SimProcTC – The Design and Realization of a Powerful Tool-Chain for OMNeT++ Simulations[*]

Thomas Dreibholz, Erwin P. Rathgeb
University of Duisburg-Essen
Institute for Experimental Mathematics
Ellernstrasse 29, 45326 Essen, Germany

Xing Zhou
Hainan University
College of Information Science and Technology
Renmin Ave. 58, 570228 Haikou, Hainan, China

ABSTRACT

In this paper, we introduce our Open Source simulation tool-chain for OMNeT++ simulations: SimProcTC. This model-independent tool-chain has been designed to perform the common and frequently recurring tasks of simulation work – which are the parametrization of runs, the distributed run processing and the results visualization – in an efficient and easy to use manner. It is already successfully deployed for several OMNeT++-based research projects.

Keywords: Simulation, Parametrization, Run Distribution, Results Visualization

1. INTRODUCTION

OMNeT++ [1] is a powerful simulation package. During the past few years, we have applied it for multiple research projects. One of these projects is the research on Reliable Server Pooling (RSerPool), the IETF's new framework for server pool and session management to support load distribution and high availability. Our simulation model RSPSIM [2,3] provides about 120 parameters, so it had become a rather time-consuming task to parametrize simulations, process them and finally visualize their results. This challenge has lead to the development of our model-independent Open Source tool-chain SimProcTC ("Simulation Processing Tool-Chain") [2], which takes care of these tasks. SimProcTC is Open Source under GPLv3 license and freely downloadable from our web site[1]. Our tool-chain works with OMNeT++ version 3.x as well as the new version 4.0.

The goal of this paper is to introduce SimProcTC, with focus on using this tool-chain as a basis for own simulation processing. An overview of SimProcTC can be found in figure 1: its core is a GNU R [4] script which parametrizes simulations runs (section 2). Using GNU Make, the runs are processed (section 3) – either on the local machine or in an RSerPool-based computation pool. The post-processing stage prepares the results for their visualization (section 5).

Up until now, SimProcTC has been successfully deployed for research on RSerPool using the RSPSIM model (e.g. the papers [5–12]), for examining Quality of Service (QoS) enhancements by flow routing (e.g. the papers [13, 14]), for evaluating the SCTP protocol using [15] as well as for simulating sensor networks.

[*]Funded by the German Research Foundation (Deutsche Forschungsgemeinschaft) and the State Administration of Foreign Experts Affairs, P. R. China (funding number 20084600036).

[1]Website: http://www.iem.uni-due.de/~dreibh/omnetpp/.

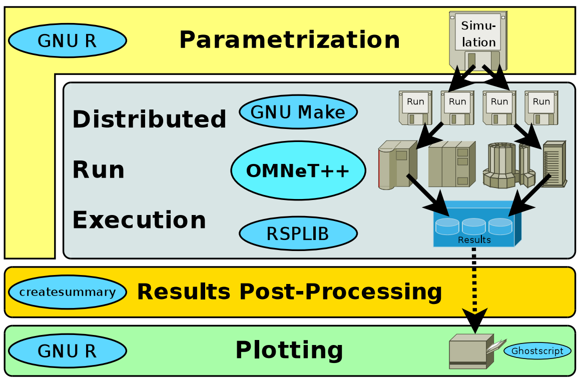

Figure 1: An Overview of Our Tool-Chain

2. SIMULATION PARAMETRIZATION

In order to perform simulation runs by using a simulation model, the first step is to parametrize the simulation by creating appropriate `.ini` files. Clearly, manually writing such files becomes extremely time-consuming for larger models – which can easily contain more than 100 parameters[2]. The core of SimProcTC is therefore a script which is responsible for performing the simulation parametrization.

2.1 Formal Definitions

For describing the simulation parametrization, it is useful to introduce some formal definitions first: let a simulation model have n *parameters* $p_1, \ldots, p_n$; $\hat{P}_1, \ldots, \hat{P}_n$ are the corresponding *parameter spaces* which contain all possible values. That is, $p_i \in \hat{P}_i$ for all $i \in \{1, \ldots, n\}$. Then, the *model parameter space* is $\hat{P} = \hat{P}_1 \times \hat{P}_2 \times \ldots \times \hat{P}_n$. Using this definition, a *simulation* $S \subset \hat{P}$ simply contains all parameter combinations $s \in S$ for which a *run* has to be performed. We assume for simplicity reasons that a run number corresponding with a certain random number generator seed is simply

[2]For example, the RSPSIM RSerPool simulation model [2, 3, 16] has almost 120 parameters.

another input parameter. The simulation binary itself constitutes a *simulation function* $f : S \to R$, which maps a run $s_j \in S$ to a result $f(s_j) \in R$ (scalars and vectors; we omit a formal definition here). We can further assume that for the same setting of s, always the same output is generated (or at least differences do not falsify the results[3]).

2.2 Realization

Clearly, the initial step of performing a simulation S is to parametrize it. An example from the RSPSIM model is presented in listing 1: `simulationConfigurations` is a list containing sub-lists. Each sub-list includes the parameter name (as first item) and all values to be used (as further items).

The following step is the generation of `.ini` files and their processing by the simulation model. This step has to meet the following two goals, in order to achieve an appropriate level of efficiency:

Extensibility It must be possible to add more values for some parameters, without having to re-process already performed runs.

Parallelizing It must be possible to process several runs in parallel – either on the same system (i.e. on multi-CPU and multi-core machines) or on different systems (using our RSerPool-based run distribution approach described in section 4).

In order to fulfill the above requirements, our simulation script first creates a separate *run directory* for each run $s \in S$. The initial version of SimProcTC has named the directory using a textual representation of s. However, this approach has easily reached the path length limit of the system – and directory names requiring several screen lines were *really* unhandy. Our solution has been to use an SHA1 hash [17] over s instead – resulting in appropriately small and usable directory names.

For each run s, a separate `.ini` file is generated in the corresponding run directory. It also specifies its own scalar and vector files, which will be placed in the same directory. A model-specific function writes the parameter section of the `.ini` file. That is: for each $s \in S$, the core script sets GNU R variables – the names are given by the parameter names – to the actual values defined by s. Then, a model-specific function called "`simCreatorWriteParameterSection()`" can write them as parameters in the `.ini` file. The example from the RSPSIM model shown in listing 1 uses a parameter `calcAppPoolElementServiceCapacityVariable` which defines a server capacity. In the scenario setup (for details, see [2]), there is actually an array of servers in an array of interconnected LAN networks. Therefore, the model-specific parametrization function actually writes the parameter line "`gammaScenario.lan[*].calcAppPoolElementArray[*].calcAppServer.serviceCapacity=1e7`" into the simulation run's `.ini` file (for `calcAppPoolElementServiceCapacityVariable`=1e7). Of course, more sophisticated parametrization – like writing multiple entries or even computing the actual values to be used in the `.ini` file – are possible as well.

Furthermore, the core simulation script will create a Makefile for GNU Make for the whole simulation S. Each $s \in S$ leads to an entry performing the following tasks:

[3]For example, the RSPSIM model also writes the actual run execution time as a scalar to allow for profiling.

1. Old output files (vector, scalar and log) are deleted.
2. The simulation model binary is executed by using the corresponding `.ini` file for s. It will write a log file as well as probably scalar and vector files.
3. The output files are compressed by BZip2 [18]. Since these files contain plain text, a significant disk space gain is herewith achieved.
4. Finally, a time-stamp file – denoted as *status file* – is written after successfully processing all former steps.

A re-run of the simulation script will update existing status files defaultly. That is, already executed runs will not be re-processed again – since their result would not change (due to our assumption for f above). If the simulation function f changes, the update step can be skipped and the runs will be executed again.

Since run directories are kept until being manually removed, this mechanism results in the desired caching behaviour: if the simulation is modified from S to $S' \subset P$, $S' \neq S$, it is only necessary to process the new runs $s \in S' \setminus S$. Note, that the runs $\overline{s} \in S \setminus S'$ are still kept on disk. They may be reused again after further modification of the simulation, e.g. after having made some small tests with a reduced number of parameters.

2.3 Handling Model Enhancements

During simulation-based research, it is a quite common task to enhance the functionalities of the existing simulation model. That is, new functionalities are added. Also, to actually use these new functionalities, new parameters are introduced – which have to be set to run the simulation. In the usual case, the new functionalities and behaviours of the model can be turned on by some parameter settings (e.g. the model for a component is enhanced by a countermeasure mechanism against Denial of Service attacks). Having already created a set of simulations by using the parametrization approach described in subsection 2.2, this leads to a problem: simply running these scripts results in the lack of parameter specifications for the new functionalities. That is, it would be necessary to modify all these scripts to set the new parameters appropriately to turn the new functionalities off (i.e. to get the old behaviour of the model).

Our approach to cope with this problem is quite simple: for the simulation model, a global *default configuration* D is specified – in the same way the simulation S is defined (see listing 2 for an example). Each time a new parameter is added to the model, a default for the new parameter is set here. Clearly, a good setting is to turn the new functionality off by default, i.e. using the default, the model behaves as before the change. Clearly, the default setting should contain exactly one value for each parameter.

To generate the actual simulation S^* from a simulation definition S and default settings D, the following merging rules are applied:

1. It is allowed that some parameter values in S are not specified. Formally, this could be reached by having each parameter space $\hat{P}_i$ containing an "undefined" entry $\varnothing$. However, we neglect a formal definition here, since the idea should be clear.
2. If a parameter value in S is missing, the corresponding value is taken from the default D. That is: if there is no setting, the default is used.

Listing 1 An Example Simulation Configuration from the RSPSIM Model

```
simulationConfigurations <- list(
   # ====== Variable Settings ================================================
   list("targetSystemUtilization", 0.70, 0.90),
   list("puToPERatio", 1, 2, 3, 4, 5, 7, 10, 15, 20),

   # ====== Pool Element Settings ============================================
   list("calcAppPoolElementServiceCapacityVariable", 1000000),
   list("calcAppPoolElementSelectionPolicy", "LeastUsed", "Random", "RoundRobin"),

   # ====== Pool User Settings ===============================================
   list("calcAppPoolUserServiceJobSizeVariable", 1e6, 1e7, 1e8),
)
```

Listing 2 The Defaults Specification from the RSPSIM Model

```
rspsimDefaultConfiguration <- list(
   # ====== Variable Settings ================================================
   list("targetSystemUtilization", 0.80),
   ...

   # ====== Pool Element Settings ============================================
   list("calcAppPoolElementServiceCapacityVariable", 1e6),
   list("calcAppPoolElementSelectionPolicy", "RoundRobin"),
   ...
   list("calcAppPoolElementServerCookieMaxCalculations", 1e7),
   list("calcAppPoolElementServiceMinCapacityPerJob", 1e5),
   ...

   # ====== Pool User Settings ===============================================
   list("calcAppPoolUserServiceJobSizeVariable", 1e7),
   ...

   ...
)
```

3. If there is a setting for a parameter p_i, it is used and the default value is simply ignored.
4. If there is a parameter setting in S but no default in D, an error will appear. Since $S \in \hat{P}$ and $D \in \hat{P}$, this cannot happen in theory – but for the simulation script, it has shown to be *very* useful to avoid this kind of problem – which is caused by typos in the parameter specifications.

An example is provided by the simulation configuration `simulationConfigurations` S in listing 1 and the default configuration `rspsimDefaultConfiguration` D in listing 2: according to rule #3, the parameter values for `targetSystemUtilization` are taken from S (i.e. 0.70 and 0.90). The simulation parameter `calcAppPoolElementServerCookieMaxCalculations` is not defined in S. It is therefore taken from the defaults D (i.e. using the value 1e7) according to rules #1 and #2. If there would be a parameter `thisIsATypo` in S – which is not defined in the defaults D – this would cause an error due to rule #4.

A positive side effect of the default configuration mechanism is that the actual simulation configuration S may remain small. In a usual simulation setup, there are only a few parameters which actually get modified – while most parameters stay at their default value. In particular, this keeps the simulation file for S also easily understandable (e.g. by users having only limited knowledge of the simulation model's full set of configurable parameters).

3. SIMULATION PROCESSING

In order to actually process a simulation which has been parametrized by our script introduced in section 2, it is sufficient to run GNU MAKE on the generated Makefile. This is realized by the simulation script itself. In particular, the simulation script also counts the number of CPUs/cores[4] the system provides and lets GNU MAKE execute the appropriate number of runs simultaneously[5]. That is, a system containing two dual-core CPUs should perform four runs simultaneously. However, this approach is still limited to a single PC only.

To allow for parallel simulation processing in our networking lab and on some spare PCs, we have first considered AKAROA, the proprietary X GRID [19] and the Grid compution system CONDOR. However, we preferred a more "lightweight", easy to use and in particular Open Source approach – something like RSerPool. This has led to the idea of actually using RSerPool for this task – in the form of our prototype implementation RSPLIB.

4. SIMULATION DISTRIBUTION

Since we apply RSerPool for our simulation distribution approach, it is first necessary to shortly introduce its architecture.

4.1 Reliable Server Pooling (RSerPool)

[4]Using the CPU information provided by the Linux kernel in `/proc/cpuinfo`.

[5]Using the GNU MAKE parameter `-j [jobs]`.

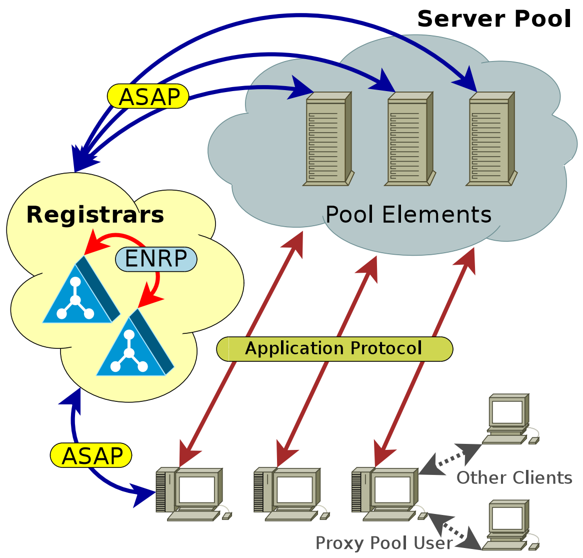

Figure 2: The RSerPool Architecture

The Reliable Server Pooling (RSerPool) architecture is the IETF's new standard for a lightweight server redundancy and session failover framework to support availability-critical applications as well as load balancing. It has become an international standard by publication as RFCs in September 2008. Figure 2 illustrates the RSerPool architecture [3, 20] which contains three types of components: servers of a pool are denoted as *pool elements* (PE), a client is called *pool user* (PU). The *handlespace* – which is the set of all pools – is managed by redundant *pool registrars* (PR). Within the handlespace, each pool is identified by a unique *pool handle*.

PRs of an *operation scope* (e.g. a LAN or company network) synchronize their view of the handlespace using the Endpoint haNdlespace Redundancy Protocol (ENRP [21]), transported via SCTP [22]. An operation scope is restricted to a single administrative domain (e.g. an organization or department), which reduces management complexity [23]. Being "lightweight" is the fundamental property of RSerPool [3]: it must also be usable on low-performance devices (e.g. routers or embedded systems). Therefore, the duty of RSerPool is the management of pools and sessions only, but it allows for a very efficient realization [23]. Nevertheless, PEs may be distributed globally, so that their service can survive localized disasters [10] (e.g. an earthquake or flooding). PRs can announce themselves to PEs, PUs and other PRs via UDP-based multicast messages. This functionality allows for the automatic configuration of all components.

PEs choose an arbitrary PR of the operation scope to register into a pool by using the Aggregate Server Access Protocol (ASAP [24]), again transported via SCTP. Within its pool, a PE is characterized by its PE ID, which is a randomly chosen 32-bit number. Upon registration at a PR, the chosen PR becomes the Home-PR (PR-H) of the newly registered PE. A PR-H is responsible for monitoring its PEs' availability by keep-alive messages (to be acknowledged by the PE within a given timeout) and propagates the information about its PEs to the other PRs of the operation scope via ENRP updates. PEs re-register regularly as well as for information updates.

In order to access the service of a pool given by its PH, a PU requests a PE selection from an arbitrary PR of the operation scope, using ASAP transported via SCTP. The PR selects the requested list of PE identities by applying a pool-specific selection rule, called *pool policy*. A basic set of adaptive and non-adaptive pool policies is defined in [25]. For this paper, only the adaptive Least Used policy is relevant: Least Used selects the least-loaded PE, according to up-to-date application-specific load information. Round robin selection is applied among multiple least-loaded PEs [16]. Details on other possible policies can be found in [3].

4.2 Distributing Simulation Runs

The "scripting service" (SS) is an example service of our RSerPool implementation RSPLIB [26]. It is included in the RSPLIB package itself. Using this service, a PU can establish a session with a pool and upload a TAR/GZIP-packed archive to a PE. The selected PE unpacks the archive into a temporary directory and executes a script included in the archive. This script can write an output archive, which is finally downloaded to the PU. The scripting pool can use the Least Used policy. Each PE can handle up to SSMaxThreads sessions simultaneously [12]; a PE's load value is set according to its actual number of sessions. Obviously, the scripting service can be applied for distributed simulation processing: instead of invoking the simulation model binary in the Makefile itself, it is provided by a PU to a PE and processed there. Finally, the received results are stored into the simulation directory. Using GNU MAKE to start multiple PU instances simultaneously, parallelizing is achieved.

The run distribution is performed by two scripts: the first one, called `ssdistribute`, is invoked by the Makefile. It calls the SS PU application with a TAR/GZIP package consisting of:

- The `.ini` file and all NED files of the project (plus optionally other files),
- The simulation binary, all shared libraries required by the simulation binary, the shared library loader[6] as well as
- The script `ssrun`.

Packaging NED and other files as well as the simulation binary, shared libraries and the shared library loader is only performed once, before the runs are processed by the Makefile. This achieves a significant efficiency improvement, since these files remain the same for all runs. Note, that the system working as SS PE does not even have to run the same Linux distribution as the system having compiled the simulation model. Since the simulation binary as well as all of its shared libraries are provided by the SS PU, the only requirement is a compatible CPU architecture. That is, if the simulation binary is compiled on x86, the SS PEs can use an x86 or x86_64 CPU (64-bit systems can run 32-bit binaries). For a 64 bit PU, and therefore for a 64-bit binary, all PEs would have to be 64-bit machines.

The `ssrun` script, which is part of the package provided by the SS PU to a PE, is executed on the remote machine and calls the simulation binary with the corresponding input file. Afterwards, it collects scalar, vector and log files, performs BZIP2-compression and puts them together into an archive. This archive is downloaded by the PU and stored in the corresponding simulation directory. If a PE rejects a session (since already serving SSMaxThreads sessions), or if it goes

[6]On a current Linux system, this is `/lib/ld-linux.so.2`.

out of service (e.g. the PC is turned off), the session is simply restarted from scratch ("abort and restart" principle [9]) after a short delay (e.g. 5s). This delay avoids overloading the network with reject-and-retry floods [12] when there are too few PEs available.

4.3 Our Pool Setup

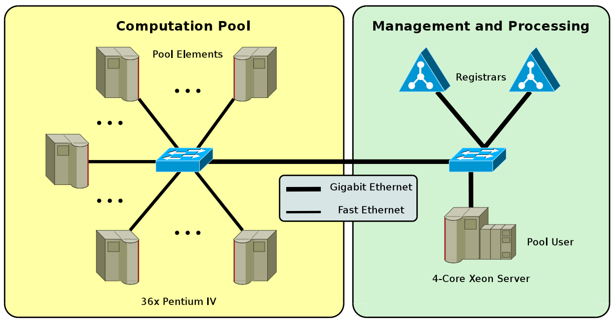

Figure 3: Our Simulation Computation Pool

Figure 3 presents our lab setup at Hainan University: 36 PCs (Linux systems with single-core Pentium 4 CPU) run a SS PE service. These machines only need a basic installation of RSPLIB – it is *not* necessary to install OMNeT++, GNU R, the simulation model itself, etc. on these systems. The only system requiring these installations is the 4-core Xeon server running Kubuntu Linux 8.04. It contains the SimProcTC tool-chain as well as the simulation model. This machine is used to distribute the simulation runs. In order to fully utilize its CPU power, it furthermore also runs an SS PE (with reduced process priority). The pool therefore has the ability to process 40 runs in parallel.

5. RESULTS VISUALIZATION

The goal of a simulation is to obtain results, which have to be visualized in an appropriate form for easy analysis and interpretation. The focus of SimProcTC is currently to visualize the scalars. The scalars of a simulation are distributed over the scalar files located in the simulation directories. Therefore, it is first necessary to bring the scalars into an appropriate form to visualize them.

5.1 Scalars Summarization

For the task of collecting the scalar values from the various scalar files, a C++-written program called `createsummary` has been developed. While this task would also be possible using GNU R itself, the requirements on memory and CPU power have quickly led to this external program.

`createsummary` is called as last step of the Makefile and iterates over all scalar files of the simulation S. Each file is read – with on-the-fly BZip2-decompression – and each scalar value as well as the configuration $s \in S$ having led to this value – are stored in memory. Depending on the number of scalars, this in-memory storage can result in huge memory requirements (e.g. multiple GiB). But in the usual case, not all scalars of a simulation are required for analysis. Consider for example the statistics results of a TCP/IP application. If only the statistics (i.e. scalars) of the application itself are of interest, it is not necessary to write e.g. IP, TCP or Ethernet statistics as well. Therefore, `createsummary` can use an exclusion list denoted as "summary skip list". Scalars matching the exclusion pattern are simply skipped. Note, that the scalars are still stored in the scalar files themselves. That is, should these values be required later, it is possible to simply re-run `createsummary` – with updated skip list – to also process them.

Listing 3 An Example GNU R Data File

```
          Size   Interval    ID   System     Speed
0001         1        100  Test    Alpha     39.21
0002        20        150  Test     Beta     48.20
0003        20        152  Test     Beta     96.03
0004        20        155  Test     Beta     12.62
0005        50        140  Test    Alpha    139.23
0006        75        180  Test     Beta     45.34
0007        80        120  Test    Alpha     73.28
0008        90        145  Test    Alpha     59.29
 ...       ...        ...   ...      ...       ...
```

Having all relevant scalars stored in memory, it is easily possible to write each scalar into a separate data file. Such a data file – which can be processed with GNU R or other programs – is simply a table in text form, containing the column names on the first line. Each following line contains the data, with line number and an entry for each column (all separated by spaces). That is, each line consists of the settings of all parameters and the resulting scalar value. An example data file is shown in listing 3; it contains the parameters *Size*, *Interval*, *ID* and *System* as well as the scalar *Speed*. Since most model parameters do not change in a simulation S – i.e. their value is constant and their table column therefore contains the same value on each line (e.g. ID=Test in the example) – there would be a *huge* waste of storage and memory space as well as CPU power. Therefore, such columns are simply not written unless explicitly requested (e.g. if needed for post-processing later). Furthermore, the resulting data files are also BZip2-compressed on the fly.

5.2 Plotting

Using the scalar output files written by `createsummary` (or alternatively other output data) the results can be presented visually. Since we have already used GNU R [4] for the parametrization, it is quite straight-forward to also use it for plotting as it also contains a rich set of graphics functions. In particular, it allows for a very fine-granular control of the output plots to adapt the presentation to special requirements (e.g. labels, grids, colours, line styles, etc.). However, it would also be possible to apply other tools – e.g. GNU Octave and GNU Plot or even Microsoft Excel – for visualizing the results.

For results analysis, it is crucial that the impact (i.e. the scalar value) for variations of multiple parameters can be presented in an easy-to-understand form. In our plotting approach, we apply the idea of axes, onto which parameters and scalars can be mapped:

- X-axis and Y-axis are obvious: the main parameter is displayed on the X-axis, the result (i.e. a scalar) on the Y-axis.
- There can be multiple lines per plot: the Z-axis identifies a line. For readability, we map a different colour or shade to each line.
- The Z-axis can be further subdivided: the V-axis uses a different line style. Also, the V-axis can be further subdivided: the W-axis uses a different point style.

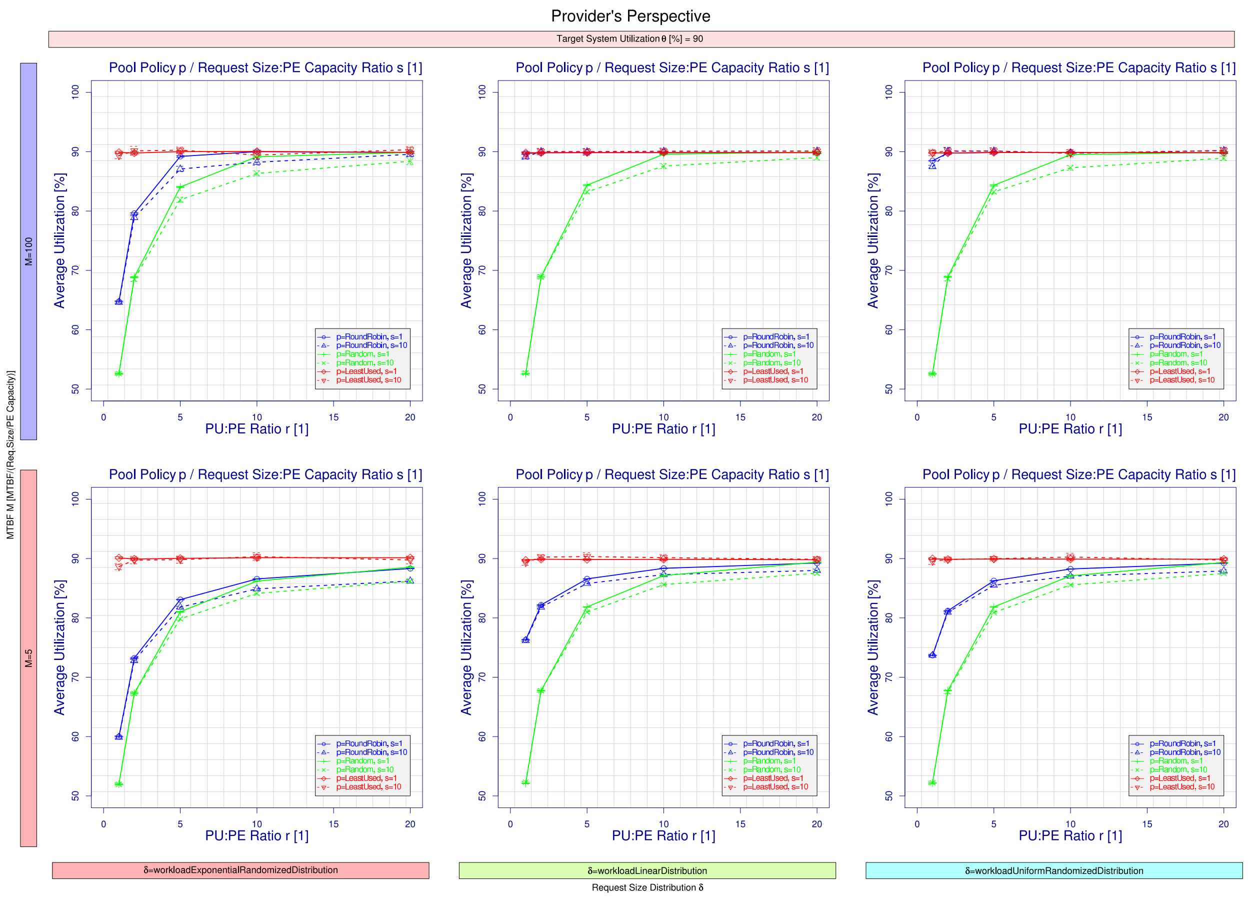

Figure 4: A Complex Example Plot using X/Y/Z/V/A/B/P Axes

- On one page, there can be multiple plots: the A-axis divides the page in horizontal direction, the B-axis in vertical direction (i.e. a separate plot for each A/B-axis value).
- Finally, the P-axis creates a separate page for each P-axis value.

To make our plot idea clear, figure 4 shows an example[7] plot from the RSPSIM model: The X-axis presents the parameter PU:PE ratio r and the Y-axis shows the scalar value of the system utilization (in %). The Z-axis represents the pool policy: i.e. each policy gets its own colour (colour plot) or shade (gray scale plot). The V-axis presents the Request Size:PE Capacity ratio s: each value – s=1 and s=100 – is marked by a different line style. There is no W-axis in this plot. The A-axis displays the Request Size Distribution δ and the B-axis shows plots for each PE MTBF value M. Using the P-axis, a separate page is created for each Target System Utilization setting ϑ. The plot shown here is for ϑ=90% (this information can be found in the box below the plot title).

The further features of our GNU R-based plot script are as follows:

1. If there are multiple values per plot (e.g. from runs with different seeds), the average value is taken for plotting. Furthermore, the confidence intervals (usually 95%) are computed and displayed.
2. It is possible to define a mapping from an axis label to a variable (e.g. $r \rightarrow \text{PU} : \text{PE}$ ratio – to simplify writing a description).
3. The output can optionally be in black and white, grey scale or colour.
4. All plots are written as PDF files (i.e. a vector format), for efficient inclusion into pdfLaTeX documents. If necessary, the PDF files could be converted into raster formats like PNG or GIF (e.g. for inclusion into MICROSOFT OFFICE).

[7]A detailed parameter description can be found in [3].

5.3 Plotting Templates

To speed up the definition of plots to be created, the SIMPROCTC plotter script allows for the definition of templates describing the mapping of table columns to axes. These templates are used to write the actual plot definitions. Listing 4 provides an example for two plots: system utilization (this plot is shown in figure 4) and request handling speed. The plot configuration in `plotConfigurations` simply consists of a list of plot definitions: the first line of each definition provides the simulation directory (i.e. where to find the results data) and the PDF output file name (created using the directory name). In the second line, the plot title, optional ranges for X-axis and Y-axis (NA denotes automatic choice) and the legend position (X and Y position from 0.0 to 1.0) are provided. The following definitions set the templates to be applied for the axes (X, Y, Z, V, W, A, B and P) and an

Listing 4 An Example Plot Definition

```
simulationDirectory <- "wp1-hom-puToPERatioI"
...

# ====== Templates ===========================================================
plotVariables <- list(
   # ------- System Utilization Template ------------------------------------------------
   list("controller.SystemAverageUtilization",
        "Average␣Utilization [%]",
        "100.0␣*␣data1$controller.SystemAverageUtilization",
        "blue4",
        list("controllerSystemAverageUtilization")),
   ...
)
...

# ====== Plots ================================================================
plotConfigurations <- list(
   filter <- "data1$targetSystemUtilization␣>=␣0.80"

   # ------- System Utilization Plot -------------------------------------------------------
   list(simulationDirectory, paste(sep="", simulationDirectory, "-Utilization-P%d.pdf"),
        "Provider's␣Perspective", NA, NA, list(1,0),
        "puToPERatio", "controller.SystemAverageUtilization",
        "calcAppPoolElementSelectionPolicy", "jsToSC", "",
        "calcAppPoolUserServiceJobSizeDistribution",
        "calcAppPoolElementComponentUptimeVariable-MTBF", "targetSystemUtilization",
        filter),

   # ------- Handling Speed Plot -----------------------------------------------------------
   list(simulationDirectory, paste(sep="", simulationDirectory, "-HandlingSpeed-P%d.pdf"),
        "User's␣Perspective", NA, NA, list(0,1),
        "puToPERatio", "controller.SystemAverageHandlingSpeed",
        "calcAppPoolElementSelectionPolicy", "jsToSC", "",
        "calcAppPoolUserServiceJobSizeDistribution",
        "calcAppPoolElementComponentUptimeVariable-MTBF", "targetSystemUtilization",
        filter)
)
```

optional filter expression (to be explained below).

The templates are defined in `plotVariables`. A template does not only correspond to a certain data table column, it can furthermore also apply data modification. For example, we plot the system utilization in the first plot – which is provided as values from 0.0 to 1.0 in the data file. For readability reasons, we have configured the template (`controllerSystemAverageUtilization`) to multiply it by 100 to obtain a value in %. The template furthermore defines the axis label. Using template-based definitions, plots can be defined very easily.

The Y-axis template also specifies the input file (here: "controllerSystemUtilization") from which the data is actually read. Its data table in GNU R is referenced by the variable data1. For purposes like creating plots for a paper, it is often not desired to plot all data. That is, a useful parameter subset has to be extracted. This is achieved by using a filter expression (given as string): in the example, "data1$targetSystemUtilization >= 0.80" selects only the table rows which have the "targetSystemUtilization" entry set to a value greater or equal 0.80. Since the filter expression is simply a GNU R expression, it is of course possible to specify more complex filters here – in particular, the OR "|" or AND "&" operators can be used to combine conditions and the NOT "!" operator can be used for negation.

GNU R is already capable of writing its plots into PDF files. However, no fonts are embedded into the output file. That is, when included into pdfLaTeX, the resulting file will contain non-embedded fonts; the PDF file simply references fonts installed on the local system. Since wrong font mappings lead to problems for printing and displaying, such files are e.g. disallowed for the camera-ready versions of conference papers. This problem can be solved easily by a PDF post-processing step: the plot file simply has to be processed by GHOSTSCRIPT using *pdfwrite* as output device. The resulting new PDF file will have all required fonts embedded. Furthermore, the resulting file will even be compressed – which can significantly reduce its size.

6. CONCLUSIONS AND OUTLOOK

In this paper, we have introduced design and realization of SIMPROCTC – our Open Source tool-chain to perform the parametrization, distributed run execution and results visualization in OMNET++-based simulations. By using SIMPROCTC, these frequently recurring tasks of simulation work can be performed in an efficient and easy to use manner. Our tool-chain is already successfully deployed for a number of simulation projects.

Currently, we are evaluating the usage of XEN-based virtualization for distributed run processing. This approach not only provides enhanced security but also allows for checkpointing simulation runs. That is, when a processing node goes out of service, a run can be resumed on another system. This can significantly improve the performance in unreliable simulation computation pools (e.g. PCs in a student lab).

7. REFERENCES

[1] A. Varga. *OMNeT++ Discrete Event Simulation System User Manual - Version 3.2.* Technical University of Budapest/Hungary, March 2005.

[2] T. Dreibholz and E. P. Rathgeb. A Powerful Tool-Chain for Setup, Distributed Processing, Analysis and Debugging of OMNeT++ Simulations. In *Proceedings of the 1st ACM/ICST OMNeT++ Workshop*, Marseille/France, March 2008. ISBN 978-963-9799-20-2.

[3] T. Dreibholz. *Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture.* PhD thesis, University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems, March 2007.

[4] R Development Core Team. *R: A language and environment for statistical computing.* R Foundation for Statistical Computing, Vienna/Austria, 2005. ISBN 3-900051-07-0.

[5] X. Zhou, T. Dreibholz, F. Fa, W. Du, and E. P. Rathgeb. Evaluation and Optimization of the Registrar Redundancy Handling in Reliable Server Pooling Systems. In *Proceedings of the IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA)*, Bradford/United Kingdom, May 2009.

[6] X. Zhou, T. Dreibholz, W. Du, and E. P. Rathgeb. Evaluation of Attack Countermeasures to Improve the DoS Robustness of RSerPool Systems by Simulations and Measurements. In *Proceedings of the 16. ITG/GI Fachtagung Kommunikation in Verteilten Systemen (KiVS)*, Kassel/Germany, March 2009.

[7] P. Schöttle, T. Dreibholz, and E. P. Rathgeb. On the Application of Anomaly Detection in Reliable Server Pooling Systems for Improved Robustness against Denial of Service Attacks. In *Proceedings of the 33rd IEEE Conference on Local Computer Networks (LCN)*, pages 207–214, Montreal/Canada, October 2008. ISBN 978-1-4244-2413-9.

[8] T. Dreibholz, E. P. Rathgeb, and X. Zhou. On Robustness and Countermeasures of Reliable Server Pooling Systems against Denial of Service Attacks. In *Proceedings of the IFIP Networking*, pages 586–598, Singapore, May 2008. ISBN 978-3-540-79548-3.

[9] T. Dreibholz and E. P. Rathgeb. Reliable Server Pooling – A Novel IETF Architecture for Availability-Sensitive Services. In *Proceedings of the 2nd IEEE International Conference on Digital Society (ICDS)*, pages 150–156, Sainte Luce/Martinique, February 2008. ISBN 978-0-7695-3087-1.

[10] T. Dreibholz and E. P. Rathgeb. On Improving the Performance of Reliable Server Pooling Systems for Distance-Sensitive Distributed Applications. In *Proceedings of the 15. ITG/GI Fachtagung Kommunikation in Verteilten Systemen (KiVS)*, pages 39–50, Bern/Switzerland, February 2007. ISBN 978-3-540-69962-0.

[11] X. Zhou, T. Dreibholz, and E. P. Rathgeb. Improving the Load Balancing Performance of Reliable Server Pooling in Heterogeneous Capacity Environments. In *Proceedings of the 3rd Asian Internet Engineering Conference (AINTEC)*, volume 4866 of *Lecture Notes in Computer Science*, pages 125–140. Springer, November 2007. ISBN 978-3-540-76808-1.

[12] X. Zhou, T. Dreibholz, and E. P. Rathgeb. A New Approach of Performance Improvement for Server Selection in Reliable Server Pooling Systems. In *Proceedings of the 15th IEEE International Conference on Advanced Computing and Communication (ADCOM)*, pages 117–121, Guwahati/India, December 2007. ISBN 0-7695-3059-1.

[13] W. Zhu, T. Dreibholz, E. P. Rathgeb, and X. Zhou. A Scalable QoS Device for Broadband Access to Multimedia Services. In *Proceedings of the IEEE International Conference on Future Generation Communication and Networking (FGCN)*, Sanya, Hainan/People's Republic of China, December 2008.

[14] W. Zhu, T. Dreibholz, and E. P. Rathgeb. Analysis and Evaluation of a Scalable QoS Device for Broadband Access to Multimedia Services. In *Proceedings of the 33rd IEEE Conference on Local Computer Networks (LCN)*, pages 504–505, Montreal/Canada, October 2008. ISBN 978-1-4244-2413-9.

[15] I. Rüngeler, M. Tüxen, and E. P. Rathgeb. Integration of SCTP in the OMNeT++ Simulation Environment. In *Proceedings of the 1st OMNeT++ Workshop*, Marseille/France, March 2008. ISBN 978-963-9799-20-2.

[16] T. Dreibholz and E. P. Rathgeb. On the Performance of Reliable Server Pooling Systems. In *Proceedings of the IEEE Conference on Local Computer Networks (LCN) 30th Anniversary*, pages 200–208, Sydney/Australia, November 2005. ISBN 0-7695-2421-4.

[17] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). Informational RFC 3174, IETF, September 2001.

[18] J. Seward. *bzip2 - A program and library for data compression.* Snowbird, Utah/U.S.A., February 2005.

[19] R. Seggelmann, I. Rüngeler, M. Tüxen, and E. P. Rathgeb. Parallelizing OMNeT++ simulations using Xgrid. In *Proceedings of the 2nd ACM/ICST OMNeT++ Workshop*, Rome/Italy, March 2009.

[20] P. Lei, L. Ong, M. Tüxen, and T. Dreibholz. An Overview of Reliable Server Pooling Protocols. Informational RFC 5351, IETF, September 2008.

[21] Q. Xie, R. Stewart, M. Stillman, M. Tüxen, and A. Silverton. Endpoint Handlespace Redundancy Protocol (ENRP). RFC 5353, IETF, September 2008.

[22] R. Stewart. Stream Control Transmission Protocol. Standards Track RFC 4960, IETF, September 2007.

[23] T. Dreibholz and E. P. Rathgeb. An Evaluation of the Pool Maintenance Overhead in Reliable Server Pooling Systems. *SERSC International Journal on Hybrid Information Technology (IJHIT)*, 1(2):17–32, April 2008.

[24] R. Stewart, Q. Xie, M. Stillman, and M. Tüxen. Aggregate Server Access Protcol (ASAP). RFC 5352, IETF, September 2008.

[25] T. Dreibholz and M. Tüxen. Reliable Server Pooling Policies. RFC 5356, IETF, September 2008.

[26] T. Dreibholz. Thomas Dreibholz's RSerPool Page, 2008.