# Building Reliable Systems based on Self-Organizing Multi-Agent Systems*

Florian Klein† and Matthias Tichy
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[fklein|mtt]@upb.de

## ABSTRACT

High reliability and availability are of utmost importance to the majority of today's software-based systems. Typically, some kind of redundancy is used in order to achieve high reliability and availability in case of faults. Dynamic redundancy setups employ backup systems which take over as soon as the primary system fails. Additionally, central fault tolerance management systems are often used to guide the replication process. In this paper, we present an approach which relies on emergent fault-tolerance. The desired behavior is the result of local reconfigurations of self-organizing agents. This allows us to abandon the classical distinction of primary and backup systems as well as the need for a central management system. Our experimental results show the benefit of our approach.

## Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; I.2.11 [**Computing Methodologies**]: Distributed Artificial Intelligence—*Multiagent systems*

## General Terms

Design, Reliability

## Keywords

Reliability, Availability, Agent, Multi-Agent Systems, Dynamic Task Allocation

†supported by the International Graduate School of Dynamic Intelligent Systems.

*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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SELMAS'06*, May 22–23, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.

## 1. INTRODUCTION

Modern society is increasingly dependent on information systems. Consequently, the reliability and availability of these systems are of utmost importance in the majority of the cases. As the recent improvements in performance and functionality are largely due to software, these systems are usually complex and software-intensive. In order to make them reliable and available, these requirements need to be considered during the software development and reflected in the design of the software itself.

Different techniques are used to develop reliable and available systems (cf. [11]). Typically, some kind of redundancy is used in order to achieve high reliability and availability. If hardware or software fail by crashing, a backup system takes over. The impact of implementation faults is reduced by using different software implementations in parallel. Many approaches employ special fault tolerance management components to ensure continuous operation in spite of faults (e.g. [12, 9]). Other approaches refrain from using a central fault tolerance management facility and include fault tolerance capabilities in the components (e.g. [6]). Both types of approaches depend on communication between the software components to coordinate the fault treatment. All of these approaches increase development and production costs or reduce performance at runtime.

We propose an approach which does not require special fault tolerance management components, special fault tolerance algorithms nor direct communication between the software components. Instead, each software component is represented by a self-interested agent. This agent participates in a decentralized market-based coordination mechanism whose rules make the desired fault-tolerant behavior the economically rational choice.

As a motivating example, we use a streaming audio processing system that applies filters to audio packets. The specific sequence of filters that is applied to each individual stream is set by the user. Filter operations are performed in software by digital signal processing (DSP) agents which can perform any type of filter operation, albeit not concurrently. Each DSP agent decides autonomously which type of filter operation it should perform. Reconfiguring the DSP agent from one operation to another is associated with a set-up time. The system is clocked, i.e., audio packets arrive at fixed intervals and are processed by the filters within a single time slot. As the audio processing has to be performed in real-time, a packet is lost if a required filter is not available. The DSP agents and the executing hardware are subject to crash faults. The goal of our approach is to minimize the resulting number of lost packets, i.e. maximize the availability (percentage of successfully processed packets) and reliability (probability of a sequence

of successfully processed packets). The reconfiguration (switching filter operations) in response to other agents' failures is instrumental in providing fault-tolerant system behavior.

In our approach, each self-interested agent tries to maximize its monetary reward. The desired fault-tolerant behavior is induced by the social rules of the system that provide the appropriate incentives. Agents receive a monetary reward for providing a filter operation. Reconfiguring the filter operation is associated with monetary costs. The behavior of the agents is consequently heavily influenced by the specific reward and reconfiguration cost functions. We therefore chose reward and reconfiguration cost functions which steer the agents towards fault-tolerant system behavior. The actual behavior of an agent is determined autonomously by its strategy. The strategy is free to decide at which time the agent should be reconfigured to provide a different filter operation. As the strategy is designed to be economically rational, however, and bases its decision on the expected rewards and penalties, the right incentives will predictably lead to globally beneficial behavior.

As the agents coordinate their reconfiguration behavior indirectly through the environment, there is no centralized control structure or other single point of failure. In other words, the fault-tolerant behavior is *emergent*, i.e., not inherent in the isolated agents (see [13]). By using the ability of the system to reconfigure flexibly to respond to current demand, we can also drop the distinction between primary and backup systems.

This scenario can be used to also represent other real-world examples, e.g., packets being transported through a packet-switched network, a distributed computing grid performing scientific computations, the internal processing of a reconfigurable microchip, or, from a totally different domain, work items passing through the stations of a flexible production system. Consequently, we use a generalization of the scenario in the remainder of this paper.

We present the abstracted scenario and a formal definition of the problem in the next section. In Section 3, we introduce a classic centralized solution that we use to benchmark our agent-based solution. We provide a detailed introduction to our approach in Section 4, presenting the incentive functions and agent strategies we employed. We then present an evaluation of the different strategies in Section 5. A discussion of related work is presented in Section 6. Finally, we conclude in Section 7 and provide an outlook on future work.

# 2. PROBLEM DOMAIN

## 2.1 Scenario

The abstract scenario consists of tasks (audio packets) passing through a system that performs sequences of transformations (filter operations) on them. The internal architecture of the system is service-oriented, i.e. the transformations correspond to invocations of stateless services. An agent is capable of providing every kind of service. However, an agent may only provide one kind of service and process only one request at the same time.

Each task requires an individual sequence of services to be performed during a specified time slot. When entering the system, the task therefore attempts to contract agents capable of providing such services at the required time slot. It does so by placing specific requests with the individual services, which it must do a configurable but fixed number of time slots before the desired execution time. If the task has managed to reserve all required services for the specified time slot, it passes through the system; otherwise the task fails and is dropped. We assume that the time required for performing the actual services is small compared to the size of a time slot. Thus, tasks pass through the system in a single time slot, provided that the services that they use on their path through the system have been properly reserved. Agent reconfiguration, i.e. switching from providing one service to providing another, is time-consuming and takes up several time slots, which makes on-demand / just-in-time reconfiguration in reaction to individual tasks difficult and inefficient.

The number of agents is limited by the available hardware; each node is capable of running a fixed number of agents. None of these components are 100 percent reliable (we are only considering crash failures in this paper): Individual agents may crash and are automatically restarted some time slots later. Hardware nodes may fail by crashing, taking down the agents running on them. Fixing them is a significantly slower manual process. We assume that the failures happen only after a service execution by an agent. We do not take other failures (e.g. value failures) into account.

According to [2], availability is the probability that an item will perform its required function under given conditions *at a stated instant of time*. In this scenario, the system is available if for any given combination of tasks that need to be processed during the same time slot, enough instances of the required services exists, which in turn necessitates a sufficient number of agents dedicated to the right services.

Reliability is known as the probability that an item will perform its required function under given conditions *for a stated time interval* [2]. In this scenario, the system is reliable if for any given combination of tasks that need to be processed during a number of time slots, enough agents are dedicated to the required services.

In the following, we formally define the problem domain. Based on these definitions, we further on present two of the employed agent strategies.

## 2.2 Formalization

***Agents*** The set of agents $A$ contains $n_A$ agents. Agents fail by crashing with an exponential distribution with $\lambda_{fa}$. The repair time of crashed agents follows an exponential distribution with $\lambda_{ra}$.

***Services*** The set of services is $S$, containing $n_S$ services. For each $s \in S$, there is a (possibly empty) set of agents $A_{(s)}$ performing the service. The number of agents in $A_{(s)}$ is $a_{(s)}$. The time to reconfigure an agent between two services is $\Delta t_R$.

***Nodes*** The set of hardware nodes is $H$, containing $n_H$ nodes. Each node $h \in H$ hosts $n_{(h)}$ agents and can host up to $nmax_{(h)}$ agents. The number of agents on node $h$ providing service $s$ is $n_{(h,s)}$. Hardware nodes crash fail with an exponential distribution with $\lambda_{fh}$. Repair times are exponentially distributed with $\lambda_{rh}$.

***Tasks*** The set of tasks is $W$, the number of currently scheduled tasks is $n_W$, the number of tasks scheduled for a time slot $t$ is $n_W(t)$. Every task $w_{(w)}$ consists of an execution time slot $t_{(w)}$ and a sequence of services $\{s^{(i)}_{(w)}\} : \{s^1_{(w)} ... s^m_{(w)}\}$ it requires. The interarrival times are exponentially distributed with $\lambda_w \geq 1$. A task's execution time is $\Delta t_W$ time units after its arrival time. Service usage patterns are stochastically modeled by an open Markov model with transition matrix $M$.

***Tokens*** The request pools $RQ_{(s)}$ of service $s$ contain request tokens $tok_{(w)}$. The number of request for a service $s$ and a time slot $t$ is then $rq_{(s)}(t) := \#(tok_{(w)}) : (t_{(w)} = t)$. Each service keeps a history of depth $t_{sqh}$ time slots. $\bar{rq}_{(s)}(t)$ is the moving average of the number of requests per time slot, $rq_{max(s)}(t)$ the maximum.

# 3. CENTRALIZED SOLUTION

In order to have a point of reference for benchmarking our decentralized approach and showing its competitiveness, we have de-

veloped a strategy where a central controller makes reconfiguration decisions for all components of the system. As a centralized controller can, ideally, enforce a globally optimal behavior, the results obtained for the centralized solution represent an approximate upper bound, not something we could hope to improve upon – especially since game theory has established that selfish agents will, in general, not reach a global optimum, but fall short by a certain margin, the 'price of anarchy'. Thus, while this margin obviously needs to keep within acceptable bounds, the strengths of an agent-based solution lie elsewhere. For large systems, a decentralized solution scales better than a single controller. Moreover, the complexity of the individual components is lower, as fault tolerance is not explicitly engineered into some central instance, but emerging from the interaction of a multitude of simple components, thus reducing the probability of design or implementation errors.

We have implemented the centralized solution in two ways: In the robust implementation, the central reconfiguration controller is a fault-tolerant agent that never fails. This solution thus represents an idealized best-case scenario. In the basic implementation, we denote one particular agent, which is subject to failure like any other agent, as the controller. This solution explores the adverse effects that a single point of failure without proper fault tolerance mechanisms may realistically have on a system.

In accordance with our goals to maximize availability and reliability, the centralized solution is designed to actively minimize the number of task failures that occur. In order to do so, the algorithm basically compares the number of registered requests for the upcoming time slots with the number of active providers for each required service and reconfigures the system if there is a shortage or potential shortage at any one service.

In a first pass, the algorithm compares the number of requests with the number of available agents for each service and thus computes the number of expected task failures for each of the next planning limit ($\leq \Delta t_W$) time slots, i.e., the near future for which tasks have already placed request tokens. For each service, it then computes $\Delta f_{-1}(s)$, the estimated number of additional failures that would occur if one agent reconfigured and stopped providing the service immediately, and $\Delta f_{+1}(s)$, the estimated number of failures that would be avoided if an agent was dispatched to the service immediately. The latter computation needs to take into account that the agent will only arrive after the reconfiguration time has elapsed so that a positive effect will only register after $\Delta t_R$ time slots. Together, these values allow estimating the net effect of a reconfiguration in a specific situation. The algorithm will then perform all those reconfigurations that yield a positive net effect, i.e. reduce the total number of failures. When all $\Delta f_{-1} > 0$, this adds failures at one service in order to avoid a greater number of failures at another. In all, this mechanism is mostly at work when agents are comparatively scarce.

If the first pass has succeeded in eliminating all previsible failures, a second pass tries to distribute a potential available surplus of agents more evenly. This is rational as an agent failure at a service with just the minimally sufficient number of agents will directly lead to a task failure, in case it is less than $\Delta t_R$ time slots in the future even a failure which cannot be remedied by a timely reconfiguration. The algorithm thus computes the overhead, the minimal number of surplus agents a service has at any one time slot, and moves agents around until the largest and the smallest overhead lie within an interval of size safety margin. By choosing a safety margin $> 1$, we prevent the undesirable effect that the strategy engages too many agents in gratuitous precautionary reconfigurations, leaving it with nothing to work with in case of an unexpected real emergency, i.e. a series of node failures.

Newly restarted, unoccupied agents are assigned to services during both passes. They are always picked for reconfiguration before all other agents as reconfiguring them never entails any negative effects.

Note that this algorithm is merely a heuristic, though one which behaves rationally and predictably in all situations. More sophisticated algorithms using statistical prediction methods could improve upon its performance. However, finding the globally optimal reconfiguration strategy *a priori* is actually impossible, as this would require advance knowledge of future failures.

# 4. SELF-ORGANIZING SOLUTION

Our decentralized approach is based on the underlying principle of self-organization of autonomous agents. The agents are self-interested and follow their own agenda, based on locally available information.

The operation of the system is governed by a set of simple social rules that impose certain requirements on the individual agents and make specific behavioral guarantees (cf. [4]). As the agents do not communicate directly, but only interact through the environment, these rules focus on the observable behavior of the agents in the environment (cf. [5]).

To coordinate task allocation in the system, there is a decentralized request pool for each service type that is accessible to all agents and tasks. Upon entering the system, a task places tokens marked with the desired time slot into the request pools of the required services. Agents can now commit to process a task by acquiring such a token if they currently provide the desired service and possess no previous commitments for the time slot in question. At the beginning of each time slot, a social rule requires the active agents that currently provide a specific service to acquire any unassigned tokens for that time slot, unless they are previously engaged, i.e. agents are forced to actually perform the service they claim to provide. Agents are also required to periodically confirm their commitments by asserting their claim on their tokens in order to ensure they have not reconfigured or crashed.

When the specified time slot arrives, agents fulfill their commitments by in turn removing their token from the pool and, if it arrives, processing the corresponding task. Unacquired tokens are simply dropped from the request pools, the corresponding tasks fail. The request pools store a limited history of depth $t_{sqh}$ containing elementary statistics (minimum, maximum and moving average) concerning the number of requests, active providers, and task failures.

In order to elicit the desired behavior, the social specification provides two incentive mechanisms that reward desirable and punish detrimental behavior.

Firstly, agents receive a reward for providing (not performing) a service. Even though the agents thus do not actively compete for tokens, the forces of supply and demand are still in effect, as the reward is computed as a function of the number of requests and providers. For a future version, we consider replacing this by auctions where agents actually bid for tokens and have to pay a penalty for dropping them, which would additionally introduce a way to prioritize tasks.

Secondly, reconfigurations carry a cost that can be used for steering the system's evolution and controlling the stability of configurations. Agents that decide to reconfigure to a more lucrative service, i.e. a less populated or more frequently requested one, need to pay a fee. As this fee pays in part for the right to quit the source service and in part for the right to provide the target service, its net effect depends on the relative valuation of the two services. Additionally, as switching to a different service disables the agent for the time

spent reconfiguring, the agent also forgoes the reward payments for the affected time slots.

How they operate within these constraints in order to maximize their profit is, however, up to the agents. The shared set of social rules only constrains their behavior w.r.t. acquiring and fulfilling commitments – internally, the agents are free to use different strategies and heuristics for determining their course of action.

## 4.1 Reward function

We quickly identified the agent-to-request ratio $atr_{(s)}(t) = a_{(s)}(t)/rq_{(s)}(t)$ as the most indicative parameter to use as the exclusive independent variable of the reward function. As it has a strong influence on the long term behavior of the agents, we experimented with different linear and non-linear functions. Ultimately, we settled on a function based on the cumulative density function of the Laplace distribution (see Function 1) which yields a monotonically decreasing S-shaped curve (see Figure 1).

$$r := weight_r\Big(r_{min} + (r_{max} - r_{min}) \operatorname{sgn}(atr_r^* - atr_{(s)})\big(e^{-\frac{|atr_r^* - atr_{(s)}|}{b_r}} - 1\big)\Big) \quad (1)$$

The function's shape is particularly suited to its purpose. When agents are scarce, they receive a high reward of nearly $r_{max}$ times $weight_r$, the standard reward. Up to a certain $atr_{(s)}$, adding agents is beneficial. The reward therefore decreases only slightly up to that point. At the target ratio $atr_r^*$, the reward is exactly $weight_r$, the standard reward. Above that $atr_{(s)}$, the load gets so low that more and more agents are constantly idle; the reward thus decreases sharply and approaches $r_{min}$ times the standard reward. The sensitivity $b_r$ controls the steepness of the decrease.

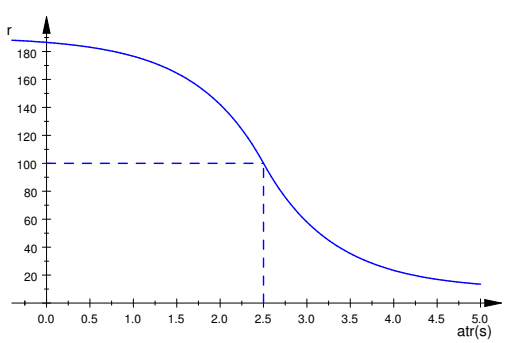


**Figure 1: Plot of the reward function for parameters** $weight_r = 100, r_{max} = 2, r_{min} = 0.1, atr_r^* = 2.5, b_r = 0.8$

The two most significant parameters are $weight_r$, which scales the reward function w.r.t. to the reconfiguration cost function, and $atr_r^*$, which moves the inflection point and thus the sharp decrease (both are shown as dashed lines in Figure 1). The ideal value of $atr_r^*$ depends on the failure rates and task arrival patterns; however, it should roughly reflect the relation between the largest and the average number of concurrent requests $\bar{rq}_{(s)}$ and $rq_{max(s)}$ so that, in the average case, the number of agents that is necessary for handling peak demands is still considered beneficial.

## 4.2 Reconfiguration Cost

The function for computing the net cost for reconfiguring from a source service to a target service consists of three components, a fixed basic penalty $p_{fixed}$ and two variable terms, depending on the agent-to-task ratios $atr_{(s)}$ and the services' degree of dependence on individual nodes, respectively.

$$p := p_{fixed} + p_{atr} + p_{nh} \quad (2)$$

An agent-to-request ratio below 1.0 corresponds to failures and is thus to be avoided. For evaluating reconfigurations, we use the recorded average number of requests rather than the current value to obtain a more reliable estimate: $atr_{(s)}(t) = a_{(s)}(t)/\bar{rq}_{(s)}(t)$.

The function $p_{atr}$ (see Function 3) is designed to make moving from providing a crowded service to a sparsely populated service very attractive and the reverse very unattractive. To this purpose, each service is scored using an inverse exponential function that grows rather quickly below the target ratio $atr_p^*$, where its value is $p_{norm}$, and falls down to $p_{min}$ for higher ratios. The sensitivity $b_r$ controls the strength of the effect. $p_{min}$ should be negative, $atr_p^*$ around 2.5.

$$p_{atr} := weight_{\Delta atr} \cdot (p_{min} - p_{norm}) \left(e^{-b_p \cdot (atr_{(src)} - atr_p^*)} - e^{-b_p \cdot (atr_{(tgt)} - atr_p^*)}\right) \quad (3)$$

The net reconfiguration cost then is the difference between the valuations for source and target, weighted with the factor $weight_{\Delta atr}$. Figure 2 illustrates the relationship for different source and target ratios.

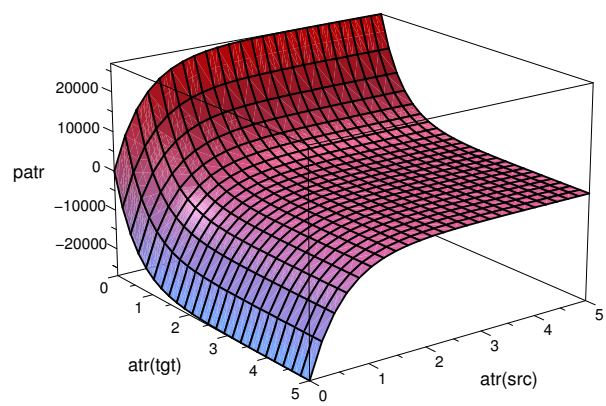


**Figure 2: Plot of the reconfiguration cost function** $p_{atr}$ **for parameters** $weight_{\Delta atr} = 100, p_{norm} = 1.0, p_{min} = -2.0, atr_p^* = 2.5, b_p = 1.8$

The term $p_{nh}$ is concerned with the *node heterogeneity* of services, i.e. the distribution of the agents providing a service among the nodes. We aim to avoid situations where too many agents providing the same service run on a single node, making that node nearly a single-point of failure for the service provision. Instead, we try to ensure that the agents providing a certain service are evenly distributed over several nodes. We therefore try to maximize the node heterogeneity of a service, which we define as the percentage of service providers that would at least still be active after any one node fails. $p_{nh}$ then corresponds to the net effect of a reconfiguration on node heterogeneity, weighted with $weight_{nh}$.

Consider the situation in Figure 3: two different agents from node 1 are providing service A. This causes a node heterogeneity of $pre(A) = 1 - max(n_{(h,A)})/a_{(A)} = 1 - 2/4 = 0.5$ for the service A before ($pre$) the reconfiguration. The node heterogeneity

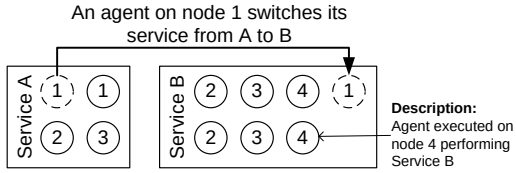


**Figure 3: Positive Node Heterogeneity Change**

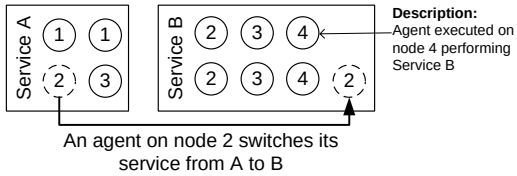


**Figure 4: Negative Node Heterogeneity Change**

for service B is $pre(B) : 1 - max(n_{h,B})/a_{(B)} = 1 - 2/6 = 0.66$. The reconfiguration of one agent which is executed on node 1 from providing service A to service B leads to node heterogeneities of $post(A) : 1 - 1/3 = 0.66$ and $post(B) : 1 - 2/7 = 0.71$ after ($post$) the reconfiguration, which corresponds to an overall improvement of node heterogeneity:

$$\delta = post(A) - pre(A) + post(B) - pre(B) = 0.17 + 0.04 = 0.21$$

Figure 4 shows how the reconfiguration of an agent on node 2 to service B has a negative effect of $-0.27$ on the overall node heterogeneity.

## 4.3 Control Strategies

Based on these functions, we then implemented and tested different agent control strategies. Strategies that based their decisions on the recorded average reward or a combination of current and historic data proved only moderately successful. Making decisions based on the distribution of available tokens in the next few time slots turned out to be the dominant strategy for the agents.

Basically, the *preview* strategy compares the estimated reward for performing the current service for the next $\Delta t_W$ time slots with the estimated reward for performing one of the other services. As reconfiguration deactivates the agent for $\Delta t_R$ time slots, the reward at the target service needs to be significantly higher to be competitive. The reconfiguration cost function usually further reduces the total reward, but may actually pay an agent to reconfigure if providers of a particular service are scarce.

If the best alternative beats the current service by more than the parameter threshold, the agent performs a reconfiguration. The threshold was introduced to avoid destabilizing cycles of gratuitous reconfigurations for minimal gains, which may occur when the $atr$ of all services is at a similarly high level.

# 5. EVALUATION

We evaluated our approach using a simulation environment implementing the scenario presented in Section 2. [1] The exact reward and reconfiguration cost functions as well as the different strategies were tuned by numerous simulation experiments.

In the following, we present selected simulation results for the different solutions and compare the availability and reliability achieved for different failure rates and deployment structures. The figures below represent 1584 simulation runs. In each run, 50,000 simulation events are processed, which is equivalent to approximately 3000 time slots. The markov model in Figure 5 was used to randomize which services were used by the generated tasks. We used the cost functions presented above, a reconfiguration time $\Delta t_R = 5$ and a delay $\Delta t_W = 10$ for tasks. We set a task rate $\lambda_w$ of 10 tasks per time slot, an agent repair rate $\lambda_{ra}$ of 1.0, an agent failure rate $\lambda_{fa}$ of 0.001, and a node repair rate of $\lambda_{rh} = 0.1$. We varied the node failure rate $\lambda_{fh} = [\,0.0000; 0.0225]$. The decentralized preview strategy and the robust and non-robust centralized solution were then applied to each resulting scenario, repeating each experiment 9 times using different random seeds.

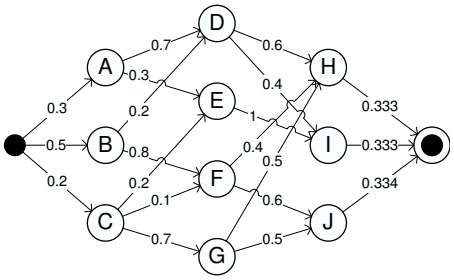


**Figure 5: Markov model of the task / service distribution**

Both the robust centralized and the decentralized strategy reliably produce perfect availability for the above scenario if provided with a sufficient number of agents ($80 - 250$, depending on $\lambda_{fh}$). Below, we therefore focus on their ability to keep a system operational with a minimum of agents.

## 5.1 Small Number of Agents per Node

In the first experiment, we used small nodes running 4 agents. Figures 6 and 7 show the results for 20 nodes and 30 nodes, respectively.

The preview strategy and the robust centralized solution are clearly superior to the non-robust centralized solution in most cases. For low failure rates, the agent-based strategy performs

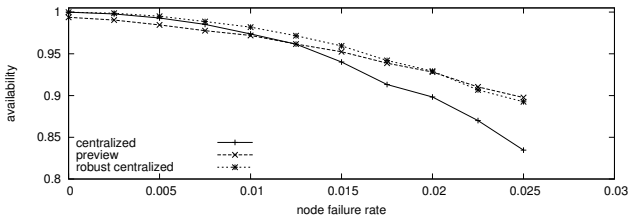


**Figure 6: Availability with 20 nodes, for different failure rates**

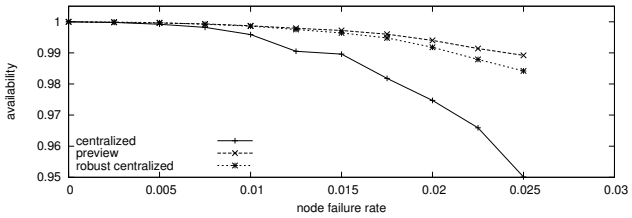


**Figure 7: Availability with 30 nodes, for different failure rates**

[1] The source code of the simulation environment is available from `http://www.uni-paderborn.de/cs/tichy/reliable_agent_simulation/`.

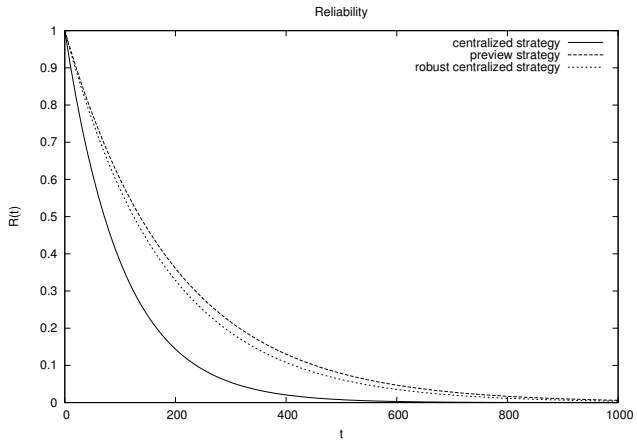


**Figure 8: Reliability functions for 30 nodes with a failure rates of 0.007**

slightly below the centralized solutions, but actually surpasses our explicit algorithm for high failure rates.

Figure 8 approximates the reliability based on the empirical failure rate $\hat{\lambda}$ of the complete system for a node failure rate of $0,0075$ and $30$ nodes. The plot shows the reliability function $e^{-\hat{\lambda}t}$. Again, the preview strategy is on par with the robust solution.

## 5.2 High Number of Agents per Node

In the second experiment, we employ a small number of large nodes. Figures 9 and 10 show the results for $4$ nodes with $20$ agents and $30$ agents per node, respectively.

In this experiment, the preview strategy excels for all but very low failure rates. For high $\lambda_{fh}$, the deterioration of the non-robust centralized strategy is even more pronounced than above, especially in the scenario with $30$ agents per node.

## 5.3 Lessons Learned

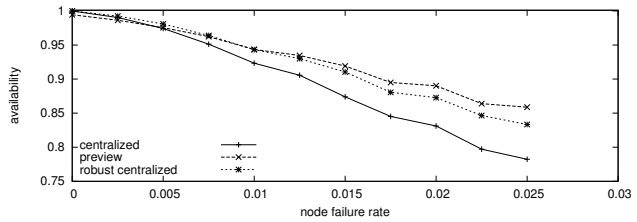


**Figure 9: Availability with 20 agents per node, for different failure rates**

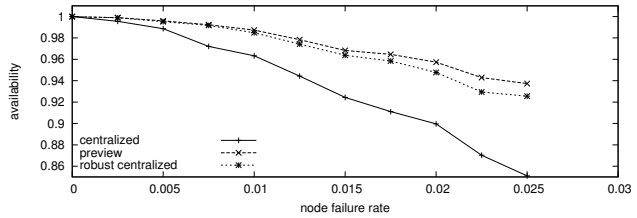


**Figure 10: Availability with 30 agents per node, for different failure rates**

The decentralized *preview* strategy is competitive w.r.t. the robust centralized solution and outperforms the non-robust centralized strategy in most scenarios. The fact that it outperforms both centralized solutions for high failure rates is, admittedly, primarily indicative of a shortcoming of our algorithm, namely its failure to use historic data to anticipate request spikes.

Nonetheless, it is noteworthy that the *preview* strategy achieves this feat while being conceptually simpler. When run using a different, more uniform Markov model, the strategy is more affected by the higher entropy than the centralized solution. This suggests that the agents' implicit ability to collectively approximate the underlying distribution (where such a distribution exists) and deploy accordingly plays a significant part in the strategy's success.

The *preview* strategy is also very robust with respect to the characteristics of the reward and reconfiguration cost functions. The weight, shape and scale of the functions only has a small influence on the results, as long as they are both monotonically decreasing and the reward function is positive in the relevant interval. This bodes well for the reusability of the approach in different contexts.

# 6. RELATED WORK

Fault tolerance in multi-agent systems has been addressed in several publications [6, 9]. These approaches tackle the problem when a fault occurs during execution of a certain task. Middleware based approaches like [9] use a middleware to coordinate the recovery from an agent fault. Other approaches are based on communication as [6]. Reliable group communication is identified as a requirement for mobile reliable agent systems in [3].

Overeinder et al. present a fault tolerance enhancement to the AgentScape middleware in [9]. The AgentScape middleware architecture consists of an AgentScape OS Kernel and a set of additional modules for communication, security, life-cycle management, etc. The ability to tolerate faults is based on standard replication approaches (cf. [11]). Several agents are combined into a replication group. The replication group is transparent to the user. An interesting aspect is that the importance of an agent can evolve over time and the replication strategy can be changed online according to the agent's importance (see [1]). Our approach differs in two aspects. Firstly, we do not consider agent faults during task execution but assume that another agent will available for the next task after an agent has failed. Secondly, we explicitly aim for a decentralized solution in order to avoid that a central system constitutes a single-point-of-failure.

Kumar and Cohen present an approach for fault tolerant broker agents [6]. Broker agents are used to facilitate co-operation between agents by supporting agent lookup and agent-to-agent connections. The approach is based on the notion of joint intentions and team work. In principle, if one broker fails, another broker steps in and adopts the failed broker's commitments. As our scenario does not require central broker agents, we consequently do not need to take care of central agents failures.

In contrast to the approaches presented above, the FANTOMAS approach [10] for fault-tolerant mobile agent systems emphasizes the autonomy of agents. Consequently, the authors argue that no central fault tolerance manager should be employed. The approach tackles the similar problem of an agent failure during task execution. Pals et al. propose using mobile agent pairs. Each pair consist of the actual agent and a message logging agent which is located on a different node. Both agents monitor each other. After a failure recovery, the agent's state is rebuilt from local checkpoints. Our approach is similar to the FANTOMAS approach with respect to the emphasis on the autonomy of agents and the avoidance of a central fault tolerance management system. In contrast to this approach,

we do not consider agent faults during task execution but tackle the problem that a replacement agent needs to be available for the next task after an agent has failed. Whether both approaches could benefit from each other might be subject of future research.

Mohindra et al. propose a variant of n-version programming combined with checkpoints and rollbacks to improve the reliability of mobile agent systems in [7]. Each agent has different ways to perform a certain task. If a failure occurs in one implementation, the agent can rollback and try a different one, which is possibly not affected by the failure. Thus, the agent is independent of central fault management systems.

Dynamic task allocation is another research area which is closely related to this work. It deals with the problem of allocating tasks to processing units. In [8], the authors present a solution for the problem of allocating newly assembled trucks to a number of painting booths. The painting booths are agents which can autonomously change the color they are using as the colors requested by the customers change over time. However, a change of color is associated with a penalty. While this scenario in principle matches our abstract scenario, failures of the painting booth agents are not considered. The treated optimization problem is to *minimize the time* for painting a predetermined set of trucks, which is carried out using an insect-based algorithm. This problem is quite different from maximizing availability and reliability of the agent based system in spite of failures: we are *minimizing the failed service requests* in a *fixed time frame*. Additionally, our scenario is more general since it supports multiple painting operations performed on the same truck. Nonetheless, tailoring insect-based algorithms to our problem domain seems possible and might yield interesting results.

# 7. CONCLUSION AND FUTURE WORK

We have proposed an agent-based approach to fault tolerance which, unlike many existing approaches, does not require central management facilities, specific coordination mechanisms or direct communication in order to achieve its aim. In our experiments, our solution has consistently achieved high reliability and availability.

Considering that we used a relatively unsophisticated strategy and reward and cost functions that were not highly optimized for specific scenarios, it is particularly remarkable that the decentralized solution performed on par with the robust centralized solution. The agents thus emulated what could be considered rational global behavior.

Compared to the basic centralized solution, the autonomous agents coped with high failure rates far better and achieved perfect availability starting from a lower number of agents. The robustness of the solution, both with respect to different settings and to the functions we used, suggests that it could be adapted to comparable scenarios with very limited effort.

We believe that using agent-based coordination to achieve fault tolerance is even more generally applicable, even if some of the idealizing assumptions we made are relaxed. We are e.g. considering an extension of the narrow fault model of our scenario to also include crash failures during task execution. In addition, we are looking into providing stateful services, as compared to the stateless services used in this paper. A combination of our approach with some of the related approaches presented above may serve as a starting point.

## Acknowledgements

We thank Lothar Wendehals for comments on earlier versions of the paper.

# 8. REFERENCES

[1] S. Aknine and O. Marin. Role of replication planning for fault tolerant multiagent systems. In *Proceedings of the Fifth European Workshop on Adaptive Agents and Multi-Agent Systems*, March 2005.

[2] A. Birolini. *Reliability engineering : theory and practice*. Springer Verlag, Berlin, 1999. 3rd Edition.

[3] F. M. de Assis Silva and R. J. de Araújo Macêdo. Reliability requirements in mobile agent systems. In *Proceedings of the Second Workshop on Tests and Fault Tolerance, Curitiba, Brazil*, July 2000.

[4] F. Klein and H. Giese. Analysis and Design of Physical and Social Contexts in Multi-Agent Systems. In A. Garcia, R. Choren, C. Lucena, A. Romanovsky, T. Holvoet, and P. Giorgini, editors, *Software Engineering for Multi-Agent Systems IV*, volume 3914 of *Lecture Notes in Computer Science*, pages 91–108. Springer Verlag, April 2006.

[5] F. Klein and H. Giese. Grounding Social Interactions in the Environment. In D. Weyns, V. Parunak, and F. Michel, editors, *Environments for Multiagent Systems II*, volume 3830 of *Lecture Notes in Artifical Intelligence*, pages 139–162. Springer Verlag, March 2006.

[6] S. Kumar and P. R. Cohen. Towards a fault-tolerant multi-agent system architecture. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, pages 459–466, New York, NY, USA, 2000. ACM Press.

[7] A. Mohindra, A. Purakayastha, and P. Thati. Exploiting non-determinism for reliability of mobile agent systems. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 144–156. IEEE Computer Society, 2000.

[8] S. Nouyan, R. Ghizzioli, M. Birattari, and M. Dorigo. An insect-based algorithm for the dynamic task allocation problem. Technical Report TR/IRIDIA/2005-031, Université Libre de Bruxelles, December 2005.

[9] B. Overeinder, F. Brazier, and O. Marin. Fault Tolerance in Scalable Agent Support Systems: Integrating DARX in the AgentScape Framework. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 688, Washington, DC, USA, 2003. IEEE Computer Society.

[10] H. Pals, S. Petri, and C. Grewe. FANTOMAS: Fault Tolerance for Mobile Agents in Clusters. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1236–1247, London, UK, 2000. Springer Verlag.

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

[12] M. Tichy and H. Giese. A Self-Optimizing Run-Time Architecture for Configurable Dependability of Services. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems II*, volume 3069 of *Lecture Notes in Computer Science*, pages 25–51. Springer Verlag, 2004.

[13] T. D. Wolf and T. Holvoet. Emergence versus self-organisation: Different concepts but promising when combined. In S. Brueckner, G. D. M. Serugendo, A. Karageorgos, and R. Nagpal, editors, *Engineering Self-Organising Systems: Methodologies and Applications*, volume 3464 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.