# An Architecture for Configurable Dependability of Application Services

Matthias Tichy[1] and Holger Giese[1]
Software Engineering Group
Department of computer science
University of Paderborn
Warburger Str. 100, 33098 Paderborn, Germany

E-mail: [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 – an architecture proposed to cope with the complexity and dynamics of today's systems – we identify in this paper a set of architectural principles which can be used to improve dependability. A service-based architecture which extends Jini and employs the identified architectural principles is further proposed and realized. The dependable operation of the infrastructure services of the architecture further enables to systematically control and configure some dependability attributes of application services.*

## 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. [10]) are the attributes of dependability that are used to describe the required system characteristics. These four attributes in practice often depend on each other. Availability and reliability can in principle be systematically studied at the level of components and their composition in form of specific architectures. The ever increasing system complexity and the increasingly ubiquitous character of computing, however, render such an analysis a difficult task.

For complex systems the required prediction models for availability and reliability become quite complex when maintenance activities and component modifications are also taken into account. When further considering dynamic systems where no static *a priori* known system configuration exists, the analysis and prediction of the reliability or availability is thus usually not possible. We therefore propose to build dynamic and dependable complex systems not by relying on design-based quantitative analysis of its static architecture. Instead the observed obstacles should be addressed by a 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 adaption. In accordance with [12], which defines *self-adaptive software* as software that modifies its own behavior in response to changes in its operating environment, we thus classify it as *self-healing software*.

We further restrict our considerations for dependability on reliability and availability and study how the dynamic management of redundant component instances with identical implementation can contribute to improvements for these two dependability attributes. The questionable impact of using multiple diverse implementations (cf. [7]) is not considered. The application services are further treated as black-boxes with given dependability characteristics. We make the strong simplification that hardware and software component failures simply result in the inability of the affected services to fulfill the regular behavior. Thus, failures can be detected externally by monitoring the services.

A number of architectural principles which permit to enhance the dependability of service-based architectures are presented in Section 2 and their benefits are motivated referring to the Jini architecture. Then, we propose in Section 3 to enhance the Jini architecture by a number of infrastructure services that systematically employ the identified principles. We then discuss the benefits achieved for application specific services concerning availability and reliabil-

[1]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.

ity in Section 4 and demonstrate the systematic application of the identified architectural principles within the enhanced architecture. For a special class of services the possible design alternatives are studied by means of the detailed design of two infrastructure services in Section 5. Related work is discussed in Section 6 and we close the paper with a final conclusion and some comments on future work.

## 2. Architectural Principles for Dependability

Software systems typically consist of different parts. Since dependencies between these parts exist, problems occur if a part fails. Service-based architectures handle the increasing complexity of today's systems by means of online lookup and binding of services. 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. Instead they can spontaneously connect to recover from failures. One example of a self-healing service-based architecture is the Jini architecture [1, 13]. It has been designed (cf. [19]) to support the development of dependable distributed systems. One of its features is a *lookup service* that remains operational even when single nodes in the network have crashed.

The *leasing* principle extends the allocation of resources with time [18]. 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 placeholder for another object [4]. 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) [15]). In the Jini architecture the proxy pattern is an integral 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.

*Redundancy* of service instances is a key factor to achieve a required degree of availability. 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* [9, 11]. A smart proxy is not restricted to forwarding but can be used much more flexible. Thus in the context of availability the proxy may communicate with multiple backends at once to recover from or mask failures. Hence a smart proxy can be used to encapsulate and hide the complexity of self-adapting 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.

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*. Replicating 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 [16]). 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 general solution can be given. The weaker the consistency model the easier 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.

## 3. Architecture

In this section we will show the application of the introduced architectural principles. We give a short introduction of the proposed architecture and the requirements of the different infrastructure services. More details and the description of the implementation can be found in [17].

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 provides availability for application services.

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. Keep this idea in mind while we describe the specific infrastructure services next.

Four different services are building the overall architecture on top of Jini. 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).

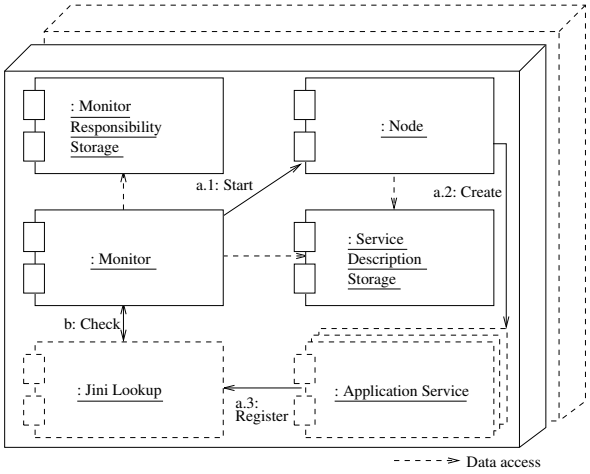


**Figure 1. Architecture**

Basically 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).

A *service description storage* contains service descriptions (like name of the service, package-path, used and provided interfaces, deployment constraints, etc.) for all services which have to be executed. 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.

*Monitors* supervise that the services contained in the service description storage are indeed executed in the system. The availability of the services will be checked periodically. The detection speed of service failures can be configured by changing this period. If more than one instance of one specific application service has to be executed in the system, each instance is monitored by a different monitor. To control which monitor is supervising which service, every monitor needs to acquire a responsibility for a service (i.e. to assure a new instance is started, if a service instance is not available).

These 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). 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 reliability for overhead 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.

## 4. Evaluation

After this short introduction to the different infrastructure services we show how the architecture achieves availability for application services in case of node failures and network partition failures. Afterwards we show how to achieve a required degree of reliability for different categories of application services based on the availability provided by the architecture.

### 4.1. Availability

In case of a node failure different scenarios, w.r.t. failures of a responsible monitor and monitored services, are possible. The case that neither a responsible monitor nor a monitor 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 they 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.

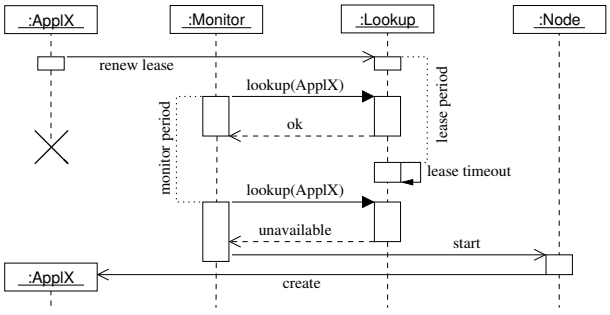


**Figure 2. Node Failure**

In the case of a failed responsible monitor and failed monitored services the responsibility lease of this monitor times out and another monitor steps in. This monitor replaces the failed monitor, and starts supervising the currently unmonitored services which includes starting new in-

stances when needed. Figure 3 shows the leases and the events in a condensed form.

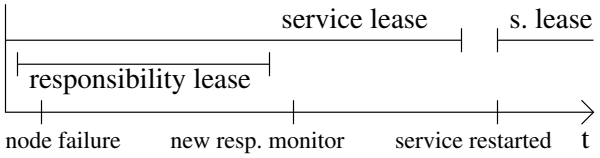


**Figure 3. Monitor Failure**

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 create 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).

After reuniting the different partitions, the responsibility storages are merged to determine a new unified responsible monitor. This new monitor takes over the service instances started by the other responsible monitors in the other partitions. Additionally it can consolidate the number of service instances in the reunited network. The monitors formerly responsible in the other partitions stop monitoring their started service instances.

As seen the availability of application services (and the Mean Time to Repair (MTTR)) can be configured 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 a high degree of availability.

### 4.2. Reliability

The presented architecture ensures the availability of the application services in the system. Nevertheless for each application service there must exist a concept to achieve the required degree of reliability based on the availability provided by the architecture. Different categories of services require different approaches to achieve reliability.

According to [3] services can be categorized in terms of modeling an *entity* or containing an activity (*session*). A session service may either be stateless or stateful. The state of a session is the history of relevant actions executed throughout this session. If an action is independent of the previous actions, the service is stateless, if not it is stateful. An entity service always has a state (its local data).

For stateless session services it is irrelevant which service instance is used for a given action, since the actions are independent of each other. Thus the availability provided by the architecture is sufficient. If a service instance fails, another instance can be used.

If a used stateful session service instance fails, 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.

To achieve reliability for entity services it is necessary to replicate copies of the entity over a number of nodes to be able to mask failures. Additionally the consistency of these entities according to a suitable consistency model must be assured. This replication is highly application-specific and thus no general solution can be given. 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 5).

The implementation of the above mentioned concepts leads to a reliable system, but unfortunately the maintainability of the resulting system deteriorates. We propose the usage of the smart proxy pattern to encapsulate the complexities of achieving reliability. Since the proxy does not fail independently of the using application, the client does not need to handle a failed proxy. Thus a service client only needs to know the interface to the service and nothing about the different means of accomplishing reliability. It uses the smart proxy via an interface and all additional processing for reliability is done internally in the smart proxy (see Figure 4).

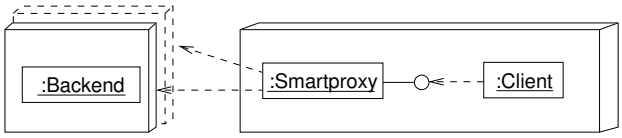


**Figure 4. Smart proxy**

## 5. Design of Infrastructure Services

In the following we further describe in detail the design of two infrastructure services. These services serve as examples how to achieve the required degree of reliability for entity services according to the last section.

**Service Description Storage** This 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 happen 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 unimportant. To exploit this bias for read operations we have chosen to implement the *weighted voting* approach [5] 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. Additionally this number of votes can be changed to match the reliability of that node. 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 ($n$ : number of votes). For our scenario we choose 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 one partition contains more than $n_w$ votes.

The weighted voting approach is implemented in a smart proxy. Thus a client 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.

**Monitor Responsibility Storage** Storing the monitor responsibilities is a problem similar to storing the service descriptions. In contrast 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 write that information back into 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* [16]. 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.

## 6. Related Work

In the Jini-context the problem of availability is somewhat supported by use of the RMI-Daemon [15]. 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 [14] 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 [2] 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. The slaves are the different service instances

whereas the smart proxy is the master in our approach. 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. [8] 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.

Gustavsson and Andler describe in [6] 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.

## 7. Conclusions and Future Work

For the proposed 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. The number of parallel running service instances and lease times for registry and monitoring can be chosen. Thus for any architecture conform application specific service availability can be configured. For different kinds of application services we presented appropriate concepts to also realize a higher reliability. The required additional efforts are systematically hidden to the service clients using the smart proxy concept of Jini. The smart proxy concept itself can be used in every service-based architecture. But the reliability provided by the presented architecture highly depends on the robustness of the underlying service-based architecture. To adapt it to other architectures than Jini, the leasing concept of Jini needs to be reimplemented in the different application services to offer a Jini-like robustness.

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 [17]. 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. We further plan to evaluate the architecture in the context of complex embedded and real-time systems. For small examples a formal analysis using Markov models will be performed. Additionally, we will look how run-time measurements of node, network and component dependability characteristics can be employed to adjust the system parameters such as monitor supervision periods accordingly. In a next step, we want to employ classical approaches for learning and adaption to automatically use this feedback to improve the system's dependability.

## Acknowledgments

The authors wish to thank Sven Burmester, Matthias Gehrke, and Matthias Meyer for comments on earlier versions of the position paper.

## References

[1] K. Arnold, B. Osullivan, R. W. Scheifler, J. Waldo, A. Wollrath, and B. O'Sullivan. *The Jini(TM) Specification*. The Jini(TM) Technology Series. Addison-Wesley, June 1999.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture*. John Wiley and Sons, Inc., 1996.

[3] L. G. DeMichiel, L. Ü. Yalcinalp, and S. Krishnan. *Enterprise JavaBeans$^{TM}$ Specification*. Sun Microsystems, August 2001. Version 2.0.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the seventh symposium on Operating systems principles*, volume 7 of *ACM Symposium on Operating Systems Principles*, pages 150–162. ACM press, 1979.

[6] S. Gustavsson and S. F. Andler. Self-stabilization and eventual consistency in replicated real-time databases. In *Proceedings of the first workshop on Self-healing systems*, pages 105–107. ACM Press, 2002.

[7] J. Knight and N. Leveson. A reply to the criticisms of the Knight & Leveson experiment. *ACM SIGSOFT Software Engineering Notes*, 15(1):25–35, January 1990.

[8] J. C. Knight, D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, P. Devanbu, and M. Gertz. The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications. In *The International Conference on Dependable Systems and Networks (DSN-2002)*, Washington DC, June 2002.

[9] R. Koster and T. Kramp. Structuring QoS-Supporting Services with Smart Proxies. In *Proceedings of Middleware'00*. Springer Verlag, April 2000.

[10] J. C. Laprie, editor. *Dependability: basic concepts and terminology in English, French, German, Italian and Japanese*, volume 5 of *Dependable computing and fault tolerant systems*. Springer Verlag, Wien, 1992.

[11] P. Ledru. Smart proxies for jini services. *ACM SIGPLAN Notices*, 37(4):57–61, Apr. 2002.

[12] P. Oreizy et al. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 2002.

[13] Sun Microsystems. *Jini Specification*, October 2000. Revision 1.1.

[14] Sun Microsystems. *RIO - Architecture Overview*, 2001. 2001/03/15.

[15] Sun Microsystems. *Java$^{TM}$ Remote Method Invocation Specification*, 2002. Revision 1.8, JDK 1.4.

[16] A. Tanenbaum and M. van Steen. *Distributed Systems, Principles and Paradigms*. Prentice Hall, 2002.

[17] M. Tichy. 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, July 2002.

[18] J. Waldo. The jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

[19] J. Waldo, G. Wyant, A. Wollrath, and S. Kendal. A Note on Distributed Computing. Technical report, Sun Microsystems Laboratories, November 1994. TR-94-29.