Nature-Inspired Interference Management in Smart Peer Groups

Felix Maximilian Roth, Christian Krupitzer, Sebastian VanSyckel, Christian Becker
University of Mannheim
Schloss, 68131 Mannheim, Germany
{*felix.maximilian.roth, christian.krupitzer, sebastian.vansyckel, christian.becker*}*@uni-mannheim.de*

***Abstract*—Applications in intelligent environments are context-aware and context-altering in order to support users in their everyday tasks. In multi-user environments with shared context, interferences are likely to occur. An interference is an application-induced context that forces other applications to react. In our COMITY project, we developed interference detection and resolution algorithms. However, especially the constraint satisfaction-based resolution algorithm is computationally expensive. It requires a full-fledged machine in order to achieve runtimes suitable for interactive systems. In spontaneously formed smart peer groups, such infrastructure is not given.**

In this paper, we present an approach for detecting and resolving interferences in smart peer groups. The approach is inspired by two coordination mechanisms from nature. We map the problems of interference detection and resolution to the rules of the flocking mechanism, show how to implement a flocking-based interference management, and present a local inhibition-based leader election for smart peer groups in order to elect the coordinating entity. Finally, we show the feasibility of our approach by evaluating our prototype.

***Keywords*-Pervasive Computing; Interference Management; Nature-Inspired; Flocking; Local Inhibition**

I. Introduction

Ubiquitous or pervasive computing scenarios make use of a rapid development of mobile devices by integrating a multitude of computing devices into our everyday environments. These devices are connected and cooperate with each other in order to execute applications. Moreover, these applications depend on their context and are even able to change their context. However, when applications that share their context have contradicting goals, problems may arise. As an example, application A is a phone application that requires a silent environment. Application B is a music application and hence turns on the music. Consequently, application A cannot be executed anymore. This problem is called an *interference* [1] or sometimes also *conflict* [2]. In our COMITY project ([3], [1], [4]), we developed a framework for interference management in multi-platform pervasive systems. Contracts specify an application's interaction with its context. The framework automatically detects interferences based on contracts and subsequently computes resolution plans via constraint satisfaction. Accordingly, the resolution algorithm's runtime is highly complex and, hence, the approach is not suitable for environments without a computing infrastructure, i.e., spontaneously formed smart peer groups.

This paper presents a nature-inspired interference management approach for such smart peer groups. The approach consists of three parts: *interference detection*, *interference resolution*, and, if necessary, a *coordinator election*. Our approach is inspired by the natural mechanisms of *flocking* and *local inhibition*, where the detection and resolution algorithms follow the rules of flocking. The election of the coordinating entity is based on local inhibition when needed. We describe a mapping of the detection and resolution problems to the flocking rules, and present our resulting interference management model. Furthermore, we implement a prototype system, evaluate it and discuss its suitability.

The remainder of this paper is structured as follows: In the next section, we will discuss related work. Then, Section III presents the system model. Based on the system model, we derive requirements for interference management in smart peer groups in Section IV. Afterwards in Section V, we give a quick background on the flocking and local inhibition mechanisms from nature. Subsequently, we introduce our interference management model in Section VI. In Section VII, we describe implementation details and evaluate our approach. We close with a conclusion and an outlook in Section VIII.

II. Related Work

Several research projects focus on interference management, e.g., [1], [4], [5], [6], [7], [8], and [9]. In our COMITY project, we use a central coordinator to resolve interferences in pervasive environments. Applications provide context configurations to the coordinator. A context configuration includes the interaction of the application with the shared environment and configurations feasible for the application. The coordinator then tries to find a configuration which is feasible for every application in the shared environment ([1], [4]). Lee et al. reply to interferences in smart environments with a central entity, called Conflict Control Manager, which consists of a lock-based conflict detection module and a dynamic-priority-based conflict resolution [5]. Shin and Woo use an ontology model to detect interferences in smart home environments. An interference is detected if several applications share resources, properties, or conditions. The interference management approach frequently checks in the ontology if interferences are occurring at runtime [6].

More approaches for interference management in pervasive environments exist (e.g., [7], [8], and [9]). However, all are designed for pervasive environments with computational infrastructure and are, therefore, not suitable for smart peer groups.

978-1-4799-2947-4/14 $31.00 © 2014 IEEE
DOI 10.1109/IE.2014.27

III. System Model

Our solution for interference management targets systems in spontaneous pervasive environments, so called smart peer groups, in which a multitude of pervasive applications run in parallel. A smart peer group consists of a set of devices, a set of users, and the physical environment in which the devices and users are located. The devices in the system are able to form ad-hoc networks in order to communicate. Users in the system make use of the functionality provided by the devices. Although the devices can be stationary or mobile, as well as resource-rich or resource-poor, we do not assume the presence of a resource-rich device.

Interference management can address resource and environment variables [5]. Resource variables are physical resources, e.g. a display. Environment variables can be measured via sensors, e.g. temperature. So far, we focus on environment variables. For further work we plan to extend our approach by including resource variables as well. Each application has requirements with respect to its context. If the context requirements of an application are not satisfied, it cannot run. A context requirement represents an application's requirement regarding a context variable, e.g., temperature. Our system model describes these requirements by numeric values, having a lower and an upper bound, in which the application can work. Moreover, applications can change their context. The existence of sensors and actuators in the environment allows applications to sense and alter the context. Furthermore, a user may execute several applications, but we assume that each application is executed by a single user.

Finally, devices are equipped with suitable system software in order to participate in the smart peer group and run pervasive applications. The system software must provide basic functionality such as device discovery, resource managers, and communication services. In this paper, we assume that all devices use the same system software.

IV. Requirements

Based on the specified system model, we derive the requirements for interference management next.

An application may join a system with specified context requirements, may change its context requirements during the execution, and eventually leave the system. An interference can only occur in two situations: when joining or when changing context requirements. When an application joins a system, the user expects it to function without much delay. However, it may happen that an interference has to be resolved before the application can be executed. Therefore, an interference should be detected and resolved fast. Ideally, the user does not even notice an occurring interference.

Aside from that, there may be resource-poor devices that have low computation speed and low memory available. We assume that not every smart peer group contains a resource-rich mobile device and, therefore, a device that is responsible for interference management cannot be determined.

Furthermore, a solution has to be computed for every set of context requirements. Thus, if there is no obvious solution, a configuration needs to be calculated that allows at least some applications to work.

Depending on the scenario, a multitude of applications may be in the same system. In order for the approach to manage numerous applications, it should be scalable. Additionally, a failing application should not cause the system to break. If one or more applications fail, the system should compensate and continue working. Thus, the approach needs to be robust.

V. Background: Flocking & Local Inhibition

Many researchers have adopted concepts from nature, such as *flocking*, *foraging*, *chemotaxis*, or *local inhibition*, in order to solve problems in computer science. As we use *flocking* and *local inhibition* for our interference management approach, this section introduces those concepts.

A. Flocking

The behavior of a group of birds in flight is called *flocking*. Similar behavior can be found in schools of fish, swarms of insects, and herds of land animals. There are several benefits of flocking, such as protection from predators, food search, and social advantages [10].

Swarm behavior in a flock occurs without a leader [11] and allows any number of participants. Each bird only has limited local information. Partridge assumes that each bird is aware of itself, its nearest neighbors and the rest of the flock [11]. Therefore, each bird only reacts to birds it can see and that are in close proximity. Three simple rules enables that a crowd of birds flies in a flock [12]:

- **Separation**: Avoid collisions with nearby birds.
- **Alignment**: Attempt to match velocity and direction with nearby birds.
- **Cohesion**: Attempt to move towards the center of nearby birds.

These three rules result in a stable flock where every bird is between a minimum (separation) and a maximum (cohesion) distance from its neighbors. Due to the alignment property, the flock stays stable despite a dynamic environment [13]. Even if a bird is too far or too close, the alignment property is applicable. Responses of a bird to changes in the neighborhood are propagated to other birds in the system, which therefore shows global coordination [14]. Besides the functional principles, flocking behavior shows scalable, adaptive, and robust characteristics as the number of birds in a flock is not restricted, the flock adapts to the environment, and to the joining or leaving of birds.

The concept of flocking is already being used in computer science, e.g., for motion coordination of autonomous agents (e.g., [15]) or robots (e.g. [16]), or for solving optimization problems (e.g., [17]).

B. Local Inhibition

The process by which an embryo develops is called *embryogenesis*. During embryogenesis, some cells within a region may distinguish from other cells [18]. For example, during the embryogenesis process of a fruit fly,

hair follicles need to be produced [19]. At the end of the process, hair follicles are precisely emerged at regular intervals. *Local inhibition* is the process of choosing which cells will differentiate and which will not.

Local inhibition is a self-organizing process and happens without a leader or any global knowledge but by interactions of cells with their neighborhood [20]. Through some random process, a cell tries to distinguish from other cells, e.g., into a hair follicle. At the same time, this cell sends out an inhibitory signal to its neighbors, so that they will not do the same.

Local inhibition is scalable, adaptive, and robust, as the number of cells is not restricted and it can cope with cell death. Nagpal proposes that local inhibition may be useful for leader election [18]. Local inhibition is, for instance, used for a dynamic assignment of cell phone channels [21].

VI. Nature-Inspired Interference Management

For the reactive approach of nature-inspired interference management, the self-organizing concepts of flocking and local inhibition are employed. The requirements of scalability, adaptivity and robustness are satisfied in these self-organizing mechanisms. Hence, it is reasonable to assume that they also hold for this purpose.

In the following, we present our approach for interference management in smart peer groups.

A. Interference Detection

The *alignment* rule in flocking yields that birds fly at the same speed and in the same direction. This means birds adapt to their neighborhood. Applications can try to adapt to their neighborhood by sensing the context and checking if they can also run in this configuration. Hence, for every context requirement it has to be checked if the value of the current context is between the lower and the upper bound. If this is not the case, there is an interference. On the other hand, if no interference is detected, the application can run without altering the context, i.e., the application has adapted to the current context. Thus, the *alignment* rule is responsible for *interference detection*.

The corresponding function (`detectInt(...)`) is shown in Algorithm 1. lB expresses the lower bound of a context requirement, uB respectively the upper bound.Its input parameters are two `ContextList` instances. The first instance is the set of context requirements of an application, the second instance is the current context in the system. The current context consists of one value for each context attribute whereas the set of context requirements for the application consists of an interval with lower and upper bound for each context attribute. The method returns true if no interference is detected and false else. It performs a check to test if the application's set of context requirements is a superset of the current context. If so, no interference exists and the application can be executed without altering the context. Otherwise, an interference needs to be resolved.

Algorithm 1 Interference Detection Algorithm

```
1: procedure DETECTINT(ctxList, currentCtx)
2:     for all ctx1 in ctxList do
3:         ctx2 ← currentCtx.get(ctx1.attribute)
4:         if ctx2 < ctx1.lB OR ctx2 > ctx1.uB then
5:             return false
6:         end if
7:     end for
8:     return true
9: end procedure
```

B. Interference Resolution

If an interference is detected, the *separation* rule can be applied. In flocking the rule helps avoiding collisions. As already an interference has occurred, it is attempted to find a configuration that works for all applications. Thus, for each environment variable is searched a subinterval which is included in each application's interval for the same environment variable. Regarding one environment variable, every application in the system can be executed if the current value for the variable is in this subinterval. If no such interval can be found for a variable, the *cohesion* rule can be employed. This rule yields that birds stay close to nearby birds. With respect to interference management, we interpret this as the new context should be close to the context requirements. Therefore, the average of the context variable is applied. Hence, the *separation* and *cohesion* rules are responsible for resolving an interference after it has been detected.

The method to resolve an interference (`resolveInt(...)`) is shown in Algorithm 2. It has a `ContextList` and an array of `ContextList` as parameters. The single instance is the set of context requirements of the local application. The array includes the set of context requirements for each remote application. The method returns a configuration as `ContextList` calculated based on the input. The method searches an interval for each context attribute that intersects with each set of context requirements. If a set of context requirements does not include the context attribute, it is not considered in the calculation since it can be executed independently of the value of this attribute.

Two nested loops are necessary to compute the intervals - one passing through the context attributes and one passing through each set of context requirements. Inside the inner loop, the `separate(...)` method is called. It has two `Context` instances as parameters and checks if the bounds of those parameters intersect. It returns an array of type double. This array then either includes the lower and upper bound of the calculated interval or an empty set. Starting with two context requirements and successively testing the resulting interval with the next context requirement, the interval may shrink over time or get Ø. If it gets Ø, the `cohere(...)` method is called. It calculates the average value of all passed sets

of context requirements with the passed String attribute. This is based on the *cohesion* rule according to which birds move towards the center of nearby birds. Other implementations of the `cohere` method are possible as for instance the median or the least square method. After passing through the `cohere(...)` method, the inner loop can be exited since all context requirements with this attribute have already been considered for the calculation. If either all sets of context requirements are traversed and an interval is found or the average is calculated, the computed interval is added to the new configuration stored as `ContextList`. Once all context attributes are passed through, the calculated configuration is returned.

Algorithm 2 Interference Resolution Algorithm

```
1:  procedure RESOLVEINT(ctxList, rctxListArray)
2:      attributeList ← ctxList.getAttributes()
3:      config ← Ø
4:      for all a in attributeList do
5:          lB ← ctxList.get(a).lowerBound
6:          uB ← ctxList.get(a).upperBound
7:          for all rctxList in rctxListArray do
8:              ctx ← rctxList.get(a)
9:              interval ← separate((lB, uB), ctx)
10:             if interval ≠ Ø then
11:                 set lB and uB to interval bounds
12:             else
13:                 interval ← cohere()
14:                 set lB and uB to interval bounds
15:                 break
16:             end if
17:         end for
18:         val ← mean of lB and uB
19:         config ← config+newContext(a, val, val)
20:     end for
21:     return config
22: end procedure
```

However, it is easy to find a scenario in which applying these rules does not result in an interference-free configuration. Moreover, in such a scenario it is not possible to satisfy the set of context requirements of every application. There is at least one environment variable which is not included in the calculated subinterval of the *separation* step. Hence, there is no solution for which every application can be executed. By applying the *cohesion* rule, at least a subset of the applications may be executed.

C. Coordinator Election

An interference can occur in three situations:

- The group changes.
- The context changes.
- The context requirements of an application change.

Thus, in those situations it has to be checked for an interference and, if necessary, the interference has to be resolved.

However, in some occasions it is not clear which application will perform the configuration. When an application joins a system or changes its context requirements, the application itself can configure the system. When an application leaves it is not that easy. Here, we apply the coordinator election. In case of an explicit leave, the leaving application could notify remote applications. However, if an application leaves implicitly, e.g. crashes, remote applications are not notified. Thus, the first application that notices the group change initiates the local inhibition.

Another case in which a reconfiguration makes sense is when the context changes (e.g. light is switched off by a person). Here, again the first application that notices the context change should start the local inhibition to decide on an application that performs the reconfiguration.

The concept of local inhibition for leader election can be directly applied to interference management. Each application waits a random time. If the time of an application is up, it sends a message to remote applications to prevent them from further competing. The application is then the leader.

Before explaining the algorithm for local inhibition, a problem that may occur in this regard is addressed. It might be possible that the timer of two or even more applications run out at the same moment (or almost the same moment). Then, a naive approach by that applications inhibit others without waiting for a reply may lead to multiple applications becoming a leader. We call this problem the *inhibition problem*. Therefore, it may be helpful to introduce replies when inhibiting. The application that inhibits, counts how many applications it has inhibited ($wins$) and how many it has not inhibited ($losses$). If an application gets inhibit requests by several application, only the first one counts. An application then may start the reconfiguration process if $wins > losses$. However, it is not guaranteed that it functions with two or more applications starting at the same time. Consequently, if no application wins the inhibition, then no reconfiguration takes place. As devices in the system might be resource-poor and battery-powered, no reconfiguration is assumed to be better than multiple resource wasting reconfigurations. Thus, we implented the `localInhibition()` which is shown in Algorithm 3 as follows.

The `localInhibition()` method initiates the local inhibition procedure on each remote application. Hereby, a random timer is started on every remote application. The application on which the timer is the first to run out, will try to inhibit other applications by calling the `inhibit()` method in order to prevent them from initiating the reconfiguration process. Each application holds an `inhibited` variable that stores if the application is inhibited or not. If the timer of an application runs out whose `inhibited` attribute is true, it does not start to inhibit remote applications. If an application that gets an inhibit request is already inhibited, it returns false and true else. The application that has started the inhibition counts the $wins$ and $losses$. The reason for that is the *inhibition problem* that has been explained above. If,

Algorithm 3 Local Inhibition Algorithm

```
1: procedure LOCALINHIBITION
2:     waitRandomTime()
3:     if inhibited = false then
4:         inhibited ← true
5:         win, lose ← 0
6:         apps ← getRemoteApplications()
7:         for all a in apps do
8:             result ← a.inhibit()
9:             if result = true then
10:                win++
11:            else
12:                lose++
13:            end if
14:        end for
15:        if win > lose then
16:            winner ← true
17:        end if
18:        if winner = true then
19:            resolveInt(...)
20:        end if
21:    end if
22:    inhibited ← false
23: end procedure
```

after every application has replied, $wins > losses$, the application starts the reconfiguration process by executing the `resolveInt(...)` method. If $wins \leq losses$, the reconfiguration is not started by the application. Finally, the `inhibited` attribute is reset to false.

VII. Implementation and Evaluation

In the previous section, we have presented our approach to nature-inspired conflict management. In this section, we describe the implementation of our approach for pervasive systems, evaluate our approach, and discuss its appropriateness for interference management in pervasive systems.

A. Implementation

For showing the feasibility of our approach, we implemented the algorithms presented in Section VI. The implementation is built on our system software, the BASE middleware [22]. The BASE middleware is designed for the requirements in pervasive systems and implemented in Java. It has a lightweight, but extensible core and enables the communication in environments with heterogeneous devices. BASE runs on resource-poor as well as full-fledged devices. Functionality is encapsulated in services. The BASE middleware offers the detection and use of BASE services running on remote devices using RMI. Furthermore, BASE is responsible for device detection, group formation, and communication. Errors in the aforementioned activities are handled on this layer. Therefore, we assume that our solution for interference management runs on top of such a layer.

The `InterferenceManagement` class provides the methods `senseContext()` for getting the current context as well as `setContext(ContextList)` for adjusting the current context. These tasks are actually achieved with sensors and actuators. For the evaluation of our approach, we simulated the surrounding context, as well as context-interaction of the applications. Therefore, the *context distributor* service simulates both sensors and actuators in order to distribute and adjust the current context. The context is set the first time when the first application enters the system. As all applications need to work with the same context, the context distributor is unique in the simulation.

BASE offers abstraction from implementation of the communication. Therefore, we implemented the algorithms for nature-inspired interference management and the context distributor as BASE services in Java. Next, we present our evaluation setting as well as the performance measurements. Afterwards, we analyze the evaluation results and discuss the appropriateness of our approach.

B. Setting

The simulation ran on a notebook with an Intel(R) Core(TM) dual core 2.67 GHz processor and 8 GB RAM operating Windows 7 Professional (64-bit) SP1. All interference management instances - realized as BASE services, each running in its own virtual machine - were executed on this device. Therefore, communication is only simulated. This eliminates biased results because of communication delays. After starting the context distributor, sequentially one application after another starts and joins the system. Depending on the test, an application chooses between 1 to 4 context requirement attributes and then calculates appropriate bounds. Table I illustrates these attributes and distributions. We used temperature, light, humidity, and noise as context attributes. The values were distributed ($\mathcal{N}$ for normal distribution, $Beta$ for beta distribution) among specified intervals. As the beta distribution returns values between 0 and 1, the values need to be scaled in order to have realistic values.

Attribute	Distribution	Parameter	Scaler	Unit
Temperature	$\mathcal{N}$	$\mu = 21.0$, $\sigma = 3.0$	1.0	$°C$
Light Level	$Beta$	$\alpha = 5.0$, $\beta = 995.0$	100000	lux
Humidity	$\mathcal{N}$	$\mu = 45.0$, $\sigma = 20.0$	1.0	%
Volume	$Beta$	$\alpha = 3.0$, $\beta = 7.0$	130	$phon$

Table I: Context Requirements Overview

C. Memory Requirements and Overhead

In order to determine the overhead of our solution, we measured the memory requirements (with Java VisualVM) and message overhead and compared it to a normal BASE setup. According to [1], a minimal BASE configuration requires around 290 kB. The basic memory requirement of the application is additional 173 Bytes. Furthermore, the context list needs 24

Bytes as well as 120 Bytes for each context requirement, which leads to $173Bytes + Size(ContextList)$ with $Size(ContextList) = 24Bytes + 120Bytes * \#ContextRequirements$ for one application. When an application joins a system, it may request the sets of context requirements of remote applications for interference resolution which increases the memory requirements by $\#RemoteApplications * Size(ContextList)$. Even in environments with many applications, the memory usage stays below 1 MB, which can be easily managed even by resource-poor devices.

When an application enters the system, the interference management service exchanges uniquely $2 * \#RemoteApplications$ messages (for requesting remote context requirements and the answers), with $\#RemoteApplications = n - 1$ and n the amount of applications. The local inhibition leads to $2 * \#RemoteApplications$, if one application wins and $n*2*\#RemoteApplications$ in the worst case scenario, when all applications have the same random waiting time. As described above, the reconfiguration started by the winner application adds $2*\#RemoteApplications$ messages. In case an application leaves the system, it uniquely sends $\#RemoteApplications$ messages for notifying all other applications and starting the local inhibition. Compared to the fact, that a BASE instance exchanges messages with all other BASE instances every 100 milliseconds [1] for management reasons, the introduced message overhead is rather low.

D. Performance Metrics

Next, we evaluated the performance of our solution. First, we analyzed the time for the different processes. Second, we evaluated the utility of our solution for the system. Last, we tested our solution in an office scenario.

1) Time: We performed 50 simulation runs with 90 applications with 1, 2, 3, 4, or a random number (between 1 and 4) of context requirements sequentially entering for each measurement. To get rid of outliers, we excluded the largest 10% and the smallest 10% of the measured values and calculated the average of the remaining ones. The time needed for communication between applications is not measured in order to avoid fluctuations based on communication. Several intervals were measured:

1) **No interference**: The measurement starts when the current context has been retrieved and ends when the `detectInt(...)` method finished *without detecting* an interference.
2) **Interference detection**: The measurement starts when the current context has been retrieved and ends when the `detectInt(...)` method finished *by detecting* an interference.
3) **Interference resolution**: The measurement starts when the context requirements of remote applications have been retrieved and ends when the `resolveInt(...)` method finished.

The first measurement (*No interference*) indicates times between 0.67 and 0.86 milliseconds. For the second measurement (*Interference detection*), we measured times between 0.58 and 0.90 milliseconds. This means, that the presence of an interference does not significantly introduce an increase in time for interference detection. Moreover, the smallest values and the average values are less compared to the measurements in the interference-free scenario. The reason might be that the method does not necessarily pass through each context requirement but terminates when an interference has been found. For the *interference resolution*, all values are between 0.13 and 1.99 milliseconds. All times increase slightly with an increase in the number of applications and context requirements. To sum up, the process of joining needs up to 3 milliseconds (in a setting of 90 applications and 4 context variables) neglecting the time for communication.

For the reconfiguration process (neglecting the time for communication), the time is dependent on the inhibiting process, retrieving context requirements, and the time for reconfiguration. We expected the inhibition problem due to similar waiting intervals. For evaluating the likeliness of the problem, we evaluated solutions with random waiting times up to 500, 1000, and 1500 milliseconds. We did not observe the inhibition problem with the setting of 90 applications. Therefore, we recommend to use the 500 milliseconds waiting interval for the sake of time saving in the reconfiguration process. The evaluation of shorter intervals, as well as intervals depending on the number of applications, are part of future work.

2) Utility: In the presence of contradicting context requirements some applications cannot run. To observe how many applications can actually work, a utility function is introduced. The utility function for an application k is defined as

$$u(k) = \begin{cases} 1 & \text{if } k \text{ can run,} \\ 0 & \text{if } k \text{ cannot run.} \end{cases}$$

After an application has joined the system the utility for every application k is calculated. Finally, the results for the set of applications K are aggregated to the overall utility

$$U(K) = \sum_{k \in K} u(k).$$

Each measurement is performed with 1, 2, 3, 4, or a random number (between 1 and 4) of context requirements per application. For each variation, we executed 50 runs with 100 applications. The average for each data point is calculated.

For 1 context requirement (1 Ctx) approximately 50% of the applications can run (e.g. for 20 / 60 / 100 applications 9.66 / 29.44 / 49.98 applications can run). This number decreases for 2 context requirements (2 Ctx) to about 25% (e.g. for 20 / 60 / 100 applications 5.26 / 14.64 / 24.7 applications can run), for 3 context requirements (3 Ctx) to about 12.5% (e.g. for 20 / 60 / 100 applications 2.68 / 7.02 / 12.32 applications can run) and for 4 context requirements (4 Ctx) to approximately 6.75% (e.g. for 20 / 60 / 100 applications 1.38 / 3.96 / 6.74 applications can run). With 4 context requirements the average number of working applications is piecewise even below one

application. For a random number of context requirements, the proportion of working applications is approximately 25% (e.g. for 20 / 60 / 100 applications 4.7 / 14.42 / 24.44 applications can run). The results are illustrated in Figure 1. The x-axis represents the number of applications, the y-axis the overall utility. Thus, the following effect can be observed: The more context requirements the applications in a system have, the fewer applications can run in total.

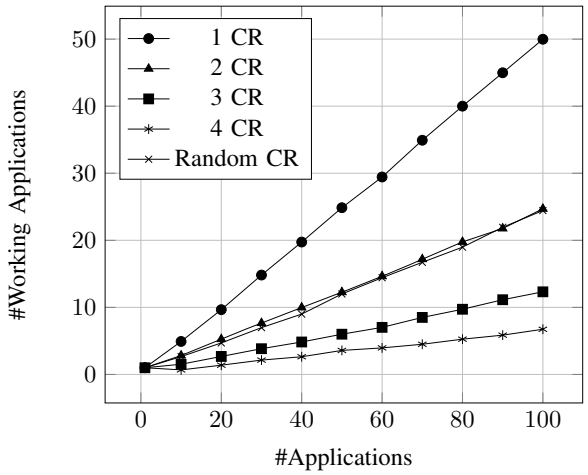

Figure 1: Utility: Context Requirement Variations

With the presence of contradicting context requirements some applications cannot run. Thus, the number of working applications should actually be compared to the number of applications that could run with an optimal resolution in the specific setting. However, the high degree of randomization essentially complicates the calculation of this value which therefore is omitted in this test. Furthermore, the randomness of context requirements might be one reason for the low results. The distributions and their parameters may be another explanation. Hence, we tested the distributions as well as the parameters for influence on the number of working applications. Nevertheless, the tests showed that the overall utility is independent of the value of the standard deviation for the normal distribution and the parameter setting of the beta distribution.

3) Office Scenario: As the random distribution of values leads to non-optimal results, we simulated a more realistic office scenario. In an office, there can be different applications running simultaneously, e.g., people are working, reading, presenting, or phoning. These applications have different predetermined context requirements. This eliminates the randomness of the first test runs. Table II subsumes the different parameters.

We performed 100 simulation runs with 20 applications. The 4 application types were randomly distributed. In average, 3.26 / 5.4 / 10.27 applications of 5 / 10 / 20 applications ran with a maximum of 19 of 20 applications running simultaneously.

Furthermore, we decided to test another implementation of the `cohere()` function. The original implementation is in accordance with the cohesion rule of flocking. We decided to calculate the median instead of the average

Application	Attribute	Values
Working	Light Level [lux] Volume [phon]	200 - 500 5 - 30
Reading/Writing	Light Level [lux] Volume [phon]	300 - 500 10 - 40
Presentation	Light Level [lux] Volume [phon]	150 - 350 30,0 - 50,0
Phoning	Light Level [lux] Volume [phon]	– 5 - 20

Table II: Office Applications

for the intervals. For the simulation, we used the same setting with 20 applications and 100 simulation runs. With the new implementation of the `cohere()` function, in average, 3.61 / 6.41 / 12.49 applications of 5 / 10 / 20 applications ran with a maximum of 19 of 20 applications running simultaneously. When running 20 applications, we achieved an improvement in the average values of 25 %.

The results are illustrated in Figure 2. *Av.* stands for the original implementation of the `cohere()` function (using average values), *Med.* stands for the improved implementation (using median values). The x-axis represents the number of applications, the y-axis the overall utility.

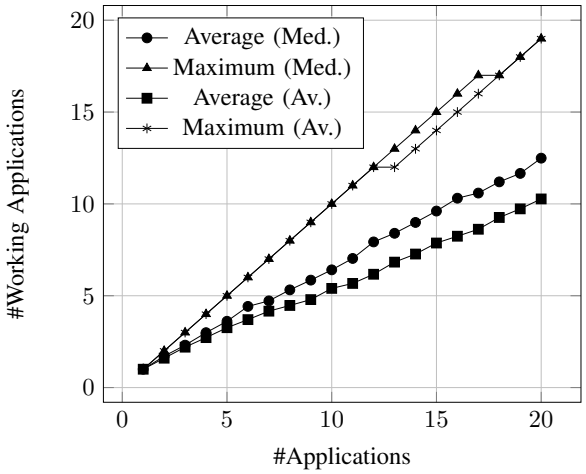

Figure 2: Utility: Office Scenario

E. Discussion

We showed that our approach is suitable for interference management. The interference detection algorithm needs less than 1 millisecond for interference detection in a smart peer group with up to 90 applications. Furthermore, the resolution process is very fast, too. In the smart peer group with 90 applications, the resolution needs less than 2 milliseconds.

Nevertheless, one has to mention that our solution for interference resolution needs further adjustment. So far, in the case of an interference, one application calculates the medium value for all context attributes and sets the context accordingly. Our evaluation showed, that this can lead to problems. Especially if the context requirements of the applications strongly differ, many context attributes are involved, and many applications run simultaneously

in the shared environment, some applications cannot run after the adaptation. Using the median value showed some improvements. Nevertheless, further solutions for resolution should be tested. Due to the modular design of the resolution procedures, developers only need to change the implementation of the `cohere()` function. In other works ([1], [4]), we have shown further approaches for interference resolution, which offer better results in terms of number of applications that can run after context adaptation, but require more computation effort, more time for calculation, and have more sophisticated requirements of context handling by the applications. Nevertheless, we achieved our objective to design and implement a very fast and easy comprehensible solution based on the nature-inspired mechanisms flocking and local inhibition.

VIII. Conclusion and Outlook

In this paper we presented a nature-inspired approach to interference management in smart peer groups. Based on the mechanisms of *flocking* and *local inhibition*, we designed algorithms for interference detection, interference resolution, and leader election for reconfiguration. This algorithms are integrated in a workflow, so that an application starts interference management after *joining* the system and self-optimization of the system is enabled after a change in the set of applications (*leave* and application crash) or the context. We implemented the general designed algorithms in Java. The BASE middleware enabled the finding and communicating between the different applications. Our evaluation showed, that the nature-inspired approach for interference management offers a very fast and easy solution. Nevertheless, the resolution process can be improved to ensure that a higher amount of applications can run after context adaptation.

For future work, we plan to improve the `cohere()` function of the resolution process for achieving a higher amount of applications that can run after a context adaptation. Besides implementing and evaluating additional resolution algorithms, another option could be to work with preferences. This can be done either by offering users the possibility to set application preferences or by developers. Furthermore, the inclusion of resource variables improve the set of adaptation possibilities.

Second, the inhibition phase may be improved by introducing a fitness function. Each application calculates its own maximum random waiting time before the competition phase starts. A fitness function may include the remaining energy and the computational power of a device as well as the number of context requirements of the application.

Third, as we tested our approach on a single computer so far, we plan to test and evaluate the approach in a distributed setting with several nodes.

References

[1] V. Majuntke, S. VanSyckel, D. Schäfer, C. Krupitzer, G. Schiele, and C. Becker, "COMITY: Coordinated Application Adaptation in Multi-Platform Pervasive Systems," in *Proc. PerCom*, 2013.

[2] I. Park, D. Lee, and S. J. Hyun, "A Dynamic Context-Conflict Management Scheme for Group-Aware Ubiquitous Computing Environments," in *Proc. COMPSAC*, 2005, pp. 359–364.

[3] V. Majuntke, G. Schiele, K. Spohrer, M. Handte, and C. Becker, "A Coordination Framework for Pervasive Applications in Multi-User Environments," in *Proc. IE*, 2010, pp. 178 – 184.

[4] S. VanSyckel, D. Schäfer, V. Majuntke, C. Krupitzer, G. Schiele, and C. Becker, "Comity: A framework for adaptation coordination in multi-platform pervasive systems," *Pervasive and Mobile Computing*, vol. 10, Part A, pp. 51–65, 2014.

[5] H. Lee, J. Park, P. Park, M. Jung, and D. Shin, "Dynamic Conflict Detection and Resolution in a Human-Centered Ubiquitous Environment," in *Proc. UAHCI*. Springer, 2007, pp. 132–140.

[6] C. Shin and W. Woo, "Service Conflict Management Framework for Multi-User Inhabited Smart Home," *Journal of Universal Computer Science*, vol. 15, no. 12, pp. 2330–2352, 2009.

[7] H. Jakob, C. Consel, and N. Loriant, "Architecturing Conflict Handling of Pervasive Computing Resources," in *Distributed Applications and Interoperable Systems*, 2011, vol. 6723, pp. 92 – 105.

[8] R. Morla and N. Davies, "A Framework for Describing Interference in Ubiquitous Computing Environments," in *PerCom Workshops*, Pisa, 2006, pp. 635–638.

[9] P. A. Haya, G. Montoro, A. Esquivel, M. Garcia-Herranz, and X. Alaman, "A Mechanism for Solving Conflicts in Ambient Intelligent Environments," *Journal of Universal Computer Science*, vol. 12, no. 3, pp. 284 – 296, 2006.

[10] E. Shaw, "Schooling in fishes: Critique and review," *Development and Evolution of Behavior*, pp. 452–480, 1970.

[11] B. L. Partridge, "The structure and function of fish schools," *Scientific American*, vol. 246, no. 6, pp. 114–123, 1982.

[12] C. W. Reynolds, "Flocks, Herds and Schools: A Distributed Behavioral Model," in *Proc. SIGGRAPH*, 1987, pp. 25–34.

[13] B. A. Kadrovach and G. B. Lamont, "A Particle Swarm Model for Swarm-Based Networked Sensor Systems," in *Proc. SAC*, 2002, pp. 918–924.

[14] H. V. D. Parunak, ""Go to the Ant": Engineering Principles from Natural Multi-Agent Systems," *Annals of Operations Research*, vol. 75, no. 0, pp. 69–101, 1997.

[15] H. Su, X. Wang, and W. Yang, "Flocking in multi-agent systems with multiple virtual leaders," *Asian Journal of Control*, vol. 10, no. 2, pp. 238–245, 2008.

[16] A. E. Turgut, H. Çelikkanat, F. Gökçe, and E. Şahin, "Self-organized flocking in mobile robot swarms," *Swarm Intelligence*, vol. 2, no. 2-4, pp. 97–120, 2008.

[17] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc. Int. Conf. on Neural Networks*, vol. 4, 1995, pp. 1942–1948.

[18] R. Nagpal, "A Catalog of Biologically-Inspired Primitives for Engineering Self-Organization," in *Engineering Self-Organising Systems*. Springer, 2004, vol. 2977, pp. 53–62.

[19] P. A. Lawrence, *The making of a fly: the genetics of animal design*. Blackwell Scientific, 1992.

[20] M. Shackleton, F. Saffre, R. Tateson, E. Bonsma, and C. Roadknight, "Autonomic Computing for Pervasive ICT - A Whole-System Perspective," *BT Technology Journal*, vol. 22, no. 3, pp. 191–199, 2004.

[21] R. Tateson, S. Howard, and R. Bradbeer, "Nature-Inspired Self-Organisation in Wireless Communications Networks," in *Engineering Self-Organising Systems*. Springer, 2004, vol. 2977, pp. 63–74.

[22] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel, "BASE - A Micro-Broker-Based Middleware for Pervasive Computing," in *Proc. PerCom*, 2003, pp. 443–451.