ABSTRACT

I am working with software in academia for more than an decade and I had the “Moment” quite often. Palladio appeared just like an ordinary tool to solve my problem. Then, I changed a single parameter of my simulation—some hours later we hunted for a bug in the depths of Palladio. Based on the open source development model of Palladio and a very elegant structure of the source code, we were able to find the root cause of the problem very fast. To start fixing the problem, we “just” had to know when—in simulation time—a measurement of the Simulizar simulator is valid. This paper summarizes our technical and philosophical discussions that ware needed to make Palladio deliver correct results and not to get lost in the depths of time and duration.

1. Introduction

Some of the readers certainly know the “Moment”. A small putative problem at the surface of a software pushes you deeper and deeper into its holy guts and to the question, whether the problem can be fixed at all. For the readers not knowing this “Moment”, we cite a former colleague who described how the “Moment” feels: Your arm is nearly completely in a warm, brown, smelling mass and by searching for the right switch to solve your problem, you have just learned that the switch is somewhere deeper in this mass.

To understand a problem in Palladio, you have to solve two problems at once. First, you need to read and understand the source code to find the right place and the right approach to fix the bug. Second, you need to understand the rationale of the realization of the model-based simulation—the part you planned to reuse as black-box. Thus, you need to understand a lot of stuff you never wanted even to know about.

In this paper, we want to sum up some of our findings of this journey. We start with a not-so-well-known feature of Palladio to model and analyze architectures with reconfigurations (Sec. 2), discuss shortly how to control such a reconfiguration (Sec. 3), discuss the bug (Sec. 4) including the fundamental requirements to specify the validity of values in time (Sec. 5), and give a conclusion (Sec. 6).

2. Architectural Templates and Reconfigurations with Palladio

A software-intensive system, e.g. a Cloud system—this is what we analyzed in our use case—often needs to handle variable load and should use as less resources as possible to reduce costs. This demands an elastic system—a system that automatically scales to handle the current load with an adequate amount of resources. This behavior can be realized, e.g., by adapting the number of used servers, so called horizontal scaling. Horizontal scaling can be realized by automatic reconfigurations.

We used an automatic reconfiguration based on the Architectural Templates (AT) method [2]. This method allows to model reconfigurations very fast and with much less hassle then by a manual realization of a reconfiguration. In short, when the system we simulated with Palladio needed more resources, additional servers were automatically added to the resource environment and removed from the resource environment when not needed any more.

Instead of modeling a load balancer, several servers and a reconfiguration strategy, the AT method allows to model only one server and annotate it with reconfiguration parameters. The load balancer, the replication of the server, and the reconfigurations are transparently generated by the AT in the background. This makes reconfigurations for elasticity less error-prone and more efficient to model.

To configure reconfigurations, parameters have to be set: a threshold for lowering the server count (scale-in), a threshold for increasing the server count (scale-out), a reconfiguration interval (time between two reconfiguration checks), and a metric (e.g. response time) with a statistical aggregation function (e.g. statistical mean). After every reconfiguration interval, the function is used to determine a representative value for the metric over the last interval. This value is compared to the threshold to determine whether a scale-out or -in needs to be executed.

3. The Metric Parameter of the Horizontal Scaling Strategy

The default parameter of a reconfiguration is the CPU utilization, but for our use case, we wanted to set the waiting requests as parameter. In the following, we briefly describe these two options and give some background based on our use case.

CPU utilization. The standard configuration of the used horizontal scaling AT is to use a server’s CPU utilization. A server that is utilized more than 80 % (scale-out threshold) is categorized as highly used and thus additional servers are provisioned. That a load of more then 80 % can get critical is proven by numerous reports. It is a rules of thumb of server administration and it is also known that this rule can be broken in some cases. See also http:
Often the scale-in threshold is set to 60 %. To think about the scale-in threshold, we conduct a small thought experiment. We consider to have two servers. If both servers are utilized slightly below 60 %, one server is removed and the load on the remaining server is doubled because it has to handle the complete load. This results in one server with about 120 % load. Thus a scale-out follows, followed by a scale-in, and so on. To avoid these infinite reconfigurations we have to use a scale-in threshold of less than 40 %, half of the scale-out threshold. Considering a system with 100 servers, a single server more or less makes just 1 % change to the load of the remaining ones, not a doubling as in the case of two servers. Thus, we finally set the scale-in threshold to 60 % and expect to have more than two servers most of the time.

In situations where the system runs in a state that not causes a reconfiguration, the load will be between 60 % and 80 %. In other words 20 % to 40 % of the server’s capacity are not utilized. Such a fact is not so easy to sell to the management—especially when we have to add the information, that used servers are very expensive. Some more information about the underlying use case can be found on the SFC project page1 and in [1].

The use case we consider needs highly secured servers that are quite slow and expensive. In opposite to, e.g. a web server, these servers process exactly one request at a time, so a single server is either 0 % or 100 % utilized. So altogether the utilization metric (the standard realization) does not look like a good choice for us.

Waiting Requests. Important for our use case is to avoid that the number of waiting requests increases which results in an endless backlog. Additionally, it has to be avoided that servers are unutilized.

Therefore, we decided to use the number of waiting requests to determine when to scale. We want to scale-in when no request is waiting to be processed and scale-out when some number of requests, e.g., five, are waiting.

Currently, this cannot be realized easily in Palladio. The number of waiting requests can be used easily as parameter. However, its measurements include the requests that are currently processed. This is the number of requests that have to be completed, while we want the number of requests that are currently waiting for processing.

In our use case, the number of waiting requests has to be decreased by the number of servers to get the number of waiting and not yet processing requests. A nice solution to realize this would be to calculate a virtual parameter based on multiple parameters. This can be realized by implementing a calculator within SimuLizar. We decided against this solution as it is too much work for a side task. For a first proof-of-concept, we took the number of waiting requests and increased the number by the number of servers we expected to be used. For a static load, we can reevaluate and adapt our model parameter. But the result of the test run was not a set of measurements that can be visualized by fancy diagrams to bring us insights, but the result was an ordinary Java NullPointerException. So we started to hunt for a bug.

1http://www.vdivide-it.de/KIS/sichere-ikt/sicheres-cloud-computing/sfc

4. Hunting the Bug, Fighting in the Depths of SimuLizar

When using Palladio, you get used to dealing with Java exceptions. The model you want to simulate is often a complex one and you have to deal with a lot of different views. In case you forgot to set all needed parameters, you get an exception. This sounds very odd but can be handled by experienced users. In most cases, the message and stack trace of the exception tells you, what is wrong with your model and you can repair it.

To make a typical example for such a behavior, we consider a data storage. To know how big your resource demand for storing the data will be, you need the data size. Thus, the data size has to be known at simulation time. To realize this, the data size has to be set within the usage scenario and has to be a parameter of many components (or better the behavior descriptions of the components (SEFF)) between the usage scenario and the storage component. According to my experience, it is very likely to forget to set the size of the output data at some point (mostly when the output size is the same as the input size and you do not think about it at all).

A best practice as a new Palladio user is to see a Palladio programmer once a day (e.g., during a coffee break). He can fix your models, identify the exceptions related to a bug, write bug reports, and sometimes fix bugs. In some minutes, you can get work done that normally takes you half a week of learning and understanding, so your learning curve is much better.

To make a long story short, I took an hour to check my model and afterwards called my colleague (coauthor and developer of Palladio) whether he can explain the exception to me. I told him that I broke his ATs, so he was very motivated to prove me wrong. We started a peer debugging session during a coffee break. Based on the stack trace, we could identify the position of the crash and it was clear that the problem was caused by SimuLizar and not by the ATs. The bad news was that the creator of the bug did no longer work on Palladio so a bug report would not solve the problem in a decent time frame.

Most of the classes and functions delivered exactly what can be expected from their name. A short documentation of prerequisites and assumptions would help from time to time. Really helpful would be to know, whether you deal with good maintained code or legacy artifacts.

Our problem was based on such a legacy artifact. One of our experts could identify this and an other expert confirmed it. He also warned us not to change the behavior—based on the fact that the code stems from the entry version of SimuLizar and its dependencies. The calculated variables are needed to calculate something that is most likely used to calculate statistics at simulation end—based on the proven reliability the code seems to work—even in case we do not know how.

Exactly this code writes (or does not write) the current state (measurements) of the simulation to a data structure that has to be used to test for reconfigurations. As a first step, we introduced a mechanism to specify which metrics are needed and have to be written to the data structure. Therefore, we introduced code to configure the metrics in the monitor/repository. The flag “Triggers Self Adaptations” can now be set for each measurement parameter to specify whether it can trigger reconfigurations or not, and
“Statistical Characterization” can define the statistical aggregation function to us (e.g., arithmetic mean).

We tried to avoid changes to the legacy code. Instead, we searched for the class triggered at the end of a reconfiguration interval. This is the entry point where we have to write the actual measurements to the needed data structure. To sum up, we had to write some code for collecting the data by implementing the according interfaces, set a call to the statistical function and send the result to the data structure. Based on the former modifications to specify the needed reconfiguration parameters, this was mostly easy.

A problem was to select the correct statistical function. This kind of function was implemented twice. This could be avoided by refactoring. That different functions were available did not mean that they were selected correctly. So we fixed and refactored the selection mechanism for the already implemented cases.

Missing was a set of functions that can deal with non-continuous measurements (e.g., waiting requests). Thus we implemented some improvements, but could not get our simulation to run yet. This was the point when we reached the depths of SimuLizar.

5. Values and their Meaning in Time
At the end of every reconfiguration interval, a statistical aggregation function is used to calculate a value for that interval. At the starting point (before fixing the bugs), only the mean of the utilization was calculated. To calculate the mean, all utilization measurements of the interval are used by SimuLizar.

When using the number of waiting requests, it gets a little bit more complicated. This metric is measured whenever a request is added or a request is completed. Thus, we can have reconfiguration intervals without a measurement.

To generalize the calculation, we need to distinguish between discrete and continuous measurements:

Discreetly taken measurements like the CPU utilization are measured by SimuLizar for each step of the simulation. These steps are equidistant. Information about the measurements between two measurements is not present. Measurements like the number of waiting requests get only updated when they change. Thus they are not equidistant. Additionally, we have information about measurement values between two measurements. The value is valid after a measurement until the next measurement changes the value.

SimuLizar handles discrete measurements since its first version. Continuous measurements are a new concept.

To get our simulation running, we needed to realize statistical aggregation functions for continuous measurements. Therefore, we needed to decide whether a measurement is discrete or continuous. We categorized all available measurement parameters for Palladio into discrete or continuous. In the discrete case, the calculations are done using the already implemented statistical aggregation functions that we refactored to also support the continuous ones.

At the moment, we assume that a continuous measurement is consistent until the next measurement occurs. In principle, it is possible to have other kinds of continuous metrics, e.g. the measurement can be valid for the time interval before it is taken or a linear changing of the value could be possible. Based on the fact that we did not get requirements for such kinds of continuous measurements and we do not want to overengineer, we just implemented the first case.

Based on the categorization of measurements into discrete or continuous, a statistical aggregation function is now automatically selected. For the continuous case, the statistical functions are mathematically defined as integrals. Based on the stepwise change of measurements during a reconfiguration interval, the integrals can be solved in linear equations. In contrast to discrete measurements, we also had to care about the start of an interval. In case that no measurement is taken at start point, the last measurement we have remains valid. To compensate this and to avoid to deal with special cases in the calculation, we set a virtual measurement at the begin of an interval. We also set a virtual measurement at the end of the reconfiguration interval. As result, the reconfiguration always starts and ends with a measurement and no overlaps between measurements and reconfiguration intervals have to be considered.

For some of the (discrete) statistical aggregation functions, we could not find an continuous equivalent. In such cases, we throw a meaningful and easy to understand exception. This follows the common behavior of Palladio analyzers.

6. Conclusion
Mission completed—we have learned a lot about the internals of SimuLizar and finally got our simulation to run. Using Palladio needs more than just experience: you need contact to the community. Based on the fact that it is really easy to communicate with Palladio developers (even if you do not have the option to join a daily coffee break), this should not be a reasonable hurdle. You also get a good feedback and learn how to model and analyze in general. It takes a steep learning curve and it can get really complicated when you reach the rarely used and special Palladio features.

Even when we cannot provide a roadmap for the following request, it would be good to have better documentation of the fundamentals of Palladio and the resulting limitations. In the case we presented in this paper, we extended SimuLizar to support other kinds of data representations. This is an extension to Palladio’s depths but does not need to deal with restrictions of the simulation concept. A similar but more complex extension would be a pipes and filters framework to derive parameters for visualization of measurements and for reconfigurations.

Finally, after introducing the Palladio extension, we focus now on our initial goal—analyzing our models.

References