A Self-Optimizing Run-Time Architecture for Configurable Dependability of Services*

Matthias Tichy and Holger Giese

Software Engineering Group, Department of Computer Science
University of Paderborn, Germany
[mtt|hg]@uni-paderborn.de

Abstract. Many human activities today depend critically on systems where substantial functionality has been realized using complex software. Therefore, appropriate means to achieve a sufficient degree for dependability are required, which use the available information about the software components and the system architecture. For the special case of service-based architectures, we identify in this paper a set of architectural principles which can be used to improve dependability. We then describe how the identified architectural principles have been used in a realized service-based architecture which extends Jini. The dependable operation of the infrastructure services of the architecture further enables to systematically control and configure some dependability attributes of application services. We present a qualitative and quantitative evaluation of the dependability for a configuration of multiple identical services which are executed with the architecture and show how the different parameters effect the dependability. Additionally, a scheme for the dynamic control of the required dependability of the application services in the case of changing failure characteristics of the environment is outlined. Finally, we present a first evaluation of the developed architecture and its dynamic control of dependability.

1 Introduction

The dependability of today's complex systems often relies on the employed computers and their software components. Availability, reliability, safety and security (cf. [1]) are the attributes of dependability that are used to describe the required system characteristics. In this paper we only consider the dependability attributes availability and reliability. They can be systematically studied at the level of components and their composition for a given static system architecture by deriving related dependability models which can be quantitatively analyzed (cf. [2, 3]). However, due to the increasingly dynamic character of today's computing environments such as service-based architectures we often do not have a static system architecture.

When further considering dynamic systems where no static *a priori* known system configuration exists, the analysis and prediction of the reliability or availability using a model of the overall configuration is difficult. We therefore propose to build dynamic

* This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

and dependable complex systems not by relying on design-based quantitative analysis of its static overall architecture. Instead, the observed obstacles should be addressed by a component-wise analysis which at run-time monitors and controls the dynamic reconfiguration of the architecture to prevent system reliability and availability to decrease below the required level. Such a software tries to compensate failures (originated from defects of its hardware and software components) by means of repair and it tries to compensate changes of the environment by adaption of its maintenance activities to keep the system's availability and reliability above the required level.

Our architecture consists of an infrastructure, which offers the required generic and configurable maintenance activities, and the application specific services. We further study how the dynamic management of redundant component instances for infrastructure components as well as application services with identical implementation can contribute to improvements for the two considered dependability attributes.

For the infrastructure components we present an in-depth qualitative analysis of relevant hardware failures such as node crashes or network partitioning to show that no single-point-of-failure exists. The application services are in contrast treated as black-boxes with dependability characteristics that may change over time. Following the categorization of [4] we only distinguish whether a service represents an *entity* or only contains an activity (*session*). For session services we further consider stateless and stateful ones where for the latter the history of relevant actions executed throughout this session builds the state.

The failures considered for application services are crash failures, which may either result from defects of the employed hardware or result from the executed software. While the rate of the former may change due to aging or replaced hardware, the later may change over time due to changes in their usage characteristics or software updates. We further make the strong simplification that both kinds of failures simply result in an application service crash failure. As the services will then fail to fulfill their regular behavior expected by the infrastructure the crash can be detected externally by monitoring the services.

A number of established architectural principles are presented in Section 2, which permit to enhance the dependability of service-based architectures. Then, we present in Section 3 our enhancement of the Jini architecture by a number of infrastructure services that systematically employ the identified principles to provide a dependable execution of application services. For a special class of services the possible design alternatives are studied by means of the detailed design of two infrastructure services concerning a dependable operation.

We then discuss the benefits achieved for application specific services concerning availability in Section 4 in a qualitative evaluation. We demonstrate the systematic application of the identified architectural principles within the enhanced architecture. Afterwards, we present a formal model of the architecture. This model provides analytical results of the availability for application services based on the parameters of the architecture and the environment. After that we sketch how to adapt parameters of the control monitors to sustain the availability of application services w.r.t. environmental changes.

Section 5 contains a qualitative and quantitative evaluation of the reliability provided by our architecture for different classes of application services. After that, we describe how our architecture can adapt to environmental changes in order to sustain a required reliability. We show the feasibility of our approach w.r.t. the sustainment of application service dependability based on a simulation experiment in Section 6. Related work is discussed in Section 7 and we close the paper with a final conclusion and some comments on future work.

2 Architectural Principles for Dependability

Established design principles aid in the development of dependable software systems. In this section, a number of these design principles are described which are used in the design of our architecture.

Software systems typically consist of several different parts. Since dependencies between these parts exist, problems occur if one part fails. *Service*-based architectures handle the increasing complexity of today's systems by means of online lookup and binding of services. Services and components [5] share the use of contractually specified interfaces, which makes a black box reuse possible. Thus, a client does not need to know the exact implementation, used fault tolerance techniques, or the service(s) deployment, but only relies on the fulfillment of the interface. The client is working correct as long as the promised functionality of the service is delivered and is, thus, completely decoupled from how the work is processed.

The integral part of a service-based architecture is a *service registry*. The use of such a service registry is a key factor for availability, since service instance connections are not hard-wired and the execution location of services is not fixed. Instead, services can spontaneously connect to recover from failures. One example of a self-healing service-based architecture is the Jini architecture [6, 7]. It has been specially designed to support the development of dependable distributed systems (cf. [8]). One of its features is a *lookup service* that remains operational, even when single nodes in the network have crashed, due to redundancy and replicated data by the usage of multicast messages.

The *leasing* principle extends the allocation of resources with time [9]. The lease represents a period of time during which the resource is offered. Therefore, this lease needs to be extended (renewed) if the resource remains to be offered after the timeout of the lease. If the owner of the resource fails to renew the lease, a client can assume that the resource is no longer available. Leasing is the principle which provides the self-healing behavior of the Jini lookup service. Every service registration on the lookup service is accompanied by a lease. If this lease expires, the lookup service removes the accompanied service registration from its lookup tables. Thus, no service gets this apparently failed service instance in response to a search request. If this service is restarted or the communication system is repaired, the service can re-register on the lookup service.

A *proxy* provides a surrogate or a placeholder for another object [10]. In distributed systems a proxy typically acts as a local placeholder for a remote object encapsulating the forwarding of requests via network communication (e.g. as the stub in Java Remote Method Invocation (RMI) [11]). In the Jini architecture the proxy pattern is an inte-

gral part of every service. A service is divided into a proxy and an optional backend. The proxy instance is registered in the lookup service. If a service is to be used by a client, the proxy instance is downloaded as mobile code to the client and executed there. Therefore, the service can execute code on the client's side in addition to code on the backend.

Redundancy of service instances is a key factor to achieve a required degree of reliability. A non redundant service is a single-point-of-failure. Thus, in case of a failure of this service or a communication subsystem failure, which results in a network partition, all dependent clients of that service cease to work. In the Jini architecture more than one lookup service can be used. Thus, a failed lookup service does not compromise the dependability of the complete system.

This leads us to the concept of a *smart proxy* [12, 13]. A smart proxy is not restricted to forwarding but can be used much more flexible. Thus, in the context of reliability the proxy may communicate with multiple backends at once to recover from or mask failures. A client trying to use a service, which has failed although its lease time has not expired, would experience a failure in a proxy-less environment. However, after being downloaded to a client a mobile code proxy, which is stored in the lookup service and thus fails independently from its backend, can revert to another service backend when it detects the failure of its own backend. Hence, a smart proxy can be used to encapsulate and hide the complexity of fault-tolerant code and therefore the use of complex concepts becomes transparent to the user of the service. For example the service registration in the Jini architecture is sent to all available lookup services by the proxy at once using multicast messages. Fault tolerance source code can be integrated in *one* smart proxy, since the proxy is executed in the same process as the using service. Therefore, the smart proxy cannot fail independently from the using service in case of failures. Thus, the using service does not need to include fault tolerance measures in the collaboration with the smart proxy.

Analogue to the redundancy of services a key point for dependability is the availability of data in a distributed system. This can be achieved by the use of *replication*. Replication is the process of maintaining multiple copies of the same entity at different locations. In the Jini architecture the service registrations are replicated in multiple lookup services.

The maintenance of these distributed copies depends on the required consistency for the entity. There exist different *consistency* models (for an overview see [14]). A consistency model provides stronger or weaker consistency in the sense that it affects the values, a read-operation on a data item returns. There is a trade-off between consistency and availability and no optimal solution can be given. The weaker the consistency model the higher availability can be achieved. The possibility to use different consistency models for different data aids in the development of a self-healing architecture as we will show in the next section.

Control theory [15] is a fundamental concept which also permits to improve the dependability of a service-based system. However, in the original domain usually a quasi-continuous rather than a periodic event-based communication between the controller and the controlled subsystem is assumed. For our application, we employ instead an event-based communication. Then, a *monitor* which periodically checks that a service

is alive and restarts a new one if not, can be understood as a simple form of controller. If the parameters of the controller are adapted by measuring certain outputs of the system under control we have an *adaptive controller*. If parameter identification for a given dependability model is used to predict the controller parameters, we even have a *self-optimizing controller* [15]. If the required system goals w.r.t. availability are known, this approach can be employed to realize the system which optimally uses its resources to realize these goals.

3 Design of the Architecture

In this section we will show the application of the introduced architectural principles in our architecture. We give a short introduction of the presented architecture and the requirements of the different infrastructure services. The more detailed design and the implementation is available in [16].

The Jini architecture supports ubiquitous computing in ad-hoc networks and provides a dependable infrastructure for service lookup and operation. However, the basic infrastructure only avoids to provide any element that can compromise the dependability of application components. But to achieve the required dependability for any specific service or application remains to be realized by the application developer. Our proposed architecture in contrast permits to configure the required degree of availability for application services at deployment-time.

A key to the improved availability of the infrastructure services is the idea to have redundant instances of every service type running concurrently in the system to prevent a single-point-of-failure as proposed in the last section.

The overall architecture is built on top of Jini. In addition to the Jini lookup service, four different service types are used. In the following we will describe these different services. Ideally on every computation node of the system one instance of each infrastructure service is executed and will be restarted automatically during each reboot (see Figure 1). In addition on each node of the system a Jini lookup service is executed.

On every node of the distributed system an instance of the *node* service is running. Using this node service, new application service instances can be created (and old ones stopped). Additionally, the node service provides information about the node itself. This includes but is not limited to: ip-address, hostname, total memory, free memory, current load, load average.

A *service description storage* contains service descriptions (like name of the service, package-path, required and provided interfaces, deployment constraints, etc.) for all application services which have to be executed. This information has to be read prior to (re)starting them. Each instance of the service description storage contains one replica of the service descriptions. A strong consistency model for these data is required since a weaker consistency model would result in a possible loss of service descriptions in case of failures. This in turn would cause the unavailability of the affected application services, since no repair is possible due to the lost service descriptions. Thus, reading the descriptions is more important than writing. Additionally, read access to these service descriptions occurs more often than write access.

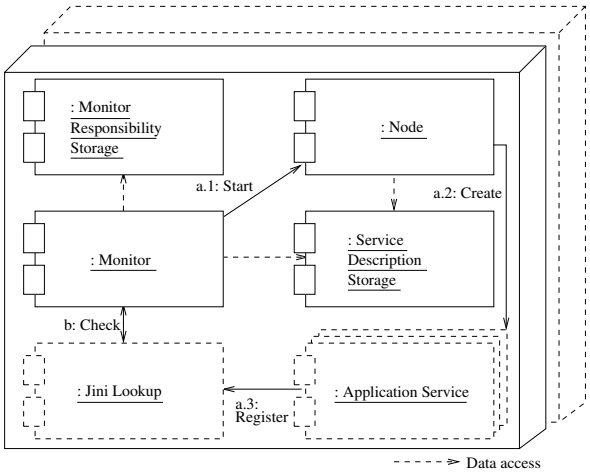

Fig. 1. Architecture design overview

Monitors supervise that the application services contained in the service description storage are indeed executed and available. The availability of the services will be checked periodically. The detection speed of service failures, and thus the mean time to repair (MTTR), can be configured by changing this period. The monitor and the related service instance thus form a closed control loop. To control which monitor is supervising which service, monitors need to acquire a responsibility for a service (i.e. to assure a new instance is started, if a service instance is not available). Thus, another overlaying control loop is used to monitor the monitors themselves. Monitors use the information provided by the node services to choose on which node a certain service will be started.

The monitoring responsibilities are stored in a *monitor responsibility storage*. Responsibilities are accompanied by a configurable lease, which is used to detect failed monitors (i.e. the lease times out). The responsibility lease influences the mean time to repair of the application services, too. Each instance of the monitor responsibility storage contains a copy of these monitor responsibilities. Inconsistencies between these copies only result in changed responsible monitors and potentially additional started service instances. Therefore, we trade overhead for improved reliability and weaken the consistency requirements for these copies. Additionally, after a repaired network partition failure merging the responsibilities in the former partitions must be possible. The monitors whose behavior depends on these responsibilities must be able to cope with a weaker consistency model.

After this overview about the different parts of the architecture, we further describe in more detail the design of two infrastructure services. These services serve as examples how to achieve the required degree of reliability for services, which deal with data, by applying the aforementioned (see Section 2) principles for dependability.

3.1 Service Description Storage

The service description storage contains the descriptions of the services which must be available in the system. These descriptions are replicated in the system on a number of service backends. A strong consistency model is required for this replication. Write operations are only executed by an administrator whereas read operations are regularly executed by the infrastructure services.

Since changes in the service descriptions occur rarely, the number of read operations on these descriptions surpasses the number of write operations. For a certain degree of the system's reliability, it is necessary that the read operations of the infrastructure services succeed with a very high probability in case of failures whereas the write operations are relatively unimportant. To exploit this bias for read operations we have chosen to implement the *weighted voting* approach [17] which provides sequential consistency.

This approach offers the possibility to configure the reliability, based on the assumed distribution of read and write operations. Each node has a number of votes to weight its data specified by the administrator. This number of votes should match the reliability of that node. Quorums are used to determine which nodes participate in a read/write access based on their votes. The weighted voting approach uses a voting where the needed read (n_r) and write quorums (n_w) can be adjusted as long as read-write quorums ($n_w + n_r > n$) and write-write quorums ($2n_w > n$) overlap to prevent inconsistencies (n : total number of votes). For our scenario we chose a high n_w and a low n_r to achieve a high probability for a successful read operation.

Multiple node failures can be masked as long as the required number of votes is available to reach the required quorum. In case of a network partition, read operations are possible in every partition containing more than n_r votes. Write operations are only possible in the rare case that during a network partition one partition contains more than n_w votes. Since we chose a low n_r value, the probability is high that read access is possible in all partitions. We state the reasonable requirement that the network partition must be repaired, before a write access by the administrator is possible.

The weighted voting approach is implemented in the service's smart proxy. Thus, a using service does not need to know about the specific implementation; it just calls read and write operations on the proxy and all replication and consistency management is done internally. JavaSpaces [18] are used as backends. JavaSpaces are a tuple-based storage [19] implemented as Jini services. Therefore, they exhibit the aforementioned principles like leases. Additionally, JavaSpaces can be used as participants in the Jini implementation of the Two-Phase-Commit-Protocol [20]. The backends store the data and an associated version number.

The smart proxy does a lookup on the lookup service gathering the required number of available JavaSpaces for a read respective write access. Executing a write access, the proxy simply writes the data in all gathered JavaSpaces using the Two-Phase-Commit-Protocol and increments the version number. Executing a read access is slightly more complex. After gathering the correct number of JavaSpaces, the smart proxy reads the data of every JavaSpace and then internally selects the data, which has been written by the last write access. This selection is based on the highest version number.

3.2 Monitor Responsibility Storage

Storing the monitor responsibilities is a problem similar to storing the service descriptions. In contrast to that, here write and read operations are equally important. In case of failures it is necessary that another monitor can take over the responsibility of a broken monitor and needs to store this information in the responsibility storage.

Therefore, we can weaken the consistency requirements for the responsibility storage to be able to read and write to it anytime especially in the failure case. An appropriate weaker consistency model is *eventual consistency* [14]. Eventual consistency demands that in absence of write operations the storages eventually stabilize in a globally consistent state after a certain amount of time.

Our approach to achieve eventual consistency is based on multicast messages and a decentral majority voting on every responsibility storage in the network. Because of the multicast messages, every message is received by every storage. Thus, in case of a read operation, all available storages receive the read request and respond by returning their local data as a multicast message. Therefore every storage and the requester get the responsibilities stored in every storage. Since the number of storages is unknown in case of failures, a timeout is used to finish waiting for responses. After that, all storages and the requester do a decentral majority voting on the received data. In case of parity each participant chooses the data with the highest hashcode to achieve a consistent result. A write operation simply sends a write multicast message which is processed by all storages, which receive the message.

Before a globally consistent state is reached there may exist local inconsistencies. For example, during a network partition failure the local data in the storages in the different partitions diverge because updates are only visible within one partition. After the failure is repaired the conflicts between all partitions are resolved by the next read operation. After the decentral majority voting the data of only one partition holds, the others are discarded. Therefore only one monitor is responsible for a specific service description. All other, former responsible monitors notice their responsibility loss on their next responsibility check.

From a user point of view this complex dealing with multicast messages and the voting is completely encapsulated within a smart proxy.

4 Ensuring availability of application services

After presenting the design of the architecture and the individual services, we show how the architecture achieves availability for application services in case of node failures and network partition failures. According to [21] availability is the probability that an item will perform its required function under given conditions at a stated instant of time. Since systems without repair have in the long run an availability of zero, we have to repair the services if they have experienced a failure.

4.1 Qualitative evaluation

In case of a node failure different scenarios, w.r.t. failures of a responsible monitor and monitored application services, are possible. The case that neither a responsible monitor

nor a monitored service is affected by the node failure is trivial. If a node is affected by the failure, which does host only application services, the monitors responsible for these application services will detect the services' failures because the application services do not renew their leases with the lookup service. The monitors will choose new nodes for the application services and start new instances there. Figure 2 shows this scenario. Note, the displayed monitor and lease periods, which influence the achievable degree of availability.

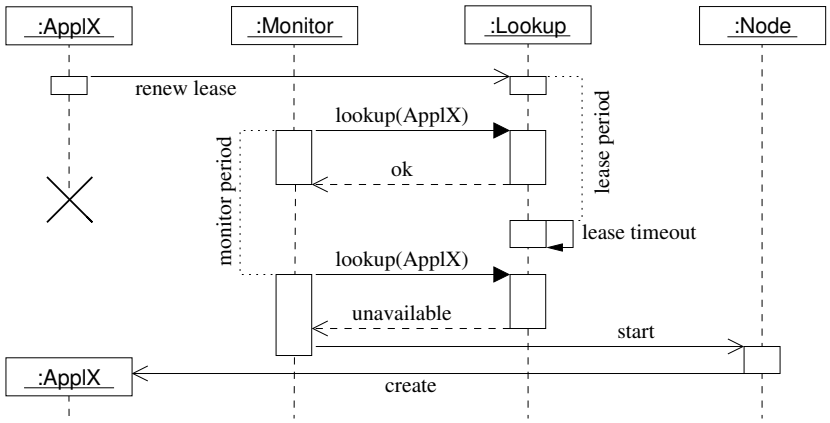

Fig. 2. Scenario of a node failure

In the case of failures in the responsible monitor as well as in the monitored service, the responsibility lease of the monitor expires and another monitor takes over the supervising responsibility. This monitor replaces the failed monitor, and starts supervising the currently unmonitored service which includes starting new instances when needed. Figure 3 shows the events in a sequence diagram, in the case that a service and its supervising monitor are executed on the same node, which experiences a failure.

During a network partition failure, communication is only possible inside the different partitions and no communication can cross the borders between the different partitions. A monitor, which has been responsible for services in the complete network, is in one part of the system during the partition. In this part the monitor recognizes the absence of some monitored services and restarts them. In the other parts the monitor's responsibility times out, other monitors step in, and restart all needed service instances. Thus, in each partition a responsible monitor and all service instances are available after a certain amount of time (cf. Figure 3). If the partitions become too small during a network partition to execute all services, a solution might be to execute only important services due to some specified service levels.

After reuniting the different partitions, the responsibility storages are merged by the above described decentral majority voting to determine a new unified responsible monitor. This new monitor takes over the responsibility for the service instances started by the other responsible monitors in the other partitions. Additionally, it can consoli-

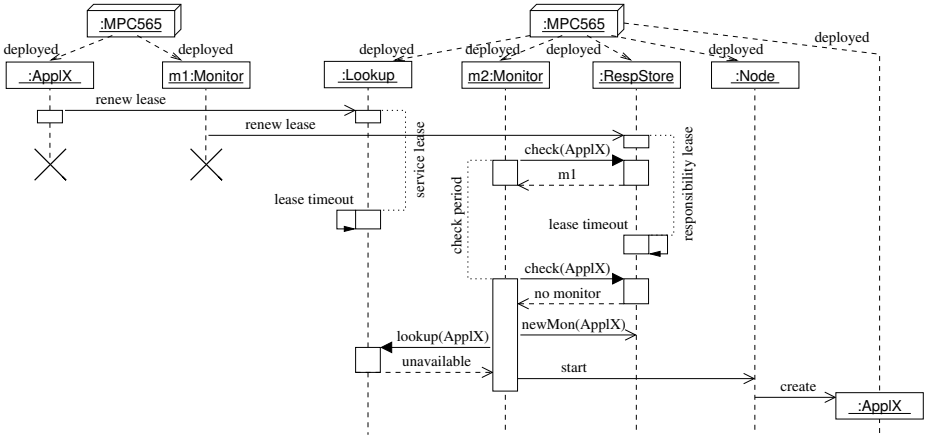

Fig. 3. Timing of a monitor failure

date the number of service instances in the reunited network by asking some service instances to shut themselves down in a graceful manner. The monitors formerly responsible in the other partitions stop monitoring their started service instances after noticing their responsibility loss.

4.2 Quantitative evaluation

In the previous section, we presented how the architecture is designed to improve the availability of application services in case of failures of application services or architecture services. In the following, we show how the availability of application services can be analytically determined using a formal model. The behavior of the architecture and application services depends on a number of parameters. Obviously, the values of most of the relevant parameters change over time and therefore can only be approximated by their expected mean values.

A first category of parameters are *configurable parameters*, which allow to adjust the system behavior to the required bounds:

Monitoring period p_{mean}**:** As described in Section 3 the monitor checks periodically, whether all of its monitored service instances are executed. Decreasing this monitor period leads to faster recognition of service failures but higher resource consumption.

Monitor responsibility lease m_{mean}**:** A lease is attached to each monitor responsibility. Decreasing this lease leads to faster recognition of monitor failures but higher resource consumption.

Service registration lease l_{mean}**:** A lease is attached to the registration of each service in the registration service (e.g. Jini lookup service). Decreasing this lease leads to faster recognition of service failures but higher resource consumption.

Number of monitors mp**:** The number of monitors is crucial to the dependability of the solution. Increasing the number of monitors decreases the probability, that all monitors have failed, but increases the resource consumption.

Number of service description storages d**:** The number of service description storages affects the availability of the service descriptions. Increasing the number of service description storages leads to a higher probability, that the service descriptions are available, but increases the resource consumption.

Number of application service instances i**:** The number of concurrent application service instances directly affects the availability of the application service. Increasing the number of concurrent instances leads to a higher availability, but increases the resource consumption.

Since availability is determined by $MTTF/(MTTF + MTTR)$ (cf. [21]), we can reach higher availability by reducing the mean time to repair. We can configure the MTTR by changing the lease given by the Jini lookup service, the monitoring period and the responsibility lease. Therefore, the proposed architecture can be customized for the required degree of availability subject to the condition that the number of working monitors, service description storages is sufficient.

Two other parameters affect the availability of application services. These parameters are *system parameters*, which are imposed by the environment, and therefore cannot be changed by the architecture. They have to be estimated up front and may later be identified at run-time using our model (e.g. in a self-optimizing controller):

Mean operating Time Between Failures (MTBF) b_{mean}**:** It is affected by the underlying hardware and software platform.

Hardware Mean Time To Repair (MTTR) r_{mean}**:** The time needed to repair a broken hardware platform on which monitors and service description storages are executed.

As it is apparent from the above parameter descriptions, unreflecting changes to the parameters lead to either low availability results for the application services or extraordinary resource consumption. A quantitative analysis is required to determine appropriate values for the configuration parameters for a required availability of application services and to permint system parameter identification. For this purpose we use generalized stochastic Petri nets.

Generalized stochastic Petri nets (GSPN) [22] can be utilized for quantitative analyses of the dependability of models (cf. [23]). A steady-state analysis of the modeled behavior gives quantitative results for the availability of a single service group managed by the architecture. We give a short introduction to GSPNs in Appendix A.

We model the architecture and some application services in a certain example scenario w.r.t. crash failures. In our example scenario, we have three instances of a certain application service group which are executed by the architecture. Further we will use the term service groups for one group of identical application services. We use the model to compute the probability, that at least one application service instance of the service group is working ($p0$), i.e. one service instance is sufficient for performing the required function (1-out-of-3 redundancy). The timed behavior of the model is determined by the above described dependability parameters.

$$p0 = p\{\#working\_instances > 0\}. \quad (1)$$

In Figure 4 the model of our example scenario is shown in form of a generalized stochastic Petri net. All architecture parameters (times and quantities), which have been described in the previous section, are used in this architectural model of a group of identical application services.

The net consists of mainly three parts. The lower left part of the system models the different states of the application service instances. An instance can be in three different states: (1) it is working (the token is in state working_instances), (2) it has failed (parameter b_{mean}) but the lease has not expired yet (token in state failed_instances), and (3) the lease has timed out (parameter l_{mean}) and the token is in state failure_recognized. We abstract from the small amount of time (compared to the other parameter values) which is required to restart a service on another node. We added two immediate transitions unavailable and available. The first fires, when no service instance is available, and thus the system of three application service instances has failed. The second fires, when there is at least one service available again, and thus the system has been repaired. These two transitions are used to analyze the mean time to failure of the complete system of the three application services.

In the lower right part of the figure the service description storages are modeled. For our example we used five service description storages (5 tokens in state deploymentstorage). Three votes are required for a read quorum. If a service description storage experiences a failure (parameter b_{mean}), a stochastic transition fires and a token moves to the state dead_deploymentstorage. After a certain amount of time (parameter r_{mean}) the deployment storage is repaired and the token moves back to state deploymentstorage.

In the upper part of the model the behavior of the monitors is specified. Initially, one monitor is supervising the application service instances (the token in state monitor). Two other monitors are running but are not supervising any application service instance (two tokens in state monitor_pool). When a monitor fails (either a supervising monitor or a monitor in the pool) determined by the parameter b_{mean}, the token moves to the state dead_monitors. If the supervising monitor fails (no token in monitor or looking_for_dead), another available monitor starts supervising the application services. The transition start_supervising models this behavior. The time value of this transition m_{mean} models the monitor responsibility lease of our architecture.

In the middle of the figure, the restarting of failed application service instances is modeled. The supervising monitor periodically checks whether a service application instance has failed (token moves from state monitor to looking_for_dead). If there is a token in state failure_recognized and enough votes can be gathered from the service description storages, the immediate transition restart fires based on the transition priority 2. If not, the transition no_restart fires to end the supervising period.

In GSPNs stochastic transitions are specified with a rate value. This rate value is the inverse of the mean duration (cf. Appendix A). In Figure 4 the rate values are shown, which are the inverse of the parameters which we described at the beginning of this section. All transitions concerning the failure of services (application and architecture) have the rate $b = 1/b_{mean}$. Typical embedded pc platforms have a MTBF of over 200,000 hours [24]. Since influences from the operating system and other infrastructure

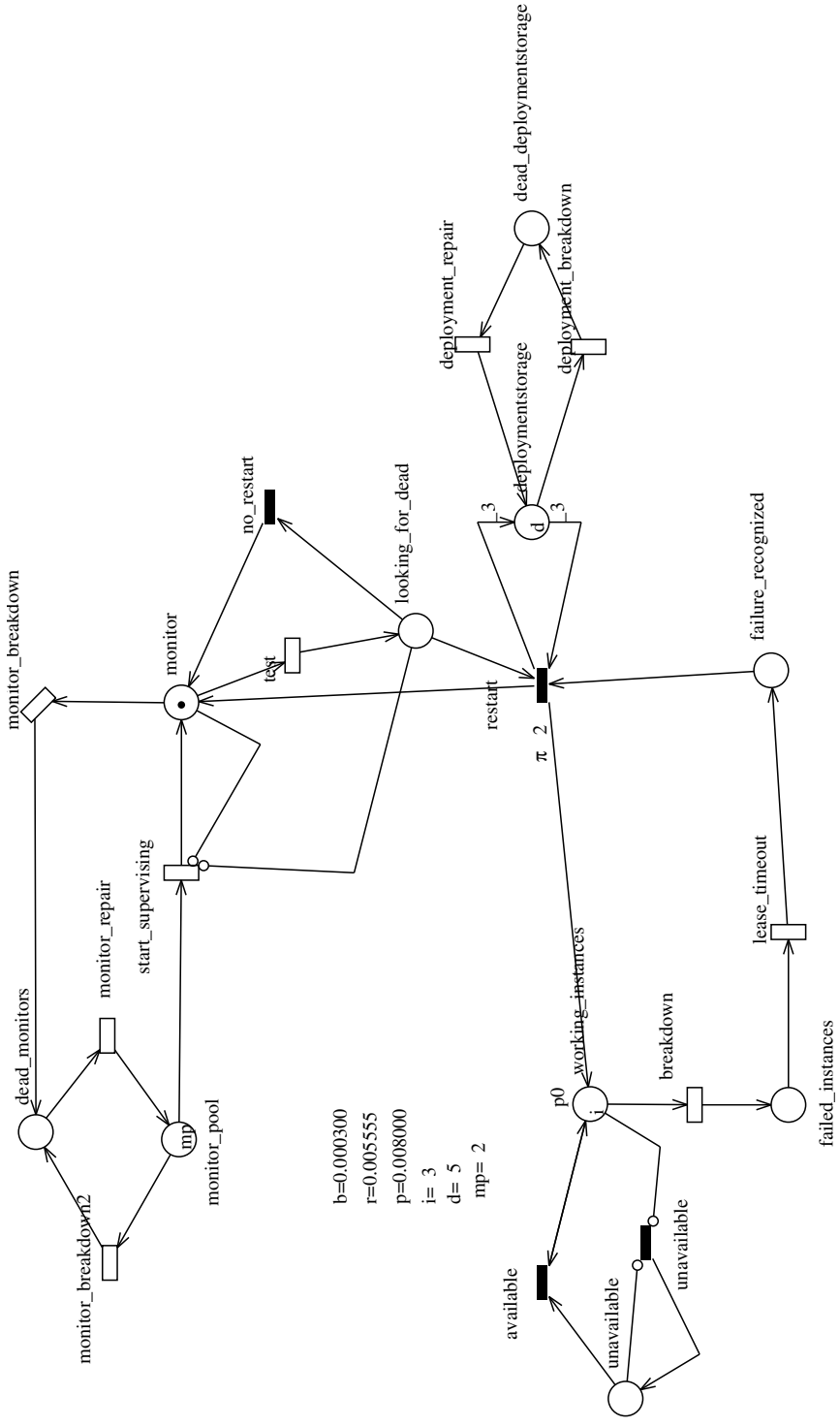

Fig. 4. GSPN model of the architecture

is unknown, we pessimistically assume a MTBF of 3,333 minutes, which equals 2.3 days. Thus, we get a failure rate $b = 0.0003$ (in failures/minute). All transitions concerning hardware repair have the rate $r = 1/r_{mean}$. We assume, that repairing hardware or replacing it is done in a mean time of 180 minutes which leads to $r = 0.005555$ (repairs per minute). The time needed for a repair of an application service instance is based on the values for the lease time (l_{mean}), the monitoring period (p_{mean}), and the monitor responsibility lease (m_{mean}). All these three parameters can be configured to reach a certain availability. Since their cost in terms of network bandwidth and processing power is equal and they equally influence the availability, we use the same value for all parameters: $p_{mean} = l_{mean} = m_{mean} = 125$ minutes. This simplifies the analysis and in the next section the adaption of the controller. Hence, we get the rate $p = 0.008$ (per minute).

We analyzed the GSPN model and got the result, that in the steady-state the probability of at least one working service instance is $p0 = 0.999281$, i.e. the availability of the service group is 99.9281%.

To be able to reach a required availability of the application services for different environmental parameters b_{mean}, we did a steady-state analysis of the model ($p0$) with different values for the parameter (p_{mean}) and the probability of service failures (b_{mean}) computing the probability of at least one working service. Figure 5 shows the results of the steady-state analysis. Note, that the parameter values shown in the figure are the reciprocal values of the means. Considering any curve b, we can improve the availability by reducing the parameter p_{mean} (reducing p_{mean} leads to higher p because of $p = 1/p_{mean}$). We used the GreatSPN tool [22] for modeling the net and the steady-state analysis.

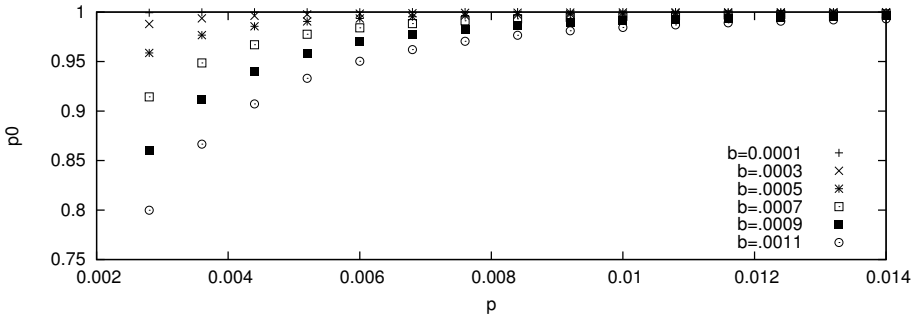

Fig. 5. Parameterized steady-state analysis of the availability

4.3 Controlling the availability

The steady-state analysis computes the availability achievable by the architecture w.r.t. different estimated values for the architecture parameter p_{mean} and the hardware mean time to failure b_{mean}. For the initial deployment we use a reasonable b_{mean} and

the according p_{mean} to achieve the required $p0$, which we computed during the steady-state analysis. Our estimated value for b_{mean} may be wrong. Therefore, if the estimated value for the service failure b_{mean} proves incorrect after deploying the system and executing it, we may adjust the monitoring period p_{mean}, based on the values of the steady-state analysis to reach the same required availability $p0$ despite the incorrect estimated b_{mean}. Additionally, the rate of the service failure may change over the course of time. The architecture parameter will be changed accordingly to sustain the required availability.

However, we can only perform a steady-state analysis for a distinct number of parameter values. The parameter values, gathered during the execution of the system, will not be equal to the values computed by the steady-state analysis. Thus, we need to be able to compute a monitoring period for a service failure rate which we did not consider during the steady-state analysis. We assume that the steady-state results are continuous w.r.t. the parameter values (p and b). Therefore, we may approximate a graph $p0 = f(p, b)$ by using a cubic bi-spline function.

The approximation by the cubic bi-spline function results in a set of polynomial functions. For each patch based on the initial discrete values, one polynomial function is used to compute the p0 value. Figure 6 shows a scaled plot of the set of functions. The figure shows which availability (shown on the z-axis) is the result of the failure rate b and the architecture parameter p. Increasing the architecture parameter p, which increases the repair rate of the architecture, increases the availability $p0$ for a given failure rate b. Vice versa, in case of a higher failure rate b, we can increase the repair rate p to reach a required availability $p0$.

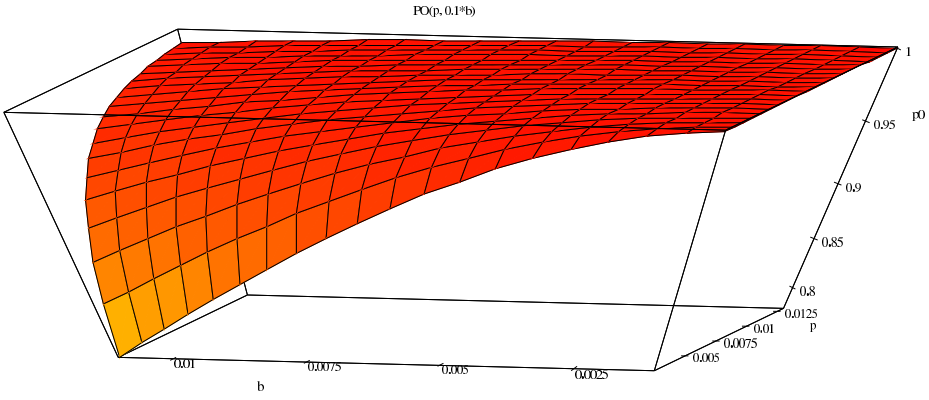

Fig. 6. Interpolated function of the distinct values provided by the steady-state analysis.

Based on this function, we can change the architecture parameter to achieve a required degree of availability in response to the measured changes of the service failure rate. Thus, we are controlling the availability of the system by reading inputs (service availability) and changing outputs (architecture parameter) to reach a certain goal, i.e. we are applying control theory to software systems as proposed in [25]. Our ap-

proach for adjusting the parameter p is a special feedback adaptive controller named *model identification adaptive controller* (MIAC). Following [15], it is also named a *self-optimizing controller*.

As depicted in Figure 7, the basic control loop in the system is built by the service group and the monitor. The service group represents the controlled process. It can be influenced by the monitor by restarting crashed services. The feedback from the process to the monitor is only implicitly present in the architecture. The monitor periodically checks via the lookup service if a specific service has not crashed.

The availability of the application service group is influenced by the service failure rate b. The measured availability of the application services $p0$ is fed into the parameter identification unit. Using the pre-computed approximation of $p0(p, b)$ the related service failure rate b can be determined and propagated to the parameter adaptation unit. There, the approximation of $p0(p, b)$ is again used to determine the parameter p to reach the assigned availability goal $p0$.

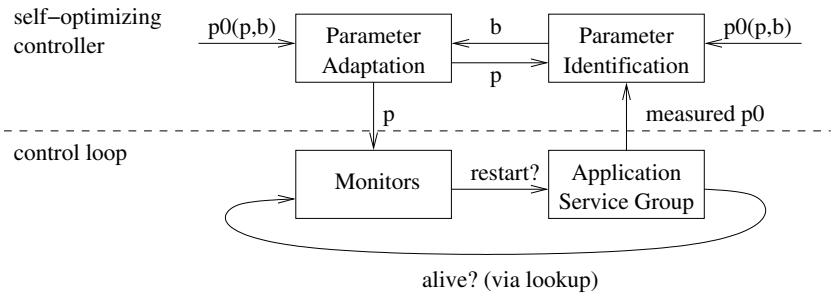

Fig. 7. Self-optimizing availability controller

5 Ensuring reliability of application services

As outlined in the preceding section, the presented infrastructure of the architecture can ensure a required degree of availability for a group of application services by restarting failed ones. Due to failures, application services are failing over time and are restarted on other nodes in the network by the architecture.

Reliability is known as the probability that an item will perform its required function under given conditions for a stated time interval [21]. Thus, we can also aim at raising the reliability of the application system by employing redundancy. As long as the failure rate of the services does not exceed a certain level, which would make it impossible to synchronize the redundant service instances, we can use the available redundant services to fulfill the incoming service requests.

5.1 Qualitative evaluation

For each class of application services we have to reflect the specific condition when the service will fail.

For stateless session services it is irrelevant which service instance is used for a given action, since the actions are independent of each other. If a service instance fails, simply another instance can be used. It is not needed to keep the service instance consistent with each other, since there is no state which needs to be shared between the individual service instances.

If a stateful session service instance fails in the middle of a session, just using another service instance from thereon does not work. Essentially, the last state of the failed service instance must be recreated on the newly used service instance. Thus, the history (relevant actions) until the point of failure needs to be replayed. As a smart proxy will only crash together with the client employing it, we can rely on this principle to ensure that the client is able to replay the history on its own as long as at least one application service instance is available.

For obtaining reliable entity services, it is necessary to replicate the entity and to distribute the replicas among a number of service instances to be able to tolerate failures. Additionally, the consistency of the entity according to a suitable consistency model must be assured. The required consistency model is application-specific and for every degraded service level a different number of active application service instances might be required. For standard consistency models like sequential consistency, application independent replication algorithms may be used, which provide the required consistency model. For example, in our architecture we have data with two very different requirements (service descriptions and monitor responsibilities) which can be provided by appropriate consistency models (see Section 3.1). For the service descriptions we have two different service levels. Either we have the full service including read and write access or the degraded service read access only.

The implementation of the above mentioned concepts leads to a system where reliability can be configured to some extend. But the maintainability[1] of the resulting system deteriorates due to the increased application of fault tolerance techniques throughout the system's source code. To overcome the situation that each client of a service must include the appropriate fault tolerance code, we apply in our architecture the smart proxy pattern (see Figure 8).

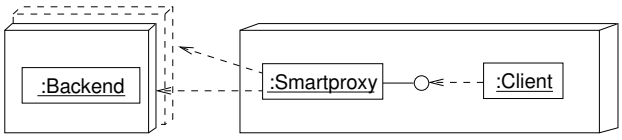

Fig. 8. A smart proxy hides the employed fault tolerance scheme

[1] "the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment" [26]

5.2 Quantitative evaluation

For an appropriate model of the reliability of a specific group of application services the same general dependability characteristics as employed in the GSPN model of Figure 4 have to be considered.

For a specific group of redundant service components we can model the expected reliability using a GSPN model such as in Figure 4 where additionally the failure rate λ of the service failure condition for the whole group is analyzed. Note, that this condition is not the failure rate of the single service instances but a condition on the whole set of service instances which is specific for the class of service and the employed redundancy scheme.

Using a steady-state analysis of the model we can then derive the failure rate. Due to the steady-state result we can assume that the measured mean transition throughput of a set of transitions detecting the failing of the service actually corresponds to a constant failure rate λ which describes the rate at which the whole group of services is failing to provide the required service.

At first we can ensure a required minimal mean time to failure. Using the formula $\mathrm{MTTF} = 1/\lambda$ for a system with constant failure rates ($\mathrm{MTTF} = \int_0^\infty R(t)$) we can derive the required upper bound $\lambda \leq 1/\mathrm{MTTF}$ on the failure rate. The value of λ is dependent on the configurable architecture parameter p. Thus, by changing p we can indirectly influence the mean time to failure.

Secondly, to ensure the required minimal reliability r for a given time interval of length l such that $R(l) \geq r$ holds, we can use the formula $R(t) = e^{-\lambda t}$ for the case of a constant failure rate to derive an upper bound for the failure rate with $\lambda_r \leq -\ln(r)/l$.

In the same manner as the availability value $p0$ can be computed using the GSPN model, we can thus use the steady-state analysis results of an appropriately extended GSPN model to derive the required results for the service failure rate of the group of services.

For stateless session services any service instance is sufficient. To derive the failure rate for this simple case we have thus added to the GSPN model in Figure 4 an additional place unavailable and transitions available and unavailable which model the event that none of the service instances is available. For the example configuration presented in Figure 4 the throughput of the transition unavailable determined by the steady-state analysis is 0.000005, i.e. the $MTTF_S$ for the complete system is $1/0.000005 = 200,000$ minutes, which is 138.8 days. The resulting reliability over a month is thus $e^{-0.000005*43,200} = 0.805735$.

For a stateful session service with a smart proxy which records the relevant history, the same extension of the GSPN model as used for the stateless case can be employed for analysis. The reliability for an entity service group, where a simple majority voting of the service instances is employed such that at least two of all three service instances have to be up, can be analyzed by extending the GSPN model of Figure 4 as follows: Two transitions unavailable1 and unavailable0 for the case of only one or zero available services and their combined transition rate has to be used to model the failure of service provisioning. Transition available has to be replaced by two transition available2 and available3 which fire when two or three of the service instances are available. It is to

be noted that a different employed redundancy scheme would also result in a different required extension of the GSPN model.

5.3 Controlling the reliability

Like in Figure 7, we can exploit an approximation of the parameterized GSPN model to realize a self-optimizing controller on top of the basic control loop build by the service group and the monitor. The required changes to the infrastructure parameters are derived from the parameterized model for multiple identical services at run-time by measuring the failure rate $r0$ of the service group.

The reliability of the application service group is influenced by the crash failure rate b. Equivalent to the controller loop used for ensuring the availability $p0$, the measured failure rate of the whole application service group $r0$ is used in an additional parameter identification unit where the pre-computed approximation of $r0(p, b)$ is used to estimate the single service failure rate b. In the parameter adaptation unit this estimation and the $r0(p, b)$ is then used to compute the parameter p to reach the assigned reliability goal $r0$.

In Section 4.3 we presented a controller for the availability of the application service group, which controls the availability for not falling below a certain required level. If we employ both controllers, we may get different values for the architecture parameter p. In this situation we simply choose the higher value. Consider a situation where the controllers compute two values p_r and p_a to ensure a required level of reliability respective availability. If $p_r > p_a$ holds, $p = p_r$ is used for controlling availability *and* reliability. Since a higher value for p than p_a is used for controlling the availability, the availability of the application service group will be higher than required.

6 Evaluation

We presented in the last sections an approach to control the availability and reliability of application services executed by the architecture in the case of changes to the crash failure rate due to external influences. In this section, we present results of a number of simulation experiments, which show that (1) our GSPN model is a correct abstraction of the simulation and (2) the controlling approach presented can indeed ensure the requested availability and reliability in case of changes to the crash failure rate.

For the simulation experiments, the scenario of the GSPN model of Section 4.2 is used: a service group of 3 application service instances is executed by the architecture, 1 running application service instance is sufficient for working (1-out-of-3 redundancy), 3 monitors are used to supervise the service group, 5 deployment storages contain the service group deployment information of which 3 must be working (3-out-of-5 redundancy).

The simulation is event driven, i.e. for each event a distinct trigger time is computed when this event will happen. This trigger time computation is either based on a negative exponential distribution or based on deterministic periods. Events are (1) crash failures, (2) repair of infrastructure services, (3) lease expirations of application service instances, (4) monitor checks, and (5) monitor responsibility checks. The simulation

sequentially processes the events in the order of the trigger time. During processing an event, new events are created, e.g. processing a lease expiration event creates a monitor check event. The simulation skips all time steps between events, since no change to the system is expected to occur between events.

In the course of the evaluation, 75 simulation runs have been conducted each running for 100,000 events. Due to the stochastic distributed event trigger times, the simulated total time of each simulation run varies. The mean simulation time for a simulation run is 12,551,511 minutes, which roughly equals 24 years.

The availability values gathered during the simulation experiments and computed in the GSPN analysis are very similar. In addition the results of the simulation run are slightly better than the analysis results of the more abstract GSPN model in the majority of the simulation runs. Table 1 shows the results for an architecture parameter $p = 0.0092$ and the different values for the crash failure rate b for different simulation runs.

Table 1. Availability results of simulation and GSPN analysis for $p = 0.0092$ and different b

	0.0001	0.0003	0.0005	0.0007	0.0009	0.0011
GSPN	0.999980	0.999516	0.997921	0.994637	0.989166	0.98102
Simulation	0.999985	0.999710	0.997589	0.994958	0.989306	0.97026

Concerning reliability, Figure 9 shows the reliability curve for different time intervals t gathered during the simulation experiment. During a simulation run, the time intervals x_i in which the application group performs its required function are stored in the set M. The set $N_t = \{x_i | x_i > t\}$ contains all time intervals x_i which are longer than the time interval t. The points depicted in the figure are computed as $R(t) = |N_t|/|M|$, i.e. the fraction between the number of time intervals bigger than t and the total number of time intervals. Hence, these individual points $R(t)$ depict the experimental results for the probability that the application group performs its required function for at least the time interval t. In addition the mean time to failure of the simulation experiment and the MTTF computed by the GSPN analysis are included to show, that concerning reliability our GSPN model is a good abstraction of the simulation.

In Sections 4.3 and 5.3, we presented the idea to use the values computed by the GSPN analysis to control availability and reliability in case of changes to the crash failure rate b. To support this claim, we conducted simulation runs, where we externally change the crash failure rate b during the simulation run. Figure 10 shows the plot of one simulation run. The average availability of the system over a time frame of 1,000 minutes is displayed as a line. The reconfigurations done by the controller are displayed as small crosses. In this run, we started with the crash failure rate $b = 0.0003$. Note, that we externally induced an increase of the crash failure rate b at time 1,114,365 (visualized by the arrow) to $b = 0.0009$. The controller is set to reconfigure the architecture parameter p, when the availability falls below a required level of 0.99. It is apparent from the figure, that after the externally induced increase of the crash failure rate, the availability falls below the required level of 0.99. Thereafter the controller reacts by

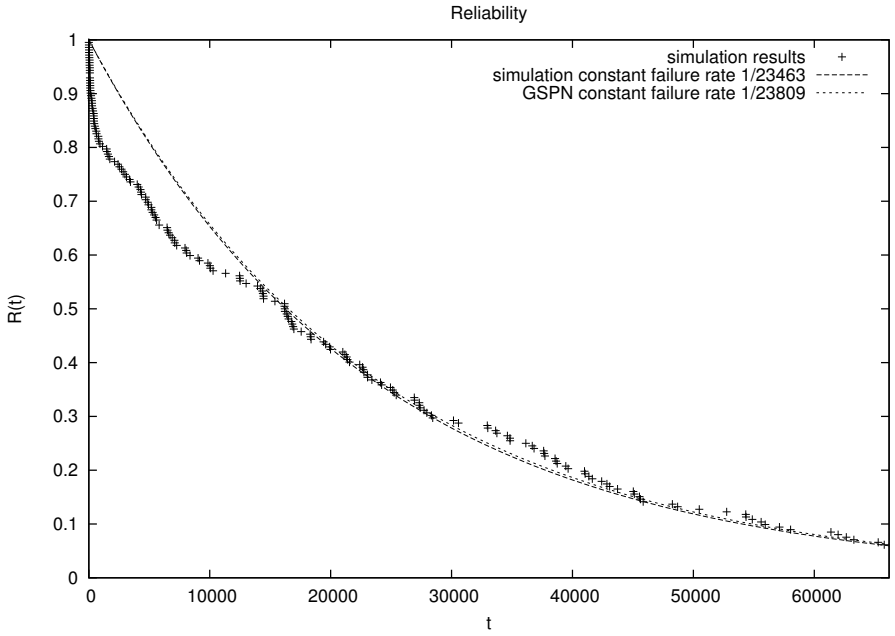

Fig. 9. Reliability results of simulation and GSPN analysis for $p = 0.0092$ and $b = 0.0007$

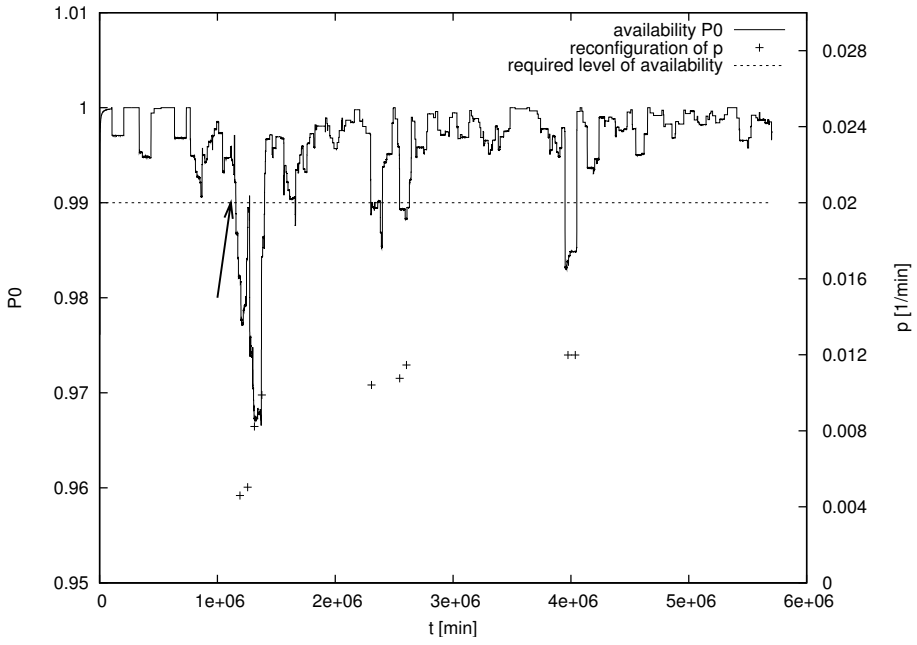

Fig. 10. Availability results showing the reaction to induced changes of the crash failure rate

increasing p, and the availability rises again above 0.99. Due to the random distributed crash failures in the simulation, there have been three additional points in time, where the availability falls below 0.99 and the controller reacts again by increasing the architecture parameter p.

In this section, we have shown by the extensively conducted simulation runs, that (1) our GSPN model is a good abstraction w.r.t. the simulation and based on that (2) our controller can indeed ensure the availability of the application service group in spite of external increases of the crash failure rate.

7 Related Work

In the Jini-context the problem of availability is somewhat addressed by the RMI-Daemon [11]. This daemon supports the on demand creation of remote objects. Additionally if the node fails, after a reboot and a restart of the daemon all remote objects are recreated. Nevertheless this daemon only restarts the remote objects on the same node. Therefore, this is not a solution if a node fails permanently or if the remote objects should be available during the repair of the node.

The RIO-Project [27] uses a somewhat similar approach compared to ours. One single monitor is loaded with the service descriptions and ensures the availability of the contained services in the system. The fact that the service descriptions are only available inside of the monitor makes the monitor a single-point-of-failure in the system. If the monitor process fails, the service descriptions are lost since they are not replicated. No other monitor can use those service descriptions and replace the existing monitor without manual intervention. Thus the reliability of the RIO approach depends heavily on the reliability of one monitor instance. Additionally during a network partition failure the approach does not work since the monitor instance cannot be in more than one partition of the network. Hence this approach is not applicable for dependable systems.

The Master-Slave pattern [28] can be applied when services are replicated and a result must be selected which is returned to the client. This is similar to our smart proxy approach. In our approach the slaves are the different service instances and the smart proxy is the master. The Master-Slave pattern is aimed at stateless session services whereas our approach can also be used for the consistent management of entity services.

The Willow-Architecture by Knight et al. [29] provides survivability for critical distributed systems. As a response to faults reconfiguration is used to ensure the survivability of the system. The response to failures is based on a monitor/analyze/respond-control loop which is similar to our behavior of the monitor.

The SAFEGUARD project [30] aims at a scalable multi-agent environment for maintaining an acceptable level of system's operability. Anomaly Detection Agents initiate countermeasures in response to bad events in order to maintain the acceptable operability level. Those agents employ *learning* algorithms (case based reasoning, neural networks, etc.) to detect previously unknown bad events. Voting schemes are used to decrease the probability that false positives are detected or anomalies are not detected. In the SAFEGUARD project the quite ambitious effort is proposed to employ artificial intelligence approaches for fault detection. In contrast we use self-optimizing behavior to recover from known faults.

Kon et al. present in [31] the distributed middleware 2K which is built on top of standard operating systems and CORBA [32]. The 2K middleware provides means for dynamic instantiation of components including management of dependencies. To react to changes in the environment, 2K uses mobile reconfiguration agents which contain reconfiguration commands and/or new implementations for system and application commands. The resource management component of 2K uses also lease times as means for the detection of resource unavailability [33]. While fault tolerance is addressed for the 2k middleware components, for application components fault tolerance and reliability is left to the developer.

Gustavsson and Andler describe in [34] a distributed real-time database which uses eventual consistency. Similar to our approach they use this consistency model to improve the availability and efficiency and to avoid blocking for unpredictable periods of time.

The general approach to analyze the reliability in the presence of maintenance activities such as the restart of services by the monitors with models is restricted to fixed structure and derives one large GSPN (cf. [23]). In the presented approach in contrast dynamic structures are supported and GSPNs are only build off-line for each service group to determine the general characteristics. This model is then used at run-time to adapt the parameters of the monitors to sustain the required availability. The required availability is itself adapted taking the service dependencies into account.

8 Conclusions and Future Work

For the presented architecture implemented on top of Jini, we have shown that the infrastructure services itself build a dependable system. This includes that in contrast to related proposals no single-point-of-failure for node crashes or network partition failures is possible. For different kinds of application services we presented appropriate concepts to also realize a higher reliability. The required additional efforts for availability and reliability are systematically hidden to the service clients using the smart proxy concept of Jini.

As the number of parallel running service instances and lease times for registry and monitoring can be freely configured, the architecture permits to adjust the deployment and maintenance scenario for a given application service such that the required availability and reliability can be achieved by determining the parameters using the presented quantitative dependability models.

Additionally, we used the results of the formal analysis to react to run-time measurements of dependability characteristics. Thus, we adjust the system parameters such as monitor supervision periods accordingly to control the required degree of availability or reliability. We supported our approach by extensive simulation experiments.

In accordance with [35], which defines *self-adaptive software* as software that modifies its own behavior in response to changes in its operating environment, we thus classify our architecture as *self-healing software*. Our scheme to employ a model of the system under control to predict the controller parameters is in the control theory community named a *self-optimizing controller* [15]. Therefore, our architecture is a *self-optimizing* system for service-based software that at run-time does not only ensure

self-healing behavior, but also optimizes the system w.r.t. the known required dependability and operation costs.

In addition to the implementation of the presented dependable architecture and its run-time system, tool support by means of UML component and deployment diagrams has been realized [36, 16]. This includes code generation for the services, generation of XML deployment descriptions, and the visualization of the current configuration by means of UML deployment diagrams.

As future work, we want to employ classical approaches for learning to improve our results. E.g., instead of the approximation $p0(p, b)$ or $r0(p, b)$ derived from the GSPN model, the measured relation between the parameters can be accumulated during operation. These selected field data may permit to find more cost effective solutions which still provide the required degree of dependability. When the predictive accuracy of this approach is detected to be not sufficient any more, the system may switch back to the model-based approach outlined in this paper to ensure the required availability in the long run. We plan to conduct real-life tests, to show whether the results of the simulation runs are repeatable under real-life conditions.

Acknowledgments

The authors thank Katrin Baptist, Sven Burmester, Matthias Gehrke, Ekkart Kindler, Matthias Meyer, and Daniela Schilling for discussions and comments on earlier versions of the paper.

References

1. Laprie, J.C., ed.: Dependability : basic concepts and terminology in English, French, German, Italian and Japanese [IFIP WG 10.4, Dependable Computing and Fault Tolerance]. Volume 5 of Dependable computing and fault tolerant systems. Springer Verlag, Wien (1992)
2. Majzik, I., Pataricza, A., Bondavalli, A.: Stochastic Dependability Analysis of System Architecture Based on UML Models. In Lemos, R.D., Gacek, C., Romanovsky, A., eds.: Architecting Dependable Systems. Volume 2677 of Lecture Notes in Computer Science. Springer-Verlag, New York (2003) 219–244
3. Gokhale, S.S., Horgan, J.R., Trivedi, K.S.: Specification-Level Integration of Simulation and Dependability Analysis. In Lemos, R.D., Gacek, C., Romanovsky, A., eds.: Architecting Dependable Systems. Volume 2677 of Lecture Notes in Computer Science. Springer-Verlag, New York (2003) 245–266
4. DeMichiel, L.G., Yalcinalp, L.Ü., Krishnan, S.: Enterprise JavaBeans^{TM} Specification. Sun Microsystems. (2001) Version 2.0.
5. Szyperski, C.: Component Software, Beyond Object-Oriented Programming. Addison-Wesley (1998)
6. Arnold, K., Osullivan, B., Scheifler, R.W., Waldo, J., Wollrath, A., O'Sullivan, B.: The Jini(TM) Specification. The Jini(TM) Technology Series. Addison-Wesley (1999)
7. Sun Microsystems: Jini Specification. (2000) Revision 1.1.
8. Waldo, J., Wyant, G., Wollrath, A., Kendal, S.: A Note on Distributed Computing. techreport, Sun Microsystems Laboratories (1994) TR-94-29.
9. Waldo, J.: The Jini architecture for network-centric computing. Communications of the ACM **42** (1999) 76–82

10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
11. Sun Microsystems: Java^{TM} Remote Method Invocation Specification. (2002) Revision 1.8, JDK 1.4.
12. Koster, R., Kramp, T.: Structuring QoS-Supporting Services with Smart Proxies. In: Proceedings of Middleware'00 (IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing), Springer Verlag (2000)
13. Ledru, P.: Smart proxies for jini services. ACM SIGPLAN Notices **37** (2002) 57–61
14. Tanenbaum, A., van Steen, M.: Distributed Systems, Principles and Paradigms. Prentice Hall (2002)
15. Isermann, R., Lachmann, K.H., Matko, D.: Adaptive control systems. Prentice Hall International series in systems and control engineering. Prentice Hall, New York (1992)
16. Tichy, M.: Durchgängige Unterstützung für Entwurf, Implementierung und Betrieb von Komponenten in offenen Softwarearchitekturen mittels UML. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany (2002)
17. Gifford, D.K.: Weighted Voting for Replicated Data. In: Proceedings of the seventh symposium on Operating systems principles. Volume 7 of ACM Symposium on Operating Systems Principles., ACM press (1979) 150–162
18. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley (1999)
19. Carriero, N., Gelernter, D.: How to Write Parallel Programs. MIT Press (1990)
20. Gray, J.N.: Notes on Database Operating Systems. In: Operating Systems an Advanced Course. Lecture Notes in Computer Science, Springer Verlag (1978)
21. Birolini, A.: Reliability engineering : theory and practice. Springer Verlag, Berlin (1999) 3rd Edition.
22. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. John Wiley and Sons, Inc. (1995)
23. Malhotra, M., Trivedi, K.S.: Dependability modeling using Petri-nets. IEEE Transactions on Reliability **44** (1995) 428–440
24. Advanced Digital Logic Inc.: MSMP3SEN/SEV Datasheet `http://www.adlogic-pc104.com/products/cpu/pc104/datasheets/msmp3sen-sev%.pdf`. (2001)
25. Kokar, M.M., Baclawski, K., Eracar, Y.A.: Control Theory-Based Foundations of Self-Controlling Software. IEEE INTELLIGENT SYSTEMS **14** (1999) 37–45
26. Standards Coordinating Committee of the IEEE Computer Society, The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017-2394, USA: IEEE standard glossary of software engineering terminology, IEEE Std 610.12-1990. (1990)
27. Sun Microsystems: RIO - Architecture Overview. (2001) 2001/03/15.
28. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern Oriented Software Architecture. John Wiley and Sons, Inc. (1996)
29. Knight, J.C., Heimbigner, D., Wolf, A., Carzaniga, A., Hill, J., Devanbu, P., Gertz, M.: The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications. In: The International Conference on Dependable Systems and Networks (DSN-2002), Washington DC (2002)
30. Bologna, S., Balducelli, C., Dipoppa, G., Vicoli, G.: Dependability and Survivability of Large Complex Critical Infrastructures. In: Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2003). Volume 2788 of Lecture Notes in Computer Science., Springer Verlag (2003) 342 – 353
31. Kon, F., Campbell, R.H., Mickunas, M.D., Nahrstedt, K., Ballesteros, F.J.: 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In: Proc. of the Ninth IEEE

International Symposium on High Performance Distributed Computing (HPDC'00), Pittsburgh, USA. (2000)
32. Object Management Group: The Common Object Request Broker: Architecture and Specification, Version 3.0 formal/02-06-33. (2002)
33. Kon, F., Yamane, T., Hess, C., Campbell, R., Mickunas, M.D.: Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In: Proceed. of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001), San Antonio, USA. (2001)
34. Gustavsson, S., Andler, S.F.: Self-stabilization and eventual consistency in replicated real-time databases. In: Proceedings of the first workshop on Self-healing systems, ACM Press (2002) 105–107
35. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems **14** (2002) 54–62
36. Tichy, M., Giese, H.: Seamless UML Support for Service-based Software Architectures. In: Proc. of the International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI) 2003, Luxembourg. Lecture Notes in Computer Science (2003)
37. Reisig, W.: Petri nets - An introduction. EATCS Monographs on Theoretical Computer Science. Springer Verlag (1985)

A An Introduction to Generalized Stochastic Petri Nets

In this section, we give a small, simplified, and informal introduction to generalized stochastic Petri nets (GSPN). We assume that the reader is familiar with s/t-nets (cf. [37]). For further information and a formal introduction for GSPNs see [22].

A GSPN consists of states and transitions. Two ideas merely have been added compared with s/t-nets: (1) the introduction of timed transitions, (2) the association of a random, exponentially distributed firing delay with timed transitions.

(a) Immediate (b) Stochastic

Fig. 11. Different transition types

The first idea leads to the distinction between immediate transitions which immediately fire, if they are enabled, and timed transitions. If a timed transition becomes enabled, it gets a timer which decreases constantly until it reaches zero. Then, the transition fires. The second idea leads to the association of a random, exponentially distributed firing delay with timed transitions. For mathematical reasons negative exponential probability density functions (pdf) are adopted. Figure 11 shows the different types of transitions in a GSPN, which are used throughout this paper.

A stochastic transition is specified using a rate. This rate is the inverse of the mean of the negative exponential pdf. For example, if you want to model an activity with

an average duration of $t = 10$ time units, the respective timed transition has a rate $\lambda = \frac{1}{t} = 0.1$.

Priorities and weights are used to specify how conflicts between two immediate transitions are resolved. Using priorities the transition fires, which has the higher priority. Weights are used to specify the probability of the conflicting transitions. For example, if two immediate transitions with the weights $w_1 = 1$ and $w_2 = 4$ are in conflict to each other, the probability that transition one fires is $p_1 = \frac{w_1}{w_1+w_2} = 0.2$ and the probability that transition two fires is $p_1 = \frac{w_2}{w_1+w_2} = 0.8$.

If an immediate and a timed transition are in conflict with each other, always the immediate transition fires. The resolution of two conflicting timed transitions is based on the facts that (1) the two timers of the transitions are sampled from the negative exponential pdf and (2) the probability of two timers expiring at the same instant is zero.

Generalized stochastic Petri nets can be mapped to continuous-time Markov chains for steady-state analysis. The GreatSPN tool [22] has been used in the paper for this analysis. It is available from `www.di.unito.it/~greatspn/index.html` on request.