Towards a Priority-Based Task Distribution Strategy for an Artificial Hormone System

Eric Hutter(✉) and Uwe Brinkschulte

Goethe University Frankfurt, Frankfurt am Main, Germany
{hutter,brinks}@es.cs.uni-frankfurt.de

Abstract. This paper presents a priority-based task distribution strategy as an extension to the Artificial Hormone System (AHS). The AHS is a distributed middleware based on self-organization principles. It allows to distribute tasks to processing nodes in a self-organizing way while neither having a single-point-of-failure nor requiring external user input. Node failures are detected automatically, resulting in relocation of any affected tasks to operational nodes. This provides self-healing capabilities if sufficient computational resources are available.

Our extension allows tasks to have priorities and enables self-healing by gracefully degrading the system based on the task priorities if the computational resources are not sufficient to completely self-heal the system. We present our extension and analyze its worst-case time bounds for self-configuration as well as self-healing. Quickly degrading the system in overload situations requires a strategy deciding which tasks to stop in such situations. We present a simple strategy and analyze its worst- and average-case self-healing duration.

Keywords: Artificial Hormone System · Organic Computing · Self-organization · Self-healing · Task distribution

1 Introduction

New ways to handle the increasing complexity observed in embedded systems while simultaneously coping with component failures have to be found. One promising approach is to adapt self-organizational principles to computer systems.

The *Artificial Hormone System* (AHS) middleware [14] adapts the natural endocrine system in order to decentrally manage tasks in a distributed system: By exchanging digital messages, called *hormones* after their biological model, via a communication network, tasks can be distributed in the system in a flexible and self-organizing manner, exhibiting properties such as self-configuration and self-healing.

This paper deals with an extension to the AHS middleware that supports task assignment priorities, thus allowing graceful system degradation if the amount of

A. Brinkmann et al. (Eds.): ARCS 2020, LNCS 12155, pp. 69–81, 2020.
https://doi.org/10.1007/978-3-030-52794-5_6

node failures is too big to allow the system to *fully* heal itself. Our contribution is three-fold:

1. We describe a priority-based task-decision strategy for the AHS.
2. We derive hard time bounds for this strategy's self-configuration and self-healing capabilities.
3. We analyze the worst- and average-case time required to degrade a system in an overload situation.

The paper is structured as follows: We first present related work and the general AHS in Sects. 2 and 3. Section 4 describes our priority-based extensions to the AHS. Its worst-case time bounds are derived in Sect. 5. Section 6 proposes a simple degradation strategy to self-heal the system in overload situations and analyzes its worst- and average-case healing times. Finally, Sect. 7 concludes this paper.

2 Related Work

The approach presented in this paper enables a distributed system to recover from hardware failures by dynamically (re)configuring itself. A classical way to improve a system's robustness against such types of failures is the duplication of functional units: By providing each unit with an identical, redundant unit in hot stand-by, a limited number of failures can be compensated. This pattern is frequently used for safety-critical control units in the automotive domain, albeit recent approaches like the AutoKonf project [10] try to reduce costs by sharing a single backup between multiple different control units so that a single failure can be compensated.

In contrast, our approach allows to assign tasks to a distributed system's computing nodes in a self-organizing way, allowing more flexibility. Task priorities allow gradual system degradation after so many node failures have occurred so that the system can no longer fully recover. This flexible mechanism allows to reduce the number of required backup nodes while still being able to tolerate a limited number of node failures.

Our approach is inspired by multiple general research trends: IBM's *Autonomic Computing* initiative [8] introduced *self-x* properties such as self-configuration, selfoptimization or selfhealing. These properties can also be observed in systems based on *Organic Computing* principles [11]: Organic Computing can be characterized as a postponement of various decisions to the system's run-time that were traditionally made at design time [9], allowing the system to dynamically adapt to changed operational conditions [13].

The resulting dynamism distinguishes our concept from approaches like [6] where a (offline) precomputed adaption scenario is applied in case of node failures or overload situations. In contrast, our approach completely postpones the calculation of adaption responses to the run-time and thus allows a more dynamic reaction.

Nevertheless, our concept is by far not the only approach to assign tasks in distributed systems.

Contract Net Protocols [12] can be employed to distribute tasks to agents in a multi-agent system. The approach presented in [17] consists of an improved Contract Net Protocol for task assignment and employs self-healing capabilities as well as task priorities. Yet, contrarily to our approach, it is not completely decentralized and does not guarantee hard real-time bounds.

Contract Net Protocols have also been used for task assignment in Wireless Sensor Networks (WSNs), e.g. in [4]. Other recent research in this domain utilizes self-organization principles [16], game theory [5] or particle swarm optimization [7,15]. Many of these approaches employ some kind of task priority, e.g. deadline-based priorities or number of dependent tasks in a task graph. Additionally, with WSN nodes typically having a limited energy budget, energy-efficiency is one of the main goals of these approaches, rather than guaranteeing real-time behavior as with our approach. In addition, WSNs do not guarantee that any two nodes have a direct communication link as required by our approach, thus needing different methods to distribute the tasks.

To the best of our knowledge, no self-organizing middleware for prioritized task allocation in distributed systems comparable to our approach exists.

3 The Artificial Hormone System

The *Artificial Hormone System (AHS)* is a completely decentralized middleware based on Organic Computing principles that allows the distribution of tasks in a self-organizing way. If a *processing element (PE)* fails, the remaining PEs will compensate this failure by re-configuring themselves.

The AHS works by realizing control loops based on *hormones*, which are short digital messages, on all PEs in the system: They exchange *eager values* for all tasks, indicating their suitability for each task. In every cycle of the hormone control loop (called *hormone cycle*), each PE tries to make a decision upon one task by comparing its own eager value with all received eager values. If it has sent the highest eager value for some task T in the current cycle, it is allowed to start executing T.

Once T has been started, its PE will start sending out a *suppressor* hormone for this task. Upon receiving this suppressor, the other PEs will reduce their eager values for T accordingly, preventing them from taking additional instances of T.

Accelerator hormones act antagonistically to suppressors by increasing a task's eager value: Tasks that cooperate in some way or work on similar problems may be defined as related. When a task T is running on some PE_γ, accelerators for all tasks related to T are spread in the vicinity of PE_γ. This furthers the execution of those tasks on neighboring PEs, forming functional clusters of related tasks.

Figure 1 shows the basic principle of the hormone control loop. It is completely decentralized and thus, the AHS has no single point of failure. In addition, the original AHS (without our priority extension) can guarantee hard time bounds, allowing its use in real-time systems:

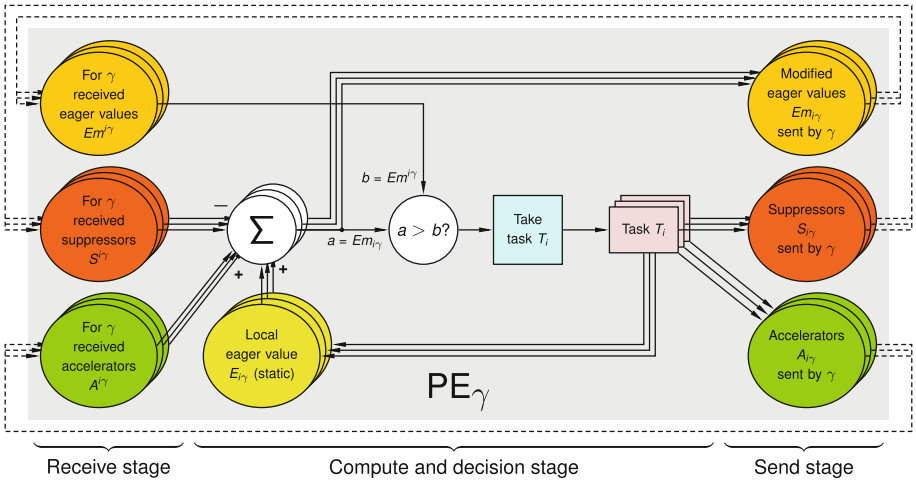


Fig. 1. Hormone control loop executed on PE_γ. For each task T_i, a modified eager value $Em_{i\gamma}$ (its static local eager value $E_{i\gamma}$ plus all received accelerators $A^{i\gamma}$ minus all received suppressors $S^{i\gamma}$) is sent to all PEs. If the sent modified eager value is positive and greater than all received modified eager values, T_i may be taken. Thus, suppressors for T_i will be sent to *all* other PEs (preventing infinite assignments of T_i) and accelerators for tasks *related* to T_i will be sent to *neighboring* PEs, forming clusters of related tasks on them.

- The system's initial self-configuration requires at most m hormone cycles, where m is the number of tasks to distribute.
- In case a PE running m_f tasks fails, the AHS will automatically perform a self-healing by reassigning those tasks among the healthy PEs. This takes at most $m_f + a$ hormone cycles where a is the number of hormone cycles required to notice the failure (by missing suppressors for the failed tasks).[1]

The length of each hormone cycle can be chosen arbitrarily, a lower bound is only imposed by the communication bus' latency and bandwidth.

For more detailed information on the AHS, please refer to [1–3,14].

4 A Priority-Based Task Decision Strategy

4.1 Motivation

The AHS' self-healing capabilities are insufficient if the remaining PEs do not have enough computational resources to execute *all* tasks. With many systems consisting of tasks with mixed criticality levels, this may easily lead to situations where low-criticality tasks are running, but high-criticality tasks previously executed on the failed PE cannot be reassigned. In fact, it is undefined which task subset will be running in such situations.

[1] In the AHS' current implementation, $a = 2$ hormone cycles holds.

The AHS models system load by means of *load suppressors* a task sends to the PE it is running on, thus limiting the number of tasks the PE can take. If it is fully loaded, it will send an eager value of 0 for all remaining tasks. Therefore, situations of system overload can in principle be recognized by examining the hormones broadcasted in the system.

We thus implemented an AHS extension that allows to give each task a *priority*. This priority is used to (a) control the order in which tasks are assigned during the initial self-configuration and (b) resolve system overload situations by stopping tasks of low priority, freeing capacities for high-priority tasks.

4.2 Conception

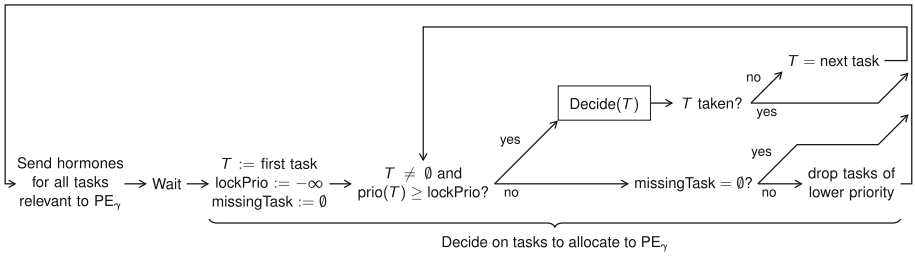
Our approach, the priority-based task decision strategy, is an extension of the AHS' so-called *aggressive* task decision strategy [2]: Each PE may take at most one task per hormone cycle. If more than one task were taken per cycle and PE, all tasks could be assigned before accelerators had a chance to become effective, failing to cluster related tasks. Using the aggressive task decision strategy, each PE actively searches for a task it may take in each cycle. This allows at least one PE to take one task per cycle, resulting in the time bound of m hormone cycles to distribute m tasks mentioned in Sect. 3.

Our priority-based task decision strategy has the same operating principle with the following differences: Each task has an integer priority that is known to and equal on *all* PEs in the system. Each PE searches its task list for a task to be taken in the order of *descending* priorities. If the received hormones suggest that another PE has won some task T (and thus *might* take it in this cycle), no task T' having a lower priority than T's may be taken in the current cycle, ensuring a correct order of task assignment. Figure 2a sketches this procedure: The variable *lockPrio* is used to stop deciding on tasks after a higher-priority task has been identified that may be taken on another PE in this cycle. The variable *missingTask* is used to track a high-priority task that is not running in the system because of an overload situation. Both variables are set by the sub-procedure Decide(T) as shown in Fig. 2b.[2]

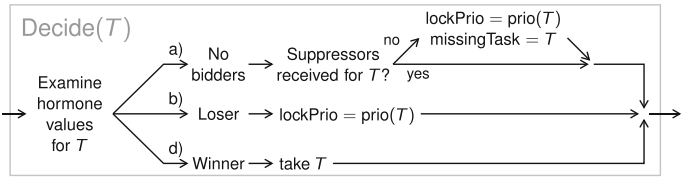
The decide procedure works by comparing the local eager value sent with all received eager values. There are three possible outcomes of this comparison:

a) No bidders: No PE sent a *positive* modified eager value for T. If suppressors were received for T instead, T is taken somewhere and the decision procedure may continue to the next task. Else, no PE has capacities to take T, so the system is in an overload situation. In this case, lockPrio is set to prevent tasks of lower priority from being taken and *missingTask* is used to track this situation.
b) Loser: The local PE did not send the highest eager value for T. Thus, it sets *lockPrio* to prevent taking any task of lower priority.

[2] For the sake of brevity, only a simplified variant of Decide (T) is depicted with some parts omitted, e.g. the offer mechanism belonging to the AHS' self-optimization.



(a) Hormone loop with priority-based task decision strategy



(b) Procedure Decide(T)

Fig. 2. Priority-based task decision strategy

c) Winner: The local PE sent the highest eager value for T. Thus, it takes T (and will send a suppressor for T in the next cycle, preventing the other PEs from taking an additional instance of this task).

The current hormone cycle ends once a task has been taken or a task with lower priority than *lockPrio* has been reached. If, in the latter case, *missingTask* is set, some high-priority task is not running because of an overload situation. In order to resolve this situation, each PE will try to drop tasks of low priority in order to free up resources and send an eager value >0 for this task.

This is, however, based on the following fundamental assumption:

Assumption. *Any task may be (temporarily) stopped* at any time *in order to free up resources for tasks of* higher *priority.*

It is up to a *task dropping strategy* to decide on which specific tasks will be stopped in an overload situation; more information on this matter as well as the priority-based task decision's time bounds will be presented in the following sections.

5 Worst-Case Analysis

Our priority-based extension can still guarantee real-time behavior for self-configuration as well as self-healing if the remaining capacities are sufficient to redistribute all failed tasks:

Self-Configuration. It can be shown that it takes at most $2m - 1$ hormone cycles to assign m tasks during the system's initial self-configuration. As our strategy is based on the aggressive strategy, a similar argument to the one given in [2] can be employed: Each PE searches actively for a won task, thus at least one task is taken per cycle in the system. However, after a task T is taken by some PE_α in cycle i, all other PEs will still send out an eager value >0 for T in cycle $i + 1$ before PE_α's suppressor for T finally becomes effective. This introduces one delay cycle in which no task allocation can happen after the last task of each priority level has been assigned. Thus, if all m tasks have different priorities, $m - 1$ delay cycles are introduced and self-configuration takes at most $2m - 1$ hormone cycles.

Self-Healing. If a PE fails, it won't send any more hormones. Thus, it is possible to detect this failure by missing hormones. This takes a constant amount of a hormone cycles.[3] Afterwards, the failed tasks are automatically re-assigned with the time bound for self-configuration applying. If m_f tasks were running on the failed PE and the remaining PEs' resources suffice to take all those tasks, self-healing thus takes at most $2m_f - 1 + a$ hormone cycles.

If, however, the remaining capacities do *not* suffice to take all failed tasks, the system is considered to be in an overload situation. Since a premise of our priority-based task decision is to allow gracefully degrading the system in such situations, the next section will deal with overload situations.

6 Overload Situations

As mentioned in Sect. 4.2, a task dropping strategy is responsible for deciding on which specific tasks to stop in overload situations. However, a model is required to facilitate the analysis of such situations. Thus, let v be the number of PEs that remain operational. Let $\mathrm{PE}_\times$ denote the failed PE and $\mathrm{PE}_1 \ldots \mathrm{PE}_v$ denote the remaining PEs. Additionally, we will make the following assumptions:

- All tasks have different priorities.
- All tasks induce equal load to the PEs and each PE may execute any task.
- At the instant $\mathrm{PE}_\times$ fails, $\mathrm{PE}_\times$ and $\mathrm{PE}_1 \ldots \mathrm{PE}_v$ are each completely utilized by executing exactly m tasks.
- No additional PE failure occurs during self-healing.

The resulting model is visualized in Fig. 3.

6.1 Task Dropping Strategy

We now propose the following task dropping strategy to resolve an overload situation:

[3] As mentioned before, $a = 2$ holds.

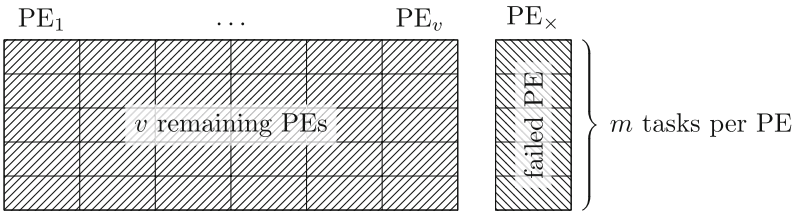


Fig. 3. Overload model used during analysis

Strategy. *Upon noticing an overload situation, each PE shall stop* all *running tasks that have a priority lower than the priority of the highest-priority task that is currently not running.*

In the context of Fig. 2, this strategy basically stops all tasks having a priority lower than *missingTask*'s priority. After the tasks have been stopped, the system re-configures itself by assigning as many of the highest-priority non-running tasks as possible. Since the system is in overload, not all tasks can be assigned, but no more tasks will be stopped by the next invocation of the task dropping strategy. As this is arguably a very simple strategy, we called it *naive task dropping*. In addition, analyzing its worst-case in the given model is straightforward:

Theorem 1 (Worst-Case Analysis for Overload Situations). *Self-healing in an overload situation takes at most* $2mv + a$ *hormone cycles when using the naive task dropping strategy.*

Proof. It takes a constant amount a of hormone cycles to notice the failure of $PE_\times$ due to missing suppressors. All v remaining PEs are fully utilized, so the system is in an overload situation.

Thus, in the next cycle, *missingTask* is set to the highest-priority task that was previously running on $PE_\times$ and no task is taken in the system. The strategy will now stop all tasks with a priority lower than *missingTask*'s priority; if *missingTask* is the single-highest priority task, a total of mv tasks will be stopped.

Starting with the following cycle, the mv highest-priority tasks (of all $mv+m$ non-running tasks) will be assigned, taking $2mv - 1$ cycles at most.

As a result, no more than $2mv - 1 + 1 + a = 2mv + a$ hormone cycles are required to complete the self-healing and reach a stable system state again. □

6.2 Average-Case Analysis

In this paper, we additionally want to analyze the time required for self-healing when using this strategy in the *average* case. Although an average-case analysis is not relevant in the context of real-time systems, we decided to nevertheless analyze it in this regard: The expected self-healing duration might especially be of interest for scenarios that don't require *hard* real-time bounds.

Preparations. In order to facilitate this analysis, some arrangements have to be made:

Definition 1. *Let $n, k \in \mathbb{N}$. Then, the* rising factorial $n^{\overline{k}}$ *shall be defined as*

$$n^{\overline{k}} := \underbrace{n \cdot (n+1) \cdots (n+k-1)}_{k \text{ factors}} = \prod_{i=n}^{n+k-1} i.$$

Lemma 1. *For $k, m \in \mathbb{N}$ with $k \geq 1$,*

$$\sum_{i=1}^{k} i^{\overline{m}} = k \cdot \frac{(k+1)^{\overline{m}}}{1+m} \quad \textit{holds.}$$

Proof. Multiplying with $(m!/m!)$ allows to convert the summands to binomial coefficients:

$$\sum_{i=1}^{k} i^{\overline{m}} = m! \cdot \sum_{i=1}^{k} \frac{i^{\overline{m}}}{m!} = m! \cdot \sum_{i=1}^{k} \binom{i+m-1}{m} = m! \cdot \sum_{i=m}^{k+m-1} \binom{i}{m}$$

This sum can now be simplified using the identity $\sum_{i=r}^{n} \binom{i}{r} = \binom{n+1}{r+1}$:

$$= m! \cdot \binom{k+m}{m+1} = m! \cdot \frac{k^{\overline{m+1}}}{(m+1)!} = k \cdot \frac{(k+1)^{\overline{m}}}{1+m}.$$

□

Analysis. We can now analyze the naive task dropping strategy's average case within our overload situation model, assuming all tasks are distributed randomly to the available PEs. For this reason, we quantify the number of tasks stopped on average:

Theorem 2. *Let X be a random variable representing the number of tasks stopped by the naive task dropping strategy and $\mathbf{E}[X]$ its expected value. Then, $\mathbf{E}[X] = (m^2 v)/(1+m)$ holds.*

Proof. We will first calculate the probability distribution of X by assuming that all $(v+1) \cdot m$ tasks are distributed in sequence of descending priorities to mv positions on $\mathrm{PE}_1 \ldots \mathrm{PE}_v$ and mv positions on $\mathrm{PE}_\times$.

Obviously, $0 \leq X \leq mv$ holds: In case the m lowest-priority tasks are running on $\mathrm{PE}_\times$, no tasks have to be stopped. If the highest-priority task is running on $\mathrm{PE}_\times$, *all* tasks on $\mathrm{PE}_1 \ldots \mathrm{PE}_v$ are stopped.

Generalizing this argument yields

- $X = mv \iff$ The highest-priority task is set to one of $\mathrm{PE}_\times$'s m positions.
- $X = mv - 1 \iff$ The second-highest-priority task is the first task to be set to one of $\mathrm{PE}_\times$'s m positions.

  ⋮

– $X = 0 \iff$ The $(m \cdot (v+1) - 1)$-highest-priority task (which is the m-lowest-priority task) is the first task to be set to one of $\mathrm{PE}_\times$'s m positions.

Thus, the probability distribution of X is given by

$$\begin{aligned}
\mathbf{P}(X = mv) &= \frac{m}{m \cdot (v+1)} \\
\mathbf{P}(X = mv-1) &= \frac{mv}{m \cdot (v+1)} \cdot \frac{m}{m \cdot (v+1) - 1} \\
\mathbf{P}(X = mv-2) &= \frac{mv}{m \cdot (v+1)} \cdot \frac{mv-1}{m \cdot (v+1) - 1} \cdot \frac{m}{m \cdot (v+1) - 2} \\
&\vdots \\
\mathbf{P}(X = 0) &= \underbrace{\frac{mv}{m \cdot (v+1)} \cdot \frac{mv-1}{m \cdot (v+1) - 1} \cdots \frac{1}{m+1}}_{mv \text{ highest-priority tasks on } \mathrm{PE}_1 \ldots \mathrm{PE}_v} \cdot \frac{m}{m}
\end{aligned}$$

This is equivalent to

$$\mathbf{P}(X = j) = \frac{\prod_{i=j+1}^{mv} i}{\prod_{i=m+j+1}^{mv+m} i} \cdot \frac{m}{m+j} = m \cdot \frac{\prod_{i=j+1}^{mv} i}{\prod_{i=m+j}^{mv+m} i} \tag{1}$$

for all $0 \leq j \leq mv$. Figure 4 plots the probability distribution as given by this equation for arbitrarily chosen values of m and v.

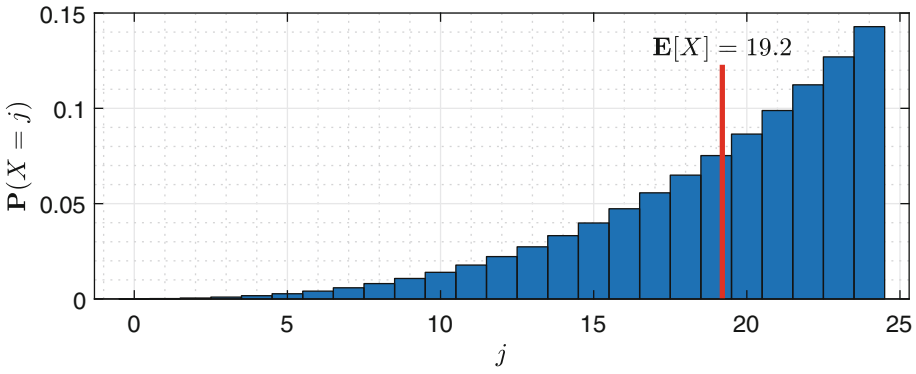


Fig. 4. Probability distribution of number of tasks stopped by naive task dropping strategy for $m = 4$ and $v = 6$

Equation 1 now allows us to calculate $\mathbf{E}[X]$:

$$\begin{aligned}
\mathbf{E}[X] = \sum_{j=0}^{mv} j \cdot \mathbf{P}(X=j) &= \sum_{j=1}^{mv} j \cdot m \cdot \frac{\prod_{i=j+1}^{mv} i}{\prod_{i=m+j}^{mv+m} i} \\
= m \cdot \sum_{j=1}^{mv} j \cdot \frac{(mv)!/j!}{(mv+m)!/(m+j-1)!} &= m \cdot \sum_{j=1}^{mv} \frac{(m+j-1)!/(j-1)!}{(mv+m)!/(mv)!} \\
= \frac{m}{\prod_{i=mv+1}^{mv+m} i} \cdot \sum_{j=1}^{mv} \left(\prod_{i=j}^{m+j-1} i \right) &= \frac{m}{(mv+1)^{\overline{m}}} \cdot \sum_{j=1}^{mv} j^{\overline{m}}
\end{aligned}$$

This expression can be simplified using Lemma 1:

$$= \frac{m}{(mv+1)^{\overline{m}}} \cdot \frac{mv \cdot (mv+1)^{\overline{m}}}{1+m} = \frac{m^2 v}{1+m}.$$

□

Discussion. When examining the number of tasks dropped by the naive task dropping strategy, it becomes obvious that its worst case is not substantially worse than the average case, especially for large values of m:

$$\lim_{m \to \infty} \frac{\text{worst case}}{\text{average case}} = \lim_{m \to \infty} \frac{mv}{\frac{m^2 v}{1+m}} = \lim_{m \to \infty} \left(1 + \frac{1}{m}\right) = 1.$$

As a result, no significant outliers from the average case are to be expected when utilizing this task dropping strategy.

In addition, initial self-*configuration* for mv tasks requires $2mv - 1$ hormone cycles, while self-*healing* in an overload situation takes at most $2mv + a$ hormone cycles: Both bounds are linear in the number of tasks with different (and small) additive constants.

7 Conclusion

This paper described a priority-based extension to the AHS middleware. We analyzed its worst-case time bounds for self-configuration and self-healing. Additionally, a strategy was proposed allowing to degrade the system in case of an overload situation. Its worst and average cases were analyzed. The results show that this strategy does not perform substantially worse in its worst case than it does on average.

Future work will deal with thorough evaluations of our concept as well as further research on degrading the system in overload situations, especially with developing more elaborate task dropping strategies, possibly guaranteeing even better time bounds for self-healing in overload situations.

Additionally, our current analyses assume a reliable communication network. Although empirical experiments suggest that the AHS can handle a limited degree of communication failures quite well, we plan to shift our research focus to guaranteeing time bounds and the system's overall consistency even in the presence of such failures.

References

1. Brinkschulte, U.: Increasing the stability of an Artificial Hormone System for task allocation by accelerator bounds. In: 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2013), pp. 1–10. IEEE, Paderborn, June 2013
2. Brinkschulte, U., Pacher, M.: An aggressive strategy for an artificial hormone system to minimize the task allocation time. In: 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, pp. 188–195. IEEE, Shenzhen, April 2012
3. Brinkschulte, U., Pacher, M., von Renteln, A., Betting, B.: Organic real-time middleware. In: Higuera-Toledano, M.T., Brinkschulte, U., Rettberg, A. (eds.) Self-Organization in Embedded Real-Time Systems, pp. 179–208. Springer, New York (2013). https://doi.org/10.1007/978-1-4614-1969-3_9
4. Chen, L., Xue-song, Q., Yang, Y., Gao, Z., Qu, Z.: The contract net based task allocation algorithm for wireless sensor network. In: 2012 IEEE Symposium on Computers and Communications (ISCC), pp. 600–604, July 2012
5. Edalat, N., Tham, C.K., Xiao, W.: An auction-based strategy for distributed task allocation in wireless sensor networks. Comput. Commun. **35**(8), 916–928 (2012)
6. Fohler, G., Gala, G., Pérez, D.G., Pagetti, C.: Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator. In: 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, January 2018
7. Guo, W., Li, J., Chen, G., Niu, Y., Chen, C.: A PSO-optimized real-time fault-tolerant task allocation algorithm in wireless sensor networks. IEEE Trans. Parallel Distrib. Syst. **26**(12), 3236–3249 (2015)
8. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
9. Müller-Schloer, C., Tomforde, S.: Organic Computing – Technical Systems for Survival in the Real World. AS. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68477-2
10. Orlov, Sergey, Korte, Matthias, Oszwald, Florian, Vollmer, Pascal: Automatically reconfigurable actuator control for reliable autonomous driving functions (AutoKonf). 10th International Munich Chassis Symposium 2019. Proceedings, pp. 355–368. Springer, Wiesbaden (2020). https://doi.org/10.1007/978-3-658-26435-2_26
11. Schmeck, H.: Organic computing - a new vision for distributed embedded systems. In: Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005), pp. 201–203, May 2005
12. Smith, R.G.: The contract net protocol: high-level communication and control in a distributed problem solver. IEEE Trans. Comput. **C–29**(12), 1104–1113 (1980)
13. Tomforde, S., Sick, B., Müller-Schloer, C.: Organic computing in the spotlight. arXiv:1701.08125 [cs], January 2017. http://arxiv.org/abs/1701.08125
14. von Renteln, A., Brinkschulte, U., Pacher, M.: The artificial hormone system—an organic middleware for self-organising real-time task allocation. In: Müller-Schloer, C., Schmeck, H., Ungerer, T. (eds.) Organic Computing—A Paradigm Shift for Complex Systems, pp. 369–384. Springer, Basel (2011). https://doi.org/10.1007/978-3-0348-0130-0_24
15. Yang, J., Zhang, H., Ling, Y., Pan, C., Sun, W.: Task allocation for wireless sensor network using modified binary particle swarm optimization. IEEE Sens. J. **14**(3), 882–892 (2014)

16. Yin, X., Dai, W., Li, B., Chang, L., Li, C.: Cooperative task allocation in heterogeneous wireless sensor networks. Int. J. Distrib. Sens. Netw. **13**(10), 1550147717735747 (2017)
17. Zhang, J., Wang, G., Song, Y.: Task assignment of the improved contract net protocol under a multi-agent system. Algorithms **12**(4), 70 (2019)