Availability Simulation of Peer-to-Peer Architectural Styles

Simon Giesecke
Software Engineering Group
Carl von Ossietzky University
26111 Oldenburg, Germany
giesecke@informatik.uni-oldenburg.de

Timo Warns
Software Engineering Group
Carl von Ossietzky University
26111 Oldenburg, Germany
timo.warns@informatik.uni-oldenburg.de

Wilhelm Hasselbring
Software Engineering Group
Carl von Ossietzky University
26111 Oldenburg, Germany
hasselbring@informatik.uni-oldenburg.de

ABSTRACT

This paper addresses the issue of quantitatively investigating availability within peer-to-peer systems. We devise a conceptual framework integrating architectural styles, architectures, and concrete systems. We identify basic characteristics of architectural styles for peer-to-peer systems and give a formal model to describe derived architectures. Architectural descriptions are used as input for simulations to predict the availability of services within real-world systems.

Keywords

dependability evaluation, simulation, architectural patterns, architectural styles, peer-to-peer

1. INTRODUCTION

Dependability is a conceptual framework for various aspects of trust in computing systems [1]. Usually, it is described by its threats, its characteristics, and the means to attain it. In accordance with the terminology used in the ISO standards 14598-1 [8] and 9126-1 [9], we distinguish attributes, which can be measured by metrics, and categories of attributes, called characteristics. Dependability integrates the basic characteristics availability, reliability, safety, confidentiality, integrity, and maintainability. Dependability is threatened by faults, errors, and failures. It can be attained by means of fault prevention, tolerance, removal, and forecasting. We focus on the aspects related to the characteristics of availability and reliability within peer-to-peer systems.

An increasing number of distributed systems is realized using peer-to-peer architectures to avoid bottlenecks when utilizing resources. Peer-to-peer is a general concept for interconnected nodes which share their resources to allow a decentralized distribution of functionality. Each peer may provide services for other peers or request services from other peers. This viewpoint is in line with other definitions of P2P networking [15].

Usually, peers enter and leave the system frequently. The parting of a peer is considered a failure of this peer, because it ceases to be accessible. Failures of single peers affect the availability of services provided by the peer-to-peer system as a whole, because a service jointly fails with the peers hosting it. Furthermore, if a system employs indirect communication, a failing peer or connection may partition the network. In this case, the failure impacts the availability of services, because the services are only accessible within the partition of their particular hosts.

ICSE 2005 Workshop on Architecting Dependable Systems, St. Louis, Missouri, USA, 2005-05-17.

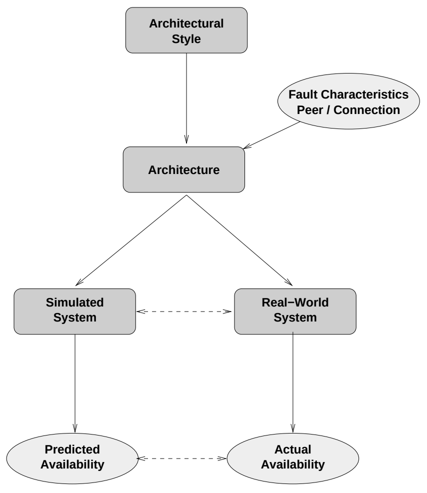


Figure 1: Evaluation Approach

On system level, the failure of a peer or connection is considered a fault. A system needs to employ means of handling such faults to enable continuous working of the services. Recent research proposed various fault handling techniques, e.g. special network topologies, self-organization, periodic description updates, and cross-partition pointers, and analyzed these techniques for specific systems [17].

The availability of services within peer-to-peer systems depends on the faults affecting the participating peers and connections and their particular architectures. A (specific) peer-to-peer architecture is developed in adherence to the constraints established by a (generic) architectural style. In this paper, we devise a conceptual framework for peer-to-peer architectural styles, architectures, and systems to investigate availability more comprehensively.

Figure 1 illustrates our approach to investigate the influence of peer-to-peer architectural styles (section 3) on the availability of services within concrete architectures and systems. Specific architectures are created in dependence on the style and on the fault characteristics of the participating peers and connections. A real-world system corresponds to a specific architecture, whereby the services of the system show actual availabilities. We predict availability by describing the architecture in a formal model (section 4) and performing measurements with the help of a simulation (section 5). The precision of the measurement and, therefore, the appropriateness of the simulation are determined by the degree of accordance of the predicted and the actual availability.

2. RELATED WORK

Walkerdine et al. provided an initial analysis of dependability within peer-to-peer contexts [18]. They discussed the basic relationships between system dependability and logical network architectures, which is one aspect of architectural styles.

Vanthournot et al. proposed several techniques of peer-to-peer systems to handle failures [17]. They simulated a specific peer-to-peer system that employs these means to investigate their influence. Their work can be seen as a special case of our approach. They focused on one architectural style including special peer-to-peer techniques and evaluated the resulting dependability characteristics by simulation.

Bhagwan et al. addressed the question of the semantics of availability within peer-to-peer contexts [3]. They investigated a real-world peer-to-peer system to measure failure characteristics of peers and proposed that the failure distribution should be an overlay of two time-varying distributions. They did not relate their results to the architectural style of the peer-to-peer system under investigation.

3. ARCHITECTURAL STYLES FOR PEER-TO-PEER SYSTEMS

3.1 Architectural Styles

We introduce our general approach to software architecture, explicate the idea of architectural decompositions, and define architectural styles.

Approach to Software Architecture. In this paper, we follow the metaphor of a software architecture as an architecture of a building. However, it is still unclear to what extent this metaphor is helpful to develop the discipline. We assume that every software system has a software architecture that may or may not be explicitly documented [13]. The building metaphor neglects that software is not a physical entity [19], which has a predominant physical structure and interconnections, around which other views are constructed. Ran argued that software is rather an entity of thought that exists in “multiple planes of existence, each having its own type of components—a component domain” [14]. Depending on the specifics of a software project, certain planes may be elevated to a dominant status, but this emphasis is an consciously attribution and not a natural property of the software system. Furthermore, Ran explained that it is a “common mistake of many projects to attempt imposing a single partition in multiple component domains”. On the contrary, any plane may exhibit a different number of components that can have different relationships. Medvidovic and Taylor have argued that the explication of a software architecture inevitably leads to complexity of traceability [13]. However, this complexity is immanent in the software being built [4].

We see the need to distinguish between an architecture and the detailed design of a software system. Several approaches separate both concepts only by their level of detail, which leads to a view of architecture as an incomplete design [14]. In contrast, we see an architecture at a different level of abstraction of content [6].

Jazayeri et al. gave the following definition of software architecture: “Software architecture is a set of concepts and design decisions that enable effective satisfaction of architecturally significant explicit functional and quality requirements and implicit requirements presented by the problem and the solution domains.” [10]

In contrast to other definitions of software architecture, this definition does not explicitly mention structure and relationships. Relationships are not neglected, but can be found as the results of design decisions. Concepts include a model of the problem domain in a form that allows bridging towards a solution domain. Moreover, underlying principles of the created software architecture, such as the used architectural styles, are regarded. This definition distinguishes three kinds of requirements: genuine architectural requirements, problem domain requirements, and solution domain requirements (i.e., the implementation technology).

In many cases, it is difficult to draw a line between the architectural concerns and those of the implementation technology, because both address software in some way. In different projects, the line may be drawn differently. It is therefore necessary to define the point of view of system design.

Architectural decompositions. The concept of imposing multiple decompositions upon a software system architecture is related to the concept of structuring a software architecture into multiple *architectural views*. Kruchten introduced his model of architectural views, whereby each addresses a specific set of concerns [11]. However, there are several important differences: First, Kruchten considers both purely conceptual models and concrete architectural configurations as architectural views. In this sense, architectural views are a broader concept than architectural decompositions. Second, system configurations in different component domains are similar to each other, because all configurations represent a decomposition of the system as a whole. Architectural views are more heterogeneous. For example, they may show state transitions of a single object, which formally are not decompositions of a system. Often, many architectural views refer to one distinguished structural decomposition and add diverse kinds of information. Other system decompositions, like deployment views, are often regarded as subordinate to a main structural view.

Because of the second aspect, architectural views may be regarded as a concept orthogonal to architectural decompositions if an “architectural view” is understood in a restricted sense. While each architectural decomposition imposes a conceptually distinguished structuring of the software system, several architectural views upon each decomposition exist. For example, different architectural views upon an architectural decomposition might focus on performance, reliability, or functionality of the structural elements. Therefore, each view is based on one decomposition, while there is no decomposition predominant for all views on a software system. This model may need to be extended by quasi-views establishing links between multiple architectural decompositions.

To formalize the discussed matter, we first define a set $\mathcal{R}$, the universe of all conceivable software requirements. Let $\equiv$ be an equivalence relation on $\mathcal{R}$, such that each (equivalence class) $C \in \mathcal{C}$ is the set of all requirements related to some characteristic, e.g. the set of all safety requirements. Here, the set $\mathcal{C} := \mathcal{R}/_\equiv$ is the set of all equivalence classes. Hence, each requirement r belongs to one equivalence class $[r]_\equiv \in \mathcal{C}$.

Requirements may be implicitly given by the application or implementation domains or explicitly stated as specific architectural requirements. For any software system S to be realized, a set $R(S) \subset \mathcal{R}$ of architecturally relevant requirements must be decided upon. The set of classes $\mathcal{C}_{R(S)} := \{[r]_\equiv \mid r \in R(S)\}$ includes all architecturally relevant classes. Ideally, the set $\mathcal{C}_{R(S)}$ has the following property:

1. there exists a partitioning $(\mathcal{C}_i)_{i=1}^k$ of $\mathcal{C}_{R(S)}$ such that
2. each partition $\mathcal{C}_i$ is satisfied by an *independent* architectural decomposition.

Formally, part 1 can be stated as:

$$\mathcal{C}_{R(S)} = \bigcup_{i=1}^{k} \mathcal{C}_i \quad \wedge \quad \forall\, i,j \in [1,\ldots,k] : \mathcal{C}_i \cap \mathcal{C}_j = \emptyset \text{ for } i \neq j$$

Regarding part 2, it must be noted that in practice, absolute independence is presumably not reachable, but interdependencies of multiple decompositions should be minimized.

Definition of architectural style. Our definition of architectural style is derived from the definition by Garlan et al. [7]: A fully elaborated architectural style within a component domain consists of

1. a definition of specific types of computational and communicational elements that serve as first-class entities of architectures built according to the architectural style being defined (a vocabulary), and a partial interpretation of these,
2. global constraints on the composition of computational and communicational elements,
3. a description of associated patterns,
4. a guideline on how to apply these patterns when building a configuration,
5. a set of evaluation methods that are enabled by the specifics of the architectural style.

The first two elements of the definition impose hard constraints on the architectures adhering to the style, while the following two elements have a much less strict influence on possible architectures. The last element does not directly influence the possible architectures, but refers to their evaluation, which can have significant impact on the overall system design process.

Architectural styles can be grouped into classes or arranged in a refinement hierarchy, to simplify attribution of evaluation methods or patterns for multiple similar architectural styles.

3.2 Architectural Decompositions for Distributed Computing

In our work, we address distributed computing systems as software-intensive systems, i.e. from a software-centric point of view. It is clear that they may also be viewed from a hardware-centric or network-centric point of view.

The dynamics of architectural decompositions is not explicitly considered by [10, 14]. The evolution of a software architecture needs to be distinguished whether it is driven by decisions during design time or influences during run-time. The former is a question of the representation, documentation, and enactment of design decisions performed by an actor of the design context, e.g. a developer. The latter is performed by an actor within the system itself or its operational context. Architecturally relevant concerns may be affected by structural changes that cannot be anticipated at design time. This run-time evolution is more substantial with regard to an architectural description framework, since architectural decompositions become a function of the running time of the software system. This kind of changeability must be explicitly captured in rules governing run-time structural changes. Run-time changes of the system structure form an architectural concern for peer-to-peer systems, because this dynamics is one of their defining properties.

It is therefore wise to cover the classes $C_1, \ldots, C_n$ of requirements explicitly addressed by distributed systems in one or more dedicated architectural decompositions (see above). Besides the structural dynamics already mentioned, the most important of these requirements are decentralization and shared resource usage. In this paper, we constrain to *one* architectural decomposition covering these requirements as a whole, i.e. we assume that $\exists\, j \in [1,k] : \mathcal{C}_j = \{C_i \mid i \in [1,n]\}$. The decomposition establishes nodes as basic computational elements and communication links as basic communicational elements. These types of elements are further refined by a specific architectural peer-to-peer style, which are discussed in the following.

3.3 Peer-to-Peer Architectural Styles

The characteristics distinguishing different architectural styles can be grouped into (i) the type of decentralization, (ii) type of communication, and (iii) overall structural characteristics.

The *type of decentralization* can essentially be *decentralized*, *hybrid* or *super-peer*. In a decentralized system, all services are decentralized and can be offered by any peer. In a hybrid system, some services, often directory services, are centrally offered by dedicated servers which do not act as regular peers. In contrast to client/server architectures, other services remain decentralized. In a super-peer system, some peers are elevated to a distinguished status, either by configuration or self-organized election. These super-peers help structuring the topology, and offer additional services. The type of centralization impacts the types of available nodes and rules for their run-time evolution.

The *type of communication* may fundamentally be distinguished as direct or indirect. The communication is considered to be direct if peers have direct connections of the underlying network layer among each other. The communication is regarded as indirect if peers communicate with the help of other intermediate peers. The intermediate peers pass messages between the final communication partners. The choice of a communication type determines the type of communication links on the one hand, and influences the rules for adding communication links on the other hand. Communication links may be *mediation links* or *data transfer links*. In the case of an architectural style allowing me-

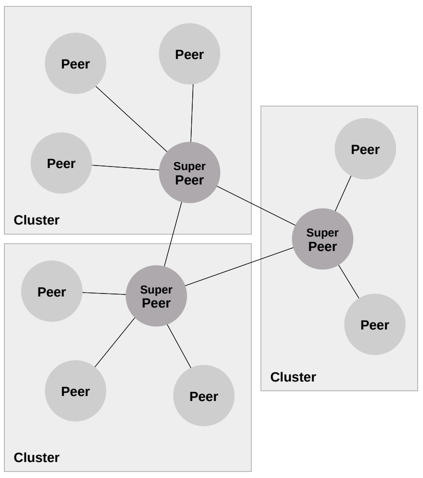


Figure 2: Architecture in a Super-Peer Style

diated direct accesses, data transfer links exist between any two nodes, while mediation links only exist between those nodes forming the network topology imposed by the overall structural characteristics. In the case of strict direct or indirect communication, the sets of nodes connected by mediation links and data transfer links are equal.

Overall structural characteristics impose global constraints on the topology of the graph. A peer-to-peer architecture may put certain restrictions on the shape of the topology, e.g. peers need to organize themselves to a tree, mesh, or ring. A structure can be utilized to reduce communication overhead or algorithmic complexity, e.g. for searching. The costs to maintain a structure can be high, because peers enter and leave the system frequently and need to find their position within the topology.

Example. Based on these characteristics, in principle an unlimited number of architectural styles can be defined. We prepared only some important representatives for simulation. One simple example can be found in figure 2. It shows an architecture in a style employing super-peers, allowing indirect communication between regular peers along the links shown as lines. Topological constraints require a completely connected network of the super-peers and direct connections of peers to a super-peer.

Dynamic Changes of Peer-to-Peer Architectures. The characteristics above are primarily concerned with the static structure of a peer-to-peer architecture. However, a peer-to-peer architectural style also captures rules that govern the run-time evolution of the logical architecture. This evolution is concerned with changes to the underlying physical network topology, i.e. removal or addition of nodes and communication links. The detailed characteristics of such rules are not elaborated in this paper.

4. DESCRIPTION MODEL

4.1 Availability of Peers and Connections

Peer-to-peer systems suffer from different types of faults. Of course, they are threatened by the same faults as every other software system, e.g. software flaws. In this paper, we focus on independent failures of single peers and connections that cause their breakdown. We consider only accidental and non-malicious deliberate faults, i.e. we neglect malicious faults, like attacks. Furthermore, we assume peers and connections to be fail-stop.

The failures of peers and connections are described by their *mean times to failure* (MTTF) and *mean times to repair* (MTTR) as metrics. The distribution of failures is assumed to be exponential. The actual values for the MTTF and MTTR of peers and connections must be estimated or observed at real-world systems. For example, Bhagwan et al. observed a real-world system and gained some insight about the actual failure distributions [3]. The availability of a peer or connection describes its readiness for service and is defined as

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}.$$

The reliability of a peer or connection describes the probability that the peer resp. connection operates free of failures for a specified period of time. It is defined as

$$R(\Delta t) = e^{-\frac{1}{\text{MTTF}}\Delta t}$$

under the assumption of an exponential distribution of failures.

4.2 Peer-to-Peer Architectures

A peer-to-peer architecture is formally defined as a tuple

$$A = (N, C, \nu, \lambda, \tau)$$

consisting of two finite sets N and C of nodes and connections, one node function for connections

$$\nu : C \rightarrow \{\{n_1, n_2\} \mid n_1 \neq n_2, n_1, n_2 \in N\}$$

that maps a connection to its endpoint peers, a node labeling function

$$\lambda : N \rightarrow L$$

that maps a node to its label where L is a fixed set of node labels, and the time mapping

$$\tau : T \rightarrow N_T \times C_T$$

for

$$N_T \subseteq N, C_T \subseteq \{c \in C \mid \nu(c) = \{n_1, n_2\} \Rightarrow n_1, n_2 \in N_T\}$$

that maps an instant to the set of peers and connections that are available in the specified instant.

The time mapping τ describes the evolution of the architecture over time. For example, if a peer p joins the system at time t_n and fails at t_m, p is in the image of the time mapping for $t \in [t_n, t_m[$. A new peer opens and closes several connections to already connected peers to find its place within the topology of the system. If the peer has found its place, it establishes permanent connections to its neighbor peers. All these connections are captured by the time mapping for the periods of time it takes to find the place. Failures of peers and connections are covered likewise: If a peer or connection fails for a period of time, it is not in the image of the time mapping for all instants in this period. Peer-to-peer systems adapt their network topology as a means of fault tolerance. The resulting changes of connections are captured by the time mapping as well.

The actual specification of τ depends on several factors. An initial configuration of peers and connections must be given for $t = 0$. A peer that enters the system is captured by the time mapping for the period of time of its participation. The occurrences and durations of failures of peers and connections are derived from the MTTF, MTTR, and distribution function defined for the peer resp. connection. We currently assume an exponential distribution of the failures and repairs. Peer-to-peer architectural styles define the runtime behavior of peers for opening and closing connections, e.g. to reach a certain position in the topology or to adapt to failures. This behavior is captured by the time mapping, but the way how it is incorporated into its specification has not been explicated yet.

Example. The following example of an evolving architecture illustrates our approach. The architecture is defined as shown in table 1. Figure 3 shows a graphical illustration of that architecture. The architecture consists of six peers and nine connections. For all instants $t \in [t_0, t_1[$ the peers p_1 to p_5 participate in the system. The peer p_5 leaves the system at instant t_1 and, therefore, its connections vanish as well. Peer p_4 had a connection to p_5 and could ask the remaining peer p_3 about other peers. Peer p_3 could tell p_4 about p_1, so that p_4 opens a connection to p_1 at instant t_2. At the same time, the new peer p_6 enters the system with an connection to p_1. Peer p_1 mediates a connection to p_2, which is available after t_3.

Table 1: Specification of Evolving Architecture

$N = \{p_1, \dots, p_6\}$,
$C = \{c_1, \dots, c_9\}$,
$L = \{\text{Peer}\}$,

$\lambda :$

N	L
p_1	Peer
p_2	Peer
p_3	Peer
p_4	Peer
p_5	Peer
p_6	Peer

$\nu :$

C	$\{n_1, n_2\}$
c_1	$\{p_1, p_2\}$
c_2	$\{p_2, p_3\}$
c_3	$\{p_1, p_3\}$
c_4	$\{p_3, p_4\}$
c_5	$\{p_3, p_5\}$
c_6	$\{p_4, p_5\}$
c_7	$\{p_1, p_6\}$
c_8	$\{p_1, p_4\}$
c_9	$\{p_2, p_6\}$

$\tau :$

T	N_T	C_T
$t \in [t_0, t_1[$	$p_1, \dots, p_5$	$c_1, \dots, c_6$
$t \in [t_1, t_2[$	$p_1, \dots, p_4$	$c_1, \dots, c_4$
$t \in [t_2, t_3[$	$p_1, \dots, p_4, p_6$	$c_1, \dots, c_4, c_7, c_8$
$t \in [t_3, t_4[$	$p_1, \dots, p_4, p_6$	$c_1, \dots, c_4, c_7, c_8, c_9$

5. AVAILABILITY SIMULATION

A peer-to-peer system establishes a logical network topology wherein most pairs of peers are connected via more than one path. Various methods to analyze availability and reliability are known for such networks. Peer-to-peer systems are complex systems. The assumption that "most real-world systems are too complex to allow realistic models to be evaluated analytically" [12] applies to peer-to-peer systems. Therefore, we employ simulation as an approximate method for the evaluation of availability.

Currently, we develop a simulator for peer-to-peer systems, which is based on a *discrete-event simulation model*. Hence, the state of the system as a whole changes instantaneously at discrete points in time, i.e. faults, repairs, sending, and handling of messages are instantaneous occurrences which change the system state. The simulator accepts a formally defined peer-to-peer architecture and implements a model of peer behavior for handling messages. It simulates accesses to services and logs the outcome of these requests. The results are used to calculate the resulting availability of services within the simulated peer-to-peer system.

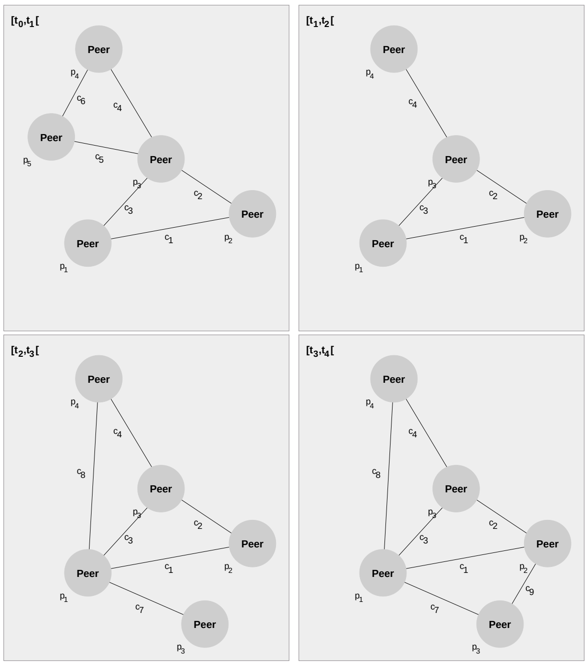
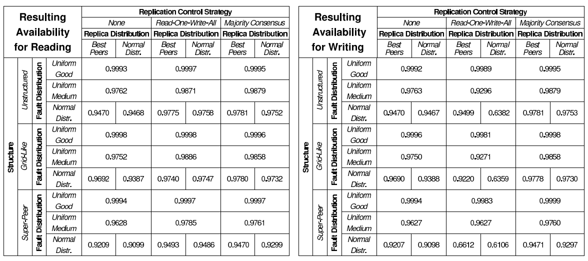


Figure 3: Illustration of Evolving Architecture



Resulting Availability for Reading				None – Replica Distribution – Best Peers	None – Replica Distribution – Normal Distr.	Read-One-Write-All – Replica Distribution – Best Peers	Read-One-Write-All – Replica Distribution – Normal Distr.	Majority Consensus – Replica Distribution – Best Peers	Majority Consensus – Replica Distribution – Normal Distr.
Structure	Unstructured	Fault Distribution	Uniform Good	0.9993		0.9997		0.9995	
			Uniform Medium	0.9762		0.9871		0.9879	
			Normal Distr.	0.9470	0.9468	0.9775	0.9758	0.9781	0.9752
	Grid-Like	Fault Distribution	Uniform Good	0.9998		0.9998		0.9996	
			Uniform Medium	0.9752		0.9886		0.9858	
			Normal Distr.	0.9692	0.9387	0.9740	0.9747	0.9780	0.9732
	Super-Peer	Fault Distribution	Uniform Good	0.9994		0.9997		0.9997	
			Uniform Medium	0.9628		0.9785		0.9761	
			Normal Distr.	0.9209	0.9099	0.9493	0.9486	0.9470	0.9299

Resulting Availability for Writing				None – Replica Distribution – Best Peers	None – Replica Distribution – Normal Distr.	Read-One-Write-All – Replica Distribution – Best Peers	Read-One-Write-All – Replica Distribution – Normal Distr.	Majority Consensus – Replica Distribution – Best Peers	Majority Consensus – Replica Distribution – Normal Distr.
Structure	Unstructured	Fault Distribution	Uniform Good	0.9992		0.9989		0.9995	
			Uniform Medium	0.9763		0.9296		0.9879	
			Normal Distr.	0.9470	0.9467	0.9499	0.6382	0.9781	0.9753
	Grid-Like	Fault Distribution	Uniform Good	0.9996		0.9981		0.9998	
			Uniform Medium	0.9750		0.9271		0.9858	
			Normal Distr.	0.9690	0.9388	0.9220	0.6359	0.9778	0.9730
	Super-Peer	Fault Distribution	Uniform Good	0.9994		0.9983		0.9999	
			Uniform Medium	0.9627		0.9627		0.9760	
			Normal Distr.	0.9207	0.9098	0.6612	0.6106	0.9471	0.9297

Figure 4: Evaluated Availability of Access to Replicated Resources

Our simulation model is based on the behavior of real-world systems such as Freenet [5]. Essentially, it describes how messages are passed in a simulated system, i.e. how a peer communicates with other peers. Aspects related to topology are already captured by the specification of the architecture. The simulation model can be augmented by other peer-specific functionality, like indexing.

We used an early prototype of the simulator to evaluate the influence of the two basic voting-based replication strategies *Read-One-Write-All* (ROWA) and *Majority Consensus* [2,16] on availability of resources within peer-to-peer systems. Figure 4 shows preliminary results for these strategies and different architectures. Figure 5 shows the relative deviation of their impact compared to non-replicated resources. The early results already indicate that the strate-

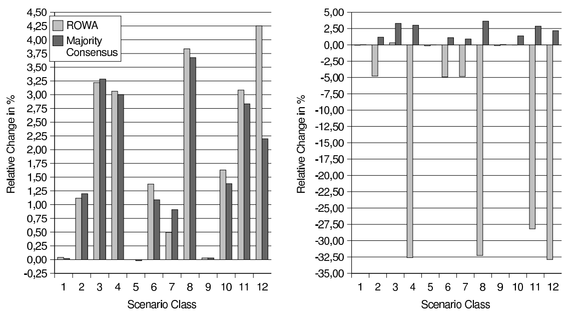


Figure 5: Relative Deviation to Non-replicated Resources

gies perform better than expected within specific peer-to-peer systems.

6. FUTURE WORK

In future work, the characteristics establishing different architectural styles introduced in section 3.3 will be further refined and formalized to a greater extent. Furthermore, we will explicate the relationships between an architectural style and a corresponding architecture. Especially, means of topology, e.g. techniques of fault tolerance, need to be formalized, e.g. by graph grammars, to ease the specification of the time mapping.

7. REFERENCES

[1] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[2] Philip A. Bernstein and Nathan Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst.*, 9(4):596–615, 1984.

[3] Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Understanding availability. In *Peer-to-Peer Systems II, Second Intl. Workshop*, volume 2735 of *Lecture Notes in Computer Science*, pages 256–267. Springer, 2003.

[4] Frederick P. Brooks, Jr. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[5] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In Hannes Federrath, editor, *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 / 2001 of *Lecture Notes in Computer Science*, pages 46–66. Springer, 2000.

[6] Timothy Colburn. Methodology of computer science. In Luciano Floridi, editor, *The Blackwell Guide to the Philosophy of Computing and Information*, chapter 24. Blackwell, Oxford, 2004.

[7] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proc. SIGSOFT '94*, pages 175–188. ACM Press, 1994.

[8] ISO/IEC. *ISO/IEC 14598-1: Information technology – Software product evaluation – Part 1: General overview*, 1999. Published standard.

[9] ISO/IEC. *ISO/IEC 9126-1: Software Engineering – Product Quality – Part 1: Quality Model*, June 2001. Published standard.

[10] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software architecture for product families: principles and practice.* Addison-Wesley, Boston, USA, 2000.

[11] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[12] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, Boston, MA, USA, 3rd edition, 2000.

[13] Nenad Medvidovic and Richard N. Taylor. Separating fact from fiction in software architecture. In *Proc. 3rd Intl. Software Architecture Workshop (ISAW3)*, pages 105–108. ACM Press, 1998.

[14] Alexander Ran. Software isn't built from lego blocks. In *Proc. 1999 symposium on Software reusability*, pages 164–169. ACM Press, 1999.

[15] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *First Intl. Conf. on Peer-to-Peer Computing (P2P 2001)*, pages 101–102. IEEE Comp. Soc. Pr., August 2001.

[16] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[17] Koen Vanthournot and Geert Deconinck. Building dependable peer-to-peer systems. In *DSN 2004 Workshop on Architecting Dependable Systems*, 2004.

[18] James Walkerdine, Lee Melville, and Ian Sommerville. Dependability properties of p2p architectures. In *Second Intl. Conf. on Peer-to-Peer Computing (P2P 2002)*, pages 173–174. IEEE Comp. Soc. Pr., 2002.

[19] Peter Wegner. Research paradigms in computer science. In *Proc. ICSE '76*, pages 322–330. IEEE Comp. Soc. Pr., 1976.