RCourse: A Robustness Benchmarking Suite for Publish/Subscribe Overlay Simulations With Peersim

Tobias R. Mayer, David Coquil, Christian Schoernich, Harald Kosch
Department of Distributed Information Systems
University of Passau, Germany
{tobias.mayer, david.coquil, harald.kosch}@uni-passau.de,
schoerni@fim.uni-passau.de

ABSTRACT

This paper introduces the RCourse benchmarking suite, an extension to the Peersim simulatior environment. RCourse supports simulative evaluations of Publish/Subscribe systems with respect to robustness. To this end, it provides among others mechanisms for the aggregation of measurement values and for an automatic graph generation representing the extracted results. The design of RCourse is characterized by its highly modular architecture, which enables the adaptation of each step of the simulation workflow to specific user needs.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]: Fault Tolerance, Measurement Techniques

Keywords

simulation, reliability, overlay networks, Publish/Subscribe

1. INTRODUCTION

Publish/subscribe (pub/sub) [**?**] systems realize multicast at the application layer. This type of interaction is indeed well suited to the needs of an event-based middleware, as it is characterized by a full decoupling in time, space and **âĂć**. Pub/sub systems typically base on a logical overlay built on top of the network layer. Among the options for constructing these overlays, peer-to-peer (P2P) based overlay networks have gained increasing attention in research and system development in recent years. When such systems are deployed in large-scale, the risk of failure of participating nodes increases strongly. Such failures may critically affect the functioning of the global system [**?**]. Therefore, the robustness of P2P systems, i.e. the degree of correct functioning in presence of erroneous conditions [**?**], should be taken into account and evaluated when designing a P2P-based pub/sub system. A good option to this end is the Peersim simulator[1] [**?**], which has been developed for the simulative study of overlay networks. Indeed, omitting the simulation of the network stack renders Peersim particularly well-suited for the design of such studies. Moreover, Peersim is appropriate for large-scale experimentations.

To the best of our knowledge, there is no extension for statistical result aggregation and analysis available for Peersim. Thus, performing network simulations requires this additional work to obtain practically usable results. Moreover, no benchmarking application for Peersim specifically targeting robustness is available. In this context, this paper introduces RCourse, an extension to the Peersim simulator. Designed as a combination of a programming library, analysis scripts and pre-defined simulation scenarios, it aims to support the user in terms of simulative studies. Thus, RCourse saves work and time for system designer while facilitating consistent performance evaluation of systems. RCourse is specially tailored to studies on pub/sub systems, however, it is designed to be easily adaptable to simulative evaluations of P2P systems in general.

The RCourse prototype is described in more detail in the remainder of this paper. The next section **??** describes related work. Section **??** gives an overview of the design goals and components of RCourse. Then, section **??** further details the use of RCourse for simulations with Peersim as well as the adaptation to record additional user-defined measurement values. Finally, the paper closes with a conclusion in section **??**.

2. RELATED WORK

The first Publish/Subscribe systems (e.g. [**?**, **?**]) provided a structured dissemination (a deterministic algorithm calculates static dissemination structure) and targeted mainly core aspects such as subscription type (e.g. topic-, content-based) or overlay organization. Later works (e.g. [**?**, **?**]) introduced also unstructured systems (non-deterministic algorithms make use of some randomness, realizing a dynamic dissemination structure), which are inherently more suited to dynamic environments. Several surveys reviewed pub/-sub systems and its routing (e.g. [**?**, **?**]).

Introduced in 2009 [**?**], the Peersim P2P simulator has been widely used by the research community. This is outlined by the list of around 140 publications provided by the authors in their website, which includes papers published in major conferences and journals (see 'Publications' link on the Peersim website). Peersim has been used for simulative

[1]http://peersim.sourceforge.net/

evaluations of several pub/sub systems, e.g. [**?**, **?**].
In a previous paper [**?**], we have shown that the few works dealing with the robustness of pub/sub environments fail to properly characterise existing systems. We argue that this is in particular due to the lack of proper benchmarking tools. Considering for example the case of Peersim, despite a large body of extensions and additional libraries, no such tool is available to the best of our knowledge; moreover, tools to support the results analysis of simulation-based evaluation are also missing. This clearly shows the need for a more standardised approach supported by dedicated software.

3. RCOURSE OVERVIEW

In this section, we introduce the RCourse benchmarking suite by describing its design goals, general functioning and architectural components.

3.1 Design Goals

For the development of the RCourse simulation library, we defined several design goals:

(i) *Time/work savings*: As many tasks as possible shall be automated in order to spare work/time to the user.

(ii) *Architectural modularity*: Each step of a simulative study shall be adaptable to specific user needs.

(iii) *Stand-alone simulations in terms of robustness and performance*: The RCourse library shall enable complete simulative studies in terms of robustness and performance without requiring other software products or further calculations.

(iv) *Free tool for research community*: RCourse shall be designed as a free tool, i.e. the library itself (as well as further software) shall be freely available under an open-source license.

Let us now consider how these goals are reached in the prototype. In order to save time and work for the user, RCourse aims to support each step of the simulation workflow (goal (i)). This can be seen in figure **??**, which describes the global workflow of a simulative study with steps supported by RCourse being placed in the middle row (continuous lines, white background). For example, RCourse provides means for data aggregation and write-out in the form of Java classes, a result analysis functionality by means of R scripts, and can automatically generate result graphs as pdf files. Thus, the user can concentrate on the development of the algorithm to be evaluated by the simulation, as all other steps may be taken over by RCourse components.
The design goal (ii) is reached by providing a support for each step in the workflow independently from the other ones. This loose coupling in the workflow results in a modular architecture and facilitates the adaptation of the different steps to specific needs (possible adaptations are indicated by dashed boxes in figure **??**).
The RCourse library is complete (goal (iii)) in the sense that a simulative study can be directly performed using Peersim and RCourse. In particular, no further specifications such as scenario definitions are needed: pre-defined scenarios are provided. To reach this goal, two pub/sub systems have been developed for use with RCourse. They are bundled with the library and serve as reference implementations. The first system is Scribe [**?**], representing a DHT-based approach. The Scribe implementation is based on the Pastry implementation of M. Cortella, which is also seperately available on the Peersim homepage. The second system is a gossiping system, which realizes a clique network with its non-deterministic dissemination. The gossiping system is developed as basic and informed gossiping and comes with Cyclon [**?**] and HyParView [**?**] as membership protocols.
RCourse has been developed in order to ease simulative studies of overlay networks (especially pub/sub systems), primarily targeting the research community. Thus, in order to fulfill design goal (iv), it makes only use of free software – indeed, this is only Peersim and the programming language R[2] – and is itself published as open source under the GPLv2 license. The RCourse library as well as additional files (scenario/graph overview, example result files etc.) are available for download on the RCourse project homepage hosted at sourceforge[3].

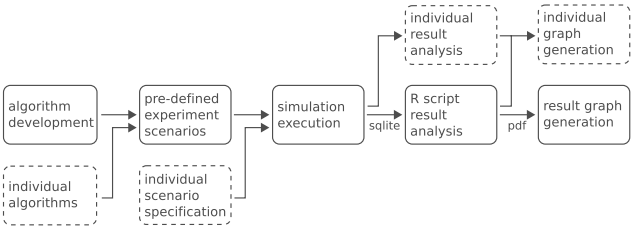

Figure 1: Workflow of simulative studies supported by RCourse.

3.2 Overview of RCourse Components

General Overview. RCourse is realized as a loose coupling of three components:

a) *A set of Java classes* supports the user in collectiong measurements values and writing them out to SQLite result files (optionally: to csv files).

b) *Pre-defined simulation scenarios* provided as Peersim configuration files can be used directly to simulate and evaluate a system after finishing the development of the algorithm.

c) *Analysis scripts* written in the R programming language aggregate the SQLite result files, analyse the result data and generate result graphs as pdf files.

A simulation that makes full use of these components is completely reduced to the development of the algorithm. During this development (or the adaptation of an already existing algorithm), the only additional step that is required is to use the RCourse API for recording measurement values. Simulations can then be executed using the pre-defined scenarios and result graphs created by means of R analysis scripts.

[2]http://www.r-project.org
[3]http://rcourse.sourceforge.net

Table 1: RCourse standard scenarios are based on the taxonomy of faults presented in [?].

Fault Type	Experimentation Scenario	
no faults	ES1.1	Standard Operation
	ES1.2	Standard Operation (Network Size Variation)
	ES1.2	Standard Operation (Fanout Variation)
Fault A Link/Node Crash	ES2.1	Catastropic Node Crash
	ES2.2	Catastropic Node Crash (Crash Size Variation)
Fault B Message Loss	ES3.1	Catastropic Message Loss
	ES3.2	Catastropic Message Loss (Malicious Nodes Variation)
	ES3.3	Message Loss
	ES3.4	Message Loss (Message Loss Variation)
Fault C Message Tampering	ES4	Message Tampering
Fault D Message Generation	ES5	---
Fault E Node Churn	ES6.1	Node Churn
	ES6.2	Node Churn (no publish)
	ES6.3	Node Churn (with publish)
	ES6.4	Node Churn (Churn Rate Variation, no publish)
	ES6.5	Node Churn (Churn Rate Variation, with publish)
Fault F Information Leak	ES7	---
Fault G Selfish Behaviour	ES8.1	Selfish Message Loss
	ES8.2	Selfish Message Tampering

RCourse aims especially to easily let the users define their own measurement values to be recorded during the simulation; therefore, the needed work for the adaptation has been strongly reduced. Section **??** presents the essential steps to adapt RCourse to user-defined values. An overview of the RCourse scenarios is given in table **??** (a complete list of possible result graphs can be found on the project homepage). The scenarios are based on our previous work [**?**], which provides a taxonomy of elementary faults for peers in a pub/sub system and discusses robustness issues for pub/-sub from a comprehensive point of view. Two entries of the taxonomy are however ignored: fault D (Message Generation) considers the injection of protocol-coherent messages with arbitrary content and F (Information Leak) deals with the privacy in terms of protecting the message content from prohibited reading. The reason for not considering these faults is that they can be avoided by means of techniques such as access control or cryptography mechanisms.
Although the scenarios are derived from further robustness studies, they are rather simplistic and serve as proposals for experimentations. For representative results, the use of real-world workload and failure traces is recommended, e.g. those of the Failure Trace Archive[4]. This is currently not yet supported by RCourse but indeed an interesting possibility for future work. Thanks to the architectural modularity, an implementation is facilitated (only the *RCTrafficGenerator* as well as *RCTurbulenceGenerator* need to be adapted, which are outlined in the next paragraph).

Main Components. The Java classes of RCourse mainly provide six essential components to support the user in the development of a simulative study (figure **??**). The class *RCParamStorage* holds simulation-wide parameters for RCourse and is accessible by the peers. These parameters include the configuration of the simulation as well as other global values. *RCTrafficGenerator* and *RCTurbulenceGenerator* are helper classes for the simulation. *RCTrafficGenerator* generates an appropriate workload. *RCTurbulenceGenerator* creates different stressful situations such as crashing nodes and node churn. Other types of failures, for example message loss, are realized by configuration file parameters in combination with the peer's implementation. The workload generation is realized by an injection of messages through the pub/sub service. Therefore, *RCTrafficGenerator* requires the implementation of a pub/sub protocol interface (see the file *util/PubSubProtocol.java*). The workload generation is based on a foregoing empirical study and aims to focus on the corresponding fault situation. For example, the publishing phase is started after a subscription phase for the study of node crash impact, while subscriptions and publishing of messages are done simultaneously for the node churn scenarios. For more details please refer to the Peersim simulation scenarios of RCourse. The class *RCStatsCont* represents the container for measurement values. Each peer of the network must hold one instance of this class; the algorithm implementation must add/update the values that are to be recorded. In this context, the *RCObserver* plays a key role in RCourse. This class collects at specified time points all measurement containers and passes them (after some processing) to the *RCResultWriter*. This class extends *RCResultWriterBase* (not shown in figure **??**). It generates the SQLite result file as well as the csv file output.
When developing the algorithm, the user only has to consider *RCStatsCont* and *RCParamStorage* in his source code. The interaction with the other classes is specified in the configuration files, and an interface (implementing for example basic pub/sub operations) must be provided. For more details, please refer to the reference implementations of RCourse, which are available on the project's website.

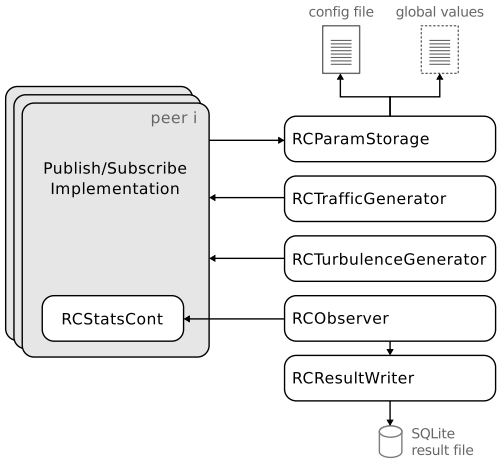

Figure 2: Architectural overview of RCourse shows the interaction with a pub/sub system.

4. SIMULATIONS WITH PEERSIM AND RCOURSE

[4]http://fta.inria.fr/apache2-default/pmwiki/index.php

```
public void publish(String groupid, Object content) {
            ...
   msgStats.increaseMsgSentCounter_Notify();
   mypastry.send(Util.strToPastryID(groupid), sdm);
            ...
}
```

```
private void deliverToChildren(ScribeDataMessage sdm,
                               boolean deliverToMyself) {
            ...
   // delivery to upper layer
   if (rlistener != null){
      rlistener.receive(sdm);
   }
   msgStats.getMsgDeliveryHops().add( sdm.deliveryHops );

   // set information for delivery percentage calculation
   if( ps.isComplexDlvPrctCalc() ) {
       // complex delivery calculation
       // (more precise, for dynamic networks)
       msgStats.getDeliveredList().add( new MLEntry(sdm) );
   } else {
      // simple delivery (# of receivers is known a priori)
      msgStats.increaseMsgCounter Delivered();
   }
            ...
}
```

Figure 3: Two examples of the additional RCourse code (italic/blue) to record measurement values in the RCourse data container object (RCStatsCont).

After a general overview of RCourse, we now more precisely describe its use for simulations with Peersim.

4.1 Utilization and Experimentation

In an optimal case in terms of time/work savings, a user can completely rely on RCourse after developing the algorithm. Then, the combination of pre-defined scenarios and analysis scripts enables the automatic generation of a broad range of result graphs. Here, we outline the use of RCourse through the steps of the simulation workflow of figure **??**.

Setting Up the RCourse Library. To set up the RCourse library, its folders must be copied into the Peersim project folder and referenced by the source code. In more detail, RCourse provides three important folders. The *src/* folder contains the example algorithms (scribe, gossiping). It is further divided into application and overlay. The folder *config/* contains the pre-defined scenarios and configuration files for the Peersim simulation and *analysis/* the R scripts for result analysis and graph generation.

Algorithm Development. A crucial task required to used RCourse is to extend the pub/sub algorithm to record measurement values. To this end, each peer must maintain a *RCStatsCont* container object and implement all method calls for value updates. Figure **??** shows the interaction with the data container for two examples with the Scribe system, showing the RCourse related code in blue/italic. The first example records the amount of published messages and the second example the dissemination time in terms of overlay hops as well as the delivery completion as a percentage. Note that RCourse provides two types of calculation for this value. The simple method uses the configuration parameters to calculate the number of packets that should be delivered. Here, publishing must start after a subscription phase (plus some stabilization time) to provide correct values. The complex method calculates delivery completion based on the current situation of subscriptions and published messages. Therefore, it is slower and computationally more intensive, but also more suitable for dynamic workload situations such as node churn. The two examples of figure **??** are only meant to outline the interaction with RCourse in order to collect measurement values. For the full spectrum of interactions with the statistical container of RCourse, we refer to the source codes of the exemplary applications to the Scribe and gossiping systems. As mentioned before, they come together with the RCourse source codes and serve as reference for implementation and library interaction.

Performing Simulations with RCourse and Peersim. Once the code of the algorithm has been extended with capabilities for recording metrics, a simulation with Peersim and RCourse is ready to be executed. To this end, RCourse requires *peersim.rangesim.RangeSimulator* as main class since several experimentation scenarios use value ranges for specific parameters. With this main class, the user only needs to specify the value range in the configuration file. The variable *range.malNodeP rcourse.malNodeProb;0:1|0.1* means for example, that the variable *range.malNodeP* varies from 0 to 1 with steps of 0.1. Thus, *RangeSimulator* performs 11 simulation, each one with a different parameter value, and independent from the other simulations. Once the main class has been specified, RCourse can be started with two configuration files as arguments plus the *rcourse.distrProcId* value, which is a parameter needed for multi-process simulations (see next section). Configuration files have been separated for ease of understanding. For example, a simulation with Peersim and RCourse for scenario ES8.1 (selfish message loss) is started with the following three arguments (note that - as an argument - the parameter and their values must be assigned by character '='):

- $config/RCourse\_Base.txt$
  This configuration file defines the general network setup and the structure of each peer's stack of protocols.
- $config/Scribe/ES8.1\text{-}SelfishMessageLoss.txt$
  The second config file represents the scenario and comprises all scenario-related parameters such as time steps for the observer execution or workload generation.
- $rcourse.distrProcId = 1$
  The distrProcId parameter defines the id for the current simulation in case of parallel processing. See section **??** for more information.

To clarify the meaning of the result file naming scheme (e.g. *ES1.1-StandardOperation_expRun6_DP1.sqlite*), let us precisely define the terms *simulation run* and *experiment repetition*. A simulation run is the execution of one simulation with a specific set of fixed parameters, whereas an experiment repetition consists of one or more simulation runs that are related to each other. In more details, simulation scenarios may be specified with parameter ranges (e.g. varying amount of malicious peers). This causes multiple simulation runs, which altogether represent one experiment repetition.

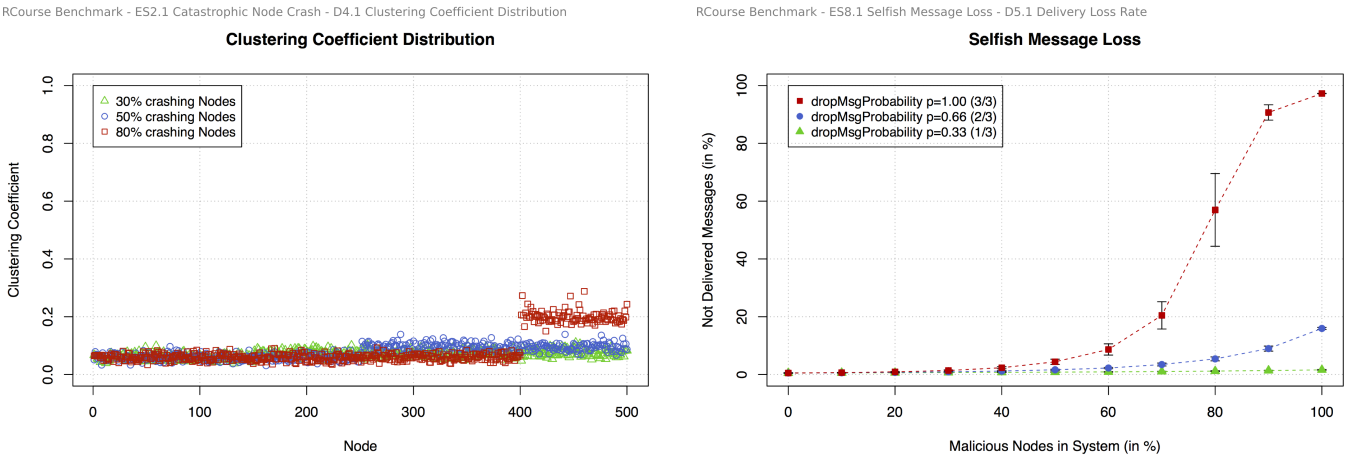

Figure 4: Examples of graphs generated by RCourse (results for basic gossiping with cyclon); more examples are available on the RCourse project's homepage.

Therefore, a scenario definition describes an experiment repetition, while the amount of repetitions can be defined with the *simulation.experiments* parameter. Each SQLite result file encapsulates the result values of one experiment repetition. Therefore, multiple simulations can be performed independently from each other (and even on different machines) to achieve more representative average simulation results. The R scripts then aggregate the result files for analysis and graph generation.

Result Data Analysis and Graph Generation. After a simulation with RCourse, i.e. once the SQLite database result files have been generated, the R scripts can be used for data analysis and result graph generation. To this end, RCourse provides a script for each simulation scenario. Each of these files (e.g. *analysis/diRgram_ES8.1.r*) calls the actual analysis scripts corresponding to the scenario. Furthermore, the configuration files in *analysis/config/* as well as *analysis/util/default_values.r* should be checked before starting the analysis. After a successful execution, the R scripts generate several result graphs. Figure **??** shows two examples of graphs generated by RCourse for basic gossiping with Cyclon. The left graph shows the (sorted) clustering coefficients of peers in case of a catastrophic node crash scenario (30%, 50% and 80% of the nodes crashing at once), the right one the impact of message drops due to the selfish goal of reduced resource usage (messages are dropped with probability of 0.33, 0.66 and 1.0).

4.2 A Note on Multi-Process Simulations

Modern processors have multiple cores and/or CPUs, a feature that should be used for simulations by implementing parallel processing. However, Peersim only allows the single-threaded execution of a simulation. To make use of the remaining processing power, RCourse enables parallel processing through parallel execution of independent simulations. For example, multiple simulation runs are processed simultaneously, which represent altogether one or more experiment repetition(s). Parallel processing can even take place on different machines since each result file independently stores all result data of one experiment repetition. This workaround makes sense since a simulation should usually be repeated a few times to achieve reliable results. Let us assume for instance that we have five cores at our disposal and a simulation that should be repeated ten times, i.e. ten overall experiment repetitions. Then, each core could perform two simulations sequentially (starting the second one after finishing the first). This can be realized by setting the *rcourse.distrProcId* and *simulation.experiments* parameters in the configuration file. The first parameter only affects the name of the result file (important for R analysis scripts), while *simulation.experiments* corresponds to a parameter of Peersim that defines the number of experiment repetitions for the corresponding execution of Peersim. This results in parallel processing on all cores and in the generation of ten SQLite result files. These files are then aggregated by the R scripts and used for result graph generation. The functioning of this multi-process example is shown in figure **??**.

4.3 Adaptation to User-defined Values

Out of the box, RCourse supports a wide range of measurement values, which are clearly arranged in the file *RCResultWriter.java*. In addition, it was specifically designed to enable users to easily define their own additional measurement values. To this end, they can use the RCourse data container. Then, only two components of RCourse have to be modified to let it record a new value.

First, the *RCStatsCont* class must be extended by the new value to be recorded. This may include writing appropriate methods to add data to the container object. Please note that the *RCStatsCont.mergeStats(...)* method may have to be modified as well depending on the type of value. This method is indeed called by the *RCObserver* to merge the statistical values of all peers into one data container with global values. Hence, it must be adapted if the new value considers network-wide values such as the average dissemination delay or average message transfers per time cycle.

The second component to modify is the *RCResultWriter* class, which is responsible for writing out all values. The required modification consists in adding the new value as an additional database column.

With these two extensions, RCourse is able to record and

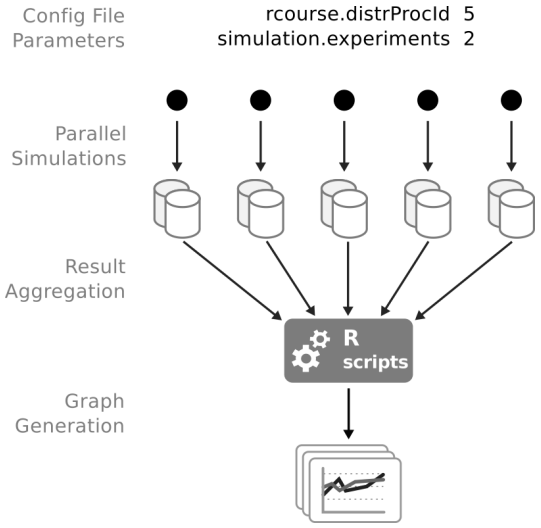

Figure 5: Even though Peersim is single-threaded, RCourse can be executed in a parallel way: each independent simulation is affected to its own process.

write out the new value. To make use of the R scripts for reading in, analysing and generating graphs, the *entrycols* (columns of a database entry) and/or the csv *fileheader* variable must be changed to the new value in the file $analysis/util/default\_values.r$. Finally, an appropriate R script can be adapted for data analysis and result graph generation of the new value.

5. CONCLUSION

This paper introduced RCourse, an extension library for the Peersim simulation environment. RCourse aims to support the user in each step of the simulation workflow in order to spare her/him time and work. The program particulary targets Publish/Subscribe systems. RCourse facilitates among others measurement value aggregation as well as data analysis, and also provides automated result graph generation. A simulative study can be directly launched once the core algorithm is available as other steps may be taken over by RCourse. The RCourse library is freely available as open-source software and comes with several pub/sub algorithms. In future versions, we plan to extend the software to support real-world workload and failures traces. This will be facilitated by RCourse's architectural modularity, which provides an easy adaptability to specific user needs in each step of the simulation workflow such as defining new simulation metrics.

6. ACKNOWLEDGEMENTS

This work was conducted in the framework of the Multimedia Distributed and Pervasive Secure Systems (MDPS) doctoral college[5], a French-German-Italian collaboration. Support was received by the Université Franco-Allemande (CDFA-05-08) and from the Région Rhône Alpes.

[5]http://mdps.dimis.fim.uni-passau.de