in that activity.

The workflow developer can decide which exceptions are critical, and which are not, and apply the appropriate strategy accordingly.

5 PRACTICAL EVALUATION: THE SCAFLOW SYSTEM

In this chapter, we demonstrate practical evaluation of the presented concepts using our prototype implementation of an actor-based workflow programming library and engine – Scaflow. The system was implemented in the Akka framework.[2] Scaflow implements the actor-based workflow model (Section 3.3), parallel processing architecture (Section 4.1) workflow persistence and recovery mechanism (Section 4.2), and the fault tolerance model (Section 4.3). In addition, Scaflow provides a library with high-level API for programming workflows.

5.1 Programming Interface

Intuitive application programming interface for workflows is arguably as important as the internal implementation of the workflow engine. One of the conclusions drawn from the analysis presented in Section 3.2 is that standard actor APIs, such as the one found in Akka, are not appropriate programmatic abstractions for this purpose. We should therefore seek a more proper API that would enable the developer to compose workflows in a convenient way.

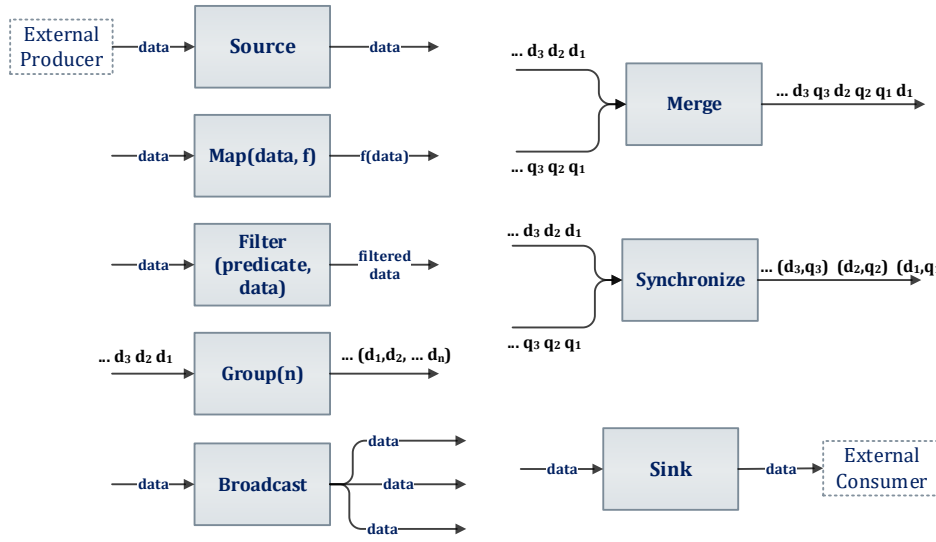


Figure 5. Workflow components in the Scaflow workflow API (visualization)

[2] `http://akka.io`

A comprehensive design of a workflow API or, better, a workflow DSL (domain-specific language), deserves a separate research and is out of scope of this work. However, we have designed a simple API that resembles the programming interface of Scala collections. The API features several basic workflow components that can be used to compose workflows. They are visualized in Figure 5.

Source component represents the starting point of a workflow. It receives data from an external source and feeds it to a workflow.

Map component is the main data transformation activity where processing logic is implemented.

Filter component filters the input data sequence according to a predicate.

Group component consumes a sequence of input data elements and produces n-element collections thereof.

Broadcast component replicates the input data sequence into all its output channels.

Merge component consumes data elements from multiple inputs and produces them as a single output sequence in the order of arrival, effectively implementing a *non-deterministic merge* semantics.

Synchronize component with n inputs waits for a data element on each of them and produces an n-element collection of these elements.

Sink component represents the final activity of a workflow which can pass the results to an external consumer.

5.2 Workflow Programming Example

In this section, we present a simple workflow example in order to illustrate the Scaflow API for composing workflows, customization of the fault tolerance strategy, and using remote workers. The workflow, shown visually in Figure 6 a), produces information about biological pathways related to specific genes. To this end, workflow activities invoke external REST services of the KEGG database.[3] The input to the workflow is a sequence of gene identifiers. For each of them, all related biological pathways are retrieved from the KEGG database (*GetPathwaysIds* activity). The result is split into a sequence of pathway identifiers which are fed in parallel to the next two activities of the workflow (*GetPathwayImage*, and *GetPathwayDescription*). These activities retrieve visual and textual descriptions of a pathway, respectively, and save them as files.

The implementation of the workflow is shown in Listing 1 and visually in Figure 6 b). The first part defines the fault tolerance strategy. The main problem that may occur is the loss of a connection to the REST services of the KEGG

[3] Kyoto Encyclopedia of Genes and Genomes, `http://www.genome.jp/kegg/kegg1.html`

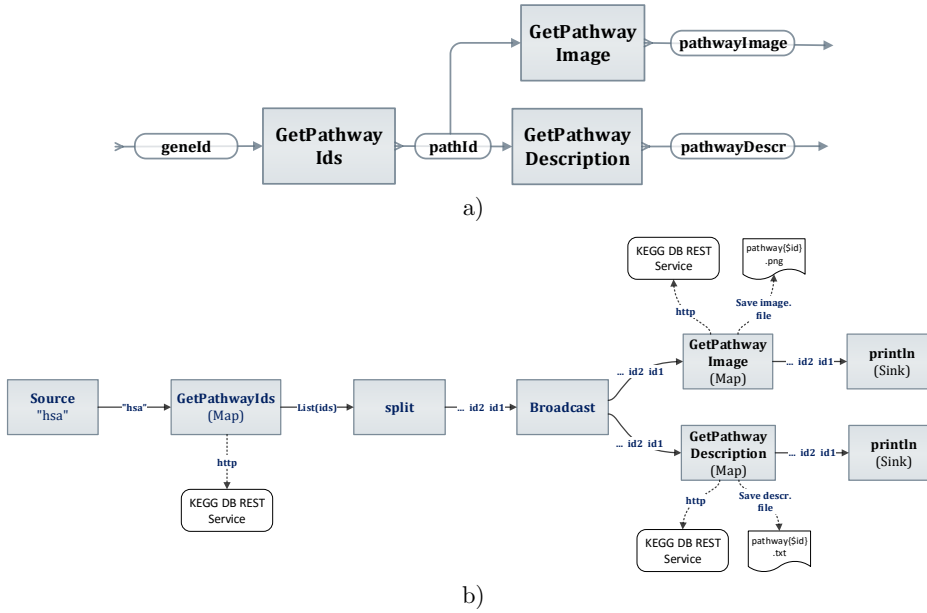


Figure 6. Biological pathways workflow (visualization): a) Conceptual flow, b) Scaflow implementation

database. Consequently, the strategy specifies that in the case of a *ConnectionProblemException*, the worker actor should be restarted (with a limit of 10 retries within 1 minute, otherwise the exception is escalated to the Workflow Master supervisor) in order to retry to establish the connection to the remote server. Another commonly occurring problem is an input error, such as an incorrect gene identifier. In this case, we do not want one bad input to cause a failure of the entire workflow, so we simply apply the *Stop* strategy in order to drop the message and restart the worker, thereby discarding the input that caused the error.

The remaining part of the file composes the workflow. The two parallel branches with activities *GetPathwayImage* and *GetPathwayDescription* are created first. The actual processing logic of these activities is implemented in functions *getPathwayMapPng* and *getPathwayDetails* (not shown). The final section of the code creates the first activity and connects the two others via the *broadcast* construct, so that they are executed in parallel. Note the "connector" API which simply allows the developer to create individual subflows separately and subsequently compose them into a bigger workflow.

All workflow components are instantiated via the *PersistentWorkflow* interface which implicitly enables logging of the workflow state changes. The extra argument passed to the persistent workflow API functions is a unique persistence identifier. The example also shows how to setup a distributed workflow with workers on remote nodes. We assume that remote actor systems are already deployed and running on

Listing 1. Implementation of the biological pathways workflow.

```
1  val HTTPSupervisorStrategy = AllForOneStrategy(
2                               maxNrOfRetries = 10,
3                               withinTimeRange = 1.minute) {
4      case _: BadInputException ⇒ Stop
5      case _: ConnectionProblemException ⇒ Restart
6      case _ ⇒ Escalate
7    }
8
9   val remoteAddress = Seq(AddressFromURIString
10      ("akka.tcp://remoteActorSystem@localhost:5150"))
11
12 implicit val actorSystem = ActorSystem("kegg")
13 val getPathwayImage =
14   PersistentWorkflow.connector[String]("pngConnector")
15     .map(
16       "getPng",
17       getPathwayMapPng,
18       Some(HTTPSupervisorStrategy),
19       workersNumber = 4, addresses = remoteAddress
20     )
21     .sink(
22       "sinkPng",
23       id ⇒ println(s"PNG map downloaded for $id")
24     )
25
26 val getPathwayDescription =
27   PersistentWorkflow.connector[String]("textConnector")
28   .map(
29       "getTxt",
30       getPathwayDetails,
31       Some(HTTPSupervisorStrategy)
32     )
33   .sink(
34       "sinkTxt",
35       id ⇒ println(s"TXT details download for pathway $id")
36     )
37
38 PersistentWorkflow
39   .source("source",List("hsa"))
40   .map("getPathways",
41                 getPathwayIds,
42                 Some(HTTPSupervisorStrategy)
43   ).split[String]("split")
44   .broadcast("broadcast", getPathwayImage,
45               getPathwayDescription)
46   .run
```

distributed nodes (the way this is achieved is an important technical issue, however, it is out of scope of this paper). Once this is done, the programmer only needs to specify the remote address of an actor system (line 10), and pass it to a "map" activity (line 20). As a result, the worker actor will be created in the remote actor system and all communication with it will be redirected there. Remote workers have some limitations, namely the operation invoked by the "map" activity needs to be serializable, and its code must be available both on the master node and the remote node.

5.3 Persistence Overhead

Scaflow allows one to create workflows with either enabled or disabled persistence. In this section, we investigate the overhead due to enabled persistence and saving all state changes to external storage. We measure this overhead by building and running the same workflow in two versions: with and without persistence. For conducting the performance test, we have created a simple test workflow (Listing 2), composed of three components: source, filter and sink. Total execution time was measured for both configurations. We have tested three different storage alternatives: the default LevelDB[4] key-value database, the Kafka[5] distributed messaging system, and the Cassandra database.[6]

Listing 2. Test workflow implementation

```
val standardWorflow = StandardWorkflow.source(1 to N)
        .filter(_ < Int.MaxValue)
        .sink(timeMeasureActor)

val persistentWorkflow = PersistentWorkflow.source("source", 1 to N)
        .filter("filter", _ < Int.MaxValue)
        .sink("sink", timeMeasureActor)
```

Results are presented in Table 5 and in Figure 7. As we can see, the total execution time increases at least two to three times when the persistence is enabled. The default storage system LevelDB induces a huge overhead and is even 80 times slower than the Cassandra database system. However, we should note that the test workflow did not contain any CPU-bound processing. In reality, such processing dominates in scientific workflows, so in the case of the most efficient backend tested, Cassandra, even the highest persistence overhead measured for 50 000 messages ($3s$) will be negligible in comparison to the total workflow execution time.

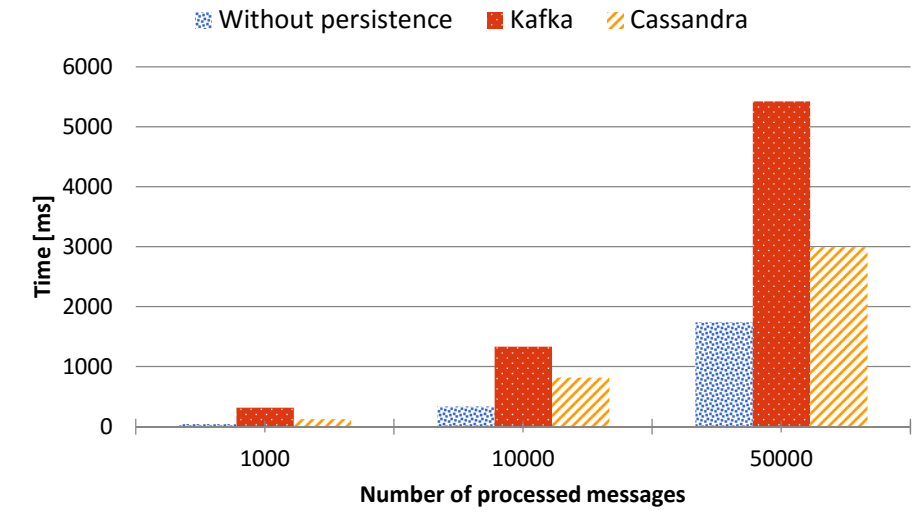
	Execution Time (ms)			
Message Count	No persistence	Kafka	Cassandra	LevelDB
1 000	43 ± 11	317 ± 28	125 ± 25	$1\,176 \pm 165$
10 000	337 ± 8	$1\,334 \pm 248$	816 ± 150	$11\,218 \pm 434$
50 000	$1\,739 \pm 30$	$5\,423 \pm 685$	$2\,989 \pm 406$	$243\,320 \pm 29\,946$

Table 5. Total workflow execution time in different persistence configurations

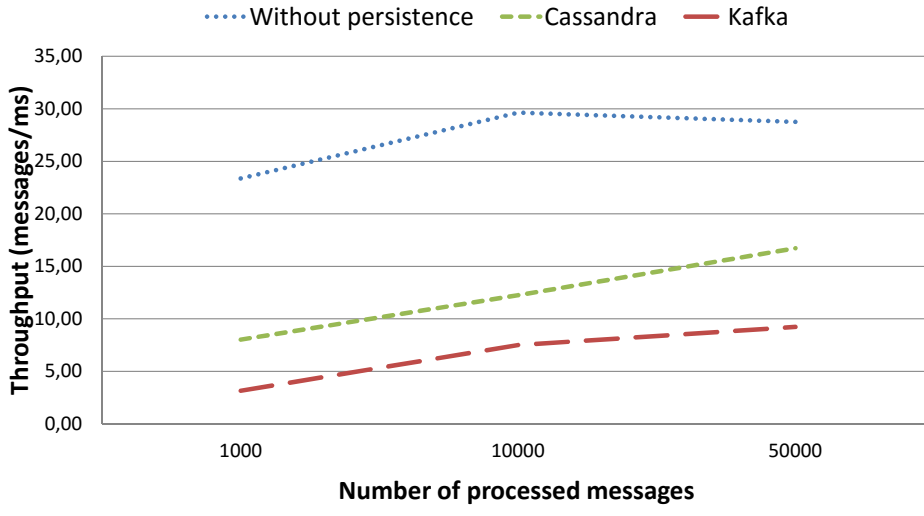
6 CONCLUSIONS

Actor-based frameworks can improve the design and execution of scientific workflows in several aspects:

[4] `https://github.com/google/leveldb`
[5] `http://kafka.apache.org/`
[6] `http://cassandra.apache.org/`



a)



b)

Figure 7. Workflow persistence overhead evaluation in different configurations: a) Workflow execution time, b) Message throughput

1. *Paralel and distributed computing*: workflow systems need to rely on a complex infrastructure/middleware to enable distributed execution in scientific workflows. An actor framework already provides an infrastructure for transparent distributed execution, so that only deployment of actor systems on distributed nodes needs to be addressed.
2. *Persistence and recovery*: scientific workflows benefit from event sourcing, the persistence and recovery approach used in actor systems. Compared to checkpointing, event sourcing is

   (a) faster: we have measured the persistence overhead and it proved that, unlike in checkpointing, every state change in workflow activities can be persisted in real time without substantially affecting the workflow orchestration performance;
   (b) more powerful: workflow state can be restored even for stateful activities, without the need of their re-execution; no existing workflow system known to us achieves that.

3. *Fault tolerance*: the supervision fault tolerance model typical for actor frameworks enables workflow developers to customize fault tolerance strategies for individual workflow activities and specific faults that may occur in a particular case.

Future work involves further research and experiments in such topics as smart re-runs of a workflow, as well as development of the Scaflow system, in particular improvements in the workflow API.

Acknowledgments

This work is supported by the AGH statutory research grant No. 11.11.230.124.

REFERENCES

[1] Deelman, E.—Gannon, D.—Shields, M.—Taylor, I.: Workflows and e-Science: An Overview of Workflow System Features and Capabilities. Future Generation Computer Systems, Vol. 25, 2009, No. 5, pp. 528–540.

[2] Agha, G.—Hewitt, C.: Concurrent Programming Using Actors. Object-Oriented Concurrent Programming, MIT Press, 1987, pp. 37–53.

[3] Karmani, R. K.—Shali, A.—Agha, G.: Actor Frameworks for the JVM Platform: A Comparative Analysis. Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, ACM, 2009, pp. 11–20.

[4] Haller, P.: On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective. Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions, ACM, 2012, pp. 1–6.

[5] Cushing, R.—Koulouzis, S.—Belloum, A. S. Z.—Bubak, M.: Prediction-Based Auto-Scaling of Scientific Workflows. Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science (MGC'11), ACM, 2011, Art. No. 1.

[6] Balis, B.: Hyperflow: A Model of Computation, Programming Approach and Enactment Engine for Complex Distributed Workflows. Future Generation Computer Systems, Vol. 55, 2016, pp. 147–162.

[7] Kacsuk, P.—Farkas, Z.—Kozlovszky, M.—Hermann, G.—Balasko, A.—Karoczkai, K.—Marton, I.: WS-PGRADE/gUSE Generic DCI Gateway Framework for a Large Variety of User Communities. Journal of Grid Computing, Vol. 10, 2012, No. 4, pp. 601–630.

[8] Balasko, A.—Farkas, Z.—Kacsuk, P.: Building Science Gateways by Utilizing the Generic WS-PGRADE/gUSE Workflow System. Computer Science Journal, Vol. 14, 2013, No. 2, pp. 307–325.

[9] Kozlovszky, M.—Karóczkai, K.—Márton, I.—Kacsuk, P.—Gottdank, T.: DCI Bridge: Executing WS-PGRADE Workflows in Distributed Computing Infrastructures. Science Gateways for Distributed Computing Infrastructures, Springer, 2014, pp. 51–67.

[10] Deelman, E.—Vahi, K.—Juve, G.—Rynge, M.—Callaghan, S.—Maechling, P. J.—Mayani, R.—Chen, W.—da Silva, R. F.—Livny, M. et al.: Pegasus, a Workflow Management System for Science Automation. Future Generation Computer Systems, Vol. 46, 2015, pp. 17–35.

[11] Oinn, T.—Li, P.—Kell, D. B.—Goble, C.—Goderis, A.—Greenwood, M.—Hull, D.—Stevens, R.—Turi, D.—Zhao, J.: Taverna/myGrid: Aligning a Workflow System with the Life Sciences Community. In: Taylor, I., Deelman, E., Gannon, D., Shields, M. (Eds.): Workflows for e-Science. Springer, New York, Secaucus, NJ, USA, 2007, pp. 300–319.

[12] Ludäscher, B.—Altintas, I.—Berkley, Ch.—Higgins, D.—Jaeger, E.—Jones, M.—Lee, E. A.—Tao, J.—Zhao, Y.: Scientific Workflow Management and the Kepler System. Concurrency and Computation: Practice and Experience, Vol. 18, 2006, No. 10, pp. 1039–1065.

[13] Płóciennik, M.—Żok, T.—Altintas, I.—Wang, J.—Crawl, D.—Abramson, D.—Imbeaux, F.—Guillerminet, B.—Lopez-Caniego, M.—Plasencia, I. C. et al.: Approaches to Distributed Execution of Scientific Workflows in Kepler. Fundamenta Informaticae, Vol. 128, 2013, No. 3, pp. 281–302.

[14] Qin, J.—Fahringer, T.: Scientific Workflows: Programming, Optimization, and Synthesis with ASKALON and AWDL. Springer, 2012.

[15] Tolosana-Calasanz, R.—Bañares, J. Á.—Álvarez, P.—Ezpeleta, J.—Rana, O.: An Uncoordinated Asynchronous Checkpointing Model for Hierarchical Scientific Workflows. Journal of Computer and System Sciences, Vol. 76, 2010, No. 6, pp. 403–415.

[16] Podhorszki, N.—Ludaescher, B.—Klasky, S. A.: Workflow Automation for Processing Plasma Fusion Simulation Data. Proceedings of the 2nd Workshop on Workflows in Support of Large-Scale Science, ACM, 2007, pp. 35–44.

[17] Köhler, S.—Riddle, S.—Zinn, D.—McPhillips, T.—Ludäscher, B.: Improving Workflow Fault Tolerance Through Provenance-Based Recovery. Scientific and Statistical Database Management, Springer, 2011, pp. 207–224.

[18] Kahn, G.: The Semantics of a Simple Language for Parallel Programming. Information Processing, North-Holland, 1974, pp. 471–475.

[19] van Der Aalst, W. M. P.—Ter Hofstede, A. H. M.—Kiepuszewski, B.—Barros, A. P.: Workflow Patterns. Distributed and Parallel Databases, Vol. 14, 2003, No. 1, pp. 5–51.

[20] Hulette, G. C.—Sottile, M. J.—Malony, A. D.: WOOL: A Workflow Programming Language. IEEE Fourth International Conference on eScience (eScience '08), 2008, pp. 71–78.

[21] Garijo, D.—Alper, P.—Belhajjame, K.—Corcho, O.—Gil, Y.—Goble, C.: Common Motifs in Scientific Workflows: An Empirical Analysis. Future Generation Computer Systems, Vol. 36, 2014, pp. 338–351.

Bartosz Balis received his Ph.D. degree in computer science. He is Assistant Professor at the Department of Computer Science, AGH University of Science and Technology, Krakow, Poland. He has (co-)authored more than 75 international publications including journal articles, conference papers, and book chapters. His research interests include environments for eScience, scientific workflows, grid and cloud computing. He participated in international research projects including EU-IST CrossGrid, CoreGRID, K-Wf Grid, ViroLab, Gredia, UrbanFlood and PaaSage. He is a member of Program Committee for many conferences (e-Science 2006, ICCS 2007-2016, ITU Kaleidoscope 2013-2016, SC16 Workshops, SIMULTECH 2016).

Krzysztof Borowski received his M.Sc. degree in computer science in 2016 from AGH University of Science and Technology, Krakow, Poland. He is a Scala programmer with extensive professional experience around JVM. He is focused on modern, statically typed languages and libraries designed for rapid feature development, with a strong emphasis on scalability and code maintainability.