Guided Self-Organisation in Open Distributed Systems

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades
Doktor-Ingenieur (abgekürzt: Dr.-Ing.)
genehmigte Dissertation

von

Jan Kantert (M.Sc.)

geboren am 21.04.1987
in Hannover

2018

1. Referent: Prof. Dr.-Ing. Christian Müller-Schloer
2. Referent: Prof. Dr.-Ing. Bernardo Wagner
Tag der Promotion: 18.12.2017

Zusammenfassung

Selbstorganisierende Systeme wie beispielsweise offene, verteilte Rechen- oder Routingsysteme führen ihre Aufgabe verteilt und ohne zentrale Kontrolle aus. Allerdings sind verteilte Systeme mit autonomen Agenten verwundbar für gezielte oder unbeabsichtigte Angriffe von bösartigen oder defekten Instanzen, da das Verhalten anderer Instanzen nicht vorhersehbar ist. Bestehende Lösungen wie implizite Trust Communities reduzieren diese Unsicherheit, indem sie Vertrauen und Reputation als Konzepte einführen, um Angreifer zu isolieren. Zusätzlich können sich Agenten organisieren und explizite Trust Communities bilden, um Geschwindigkeit und Robustheit zu erhöhen. Jedoch entsteht mit bisherigen Ansätzen häufig negativ emergentes Verhalten, wenn sich die Umgebung verändert oder Störungen auftreten. Während eines so genannten Trust Breakdowns treten mehrere Angreifer dem System bei und verursachen zeitweise Überlast. In diesem Fall nehmen Agenten keine Aufträge von anderen Agenten mehr an, da sie alle Partner als unkooperativ einschätzen und versuchen, sie zu isolieren. Somit verhalten sich nach kurzer Zeit alle Agenten unkooperativ und jeder Agent arbeitet nur noch für sich selbst. Das System erholt sich in der Regel nicht mehr selbstständig von diesem Zustand.

In dieser Arbeit wird eine Methode entwickelt, um Selbstorganisation in offenen, verteilten Systemen zu lenken. Agenten wählen einen Norm Manager (NM), welcher fortan das System überwacht und beeinflusst, indem er Normen erlässt. Der NM beobachtet Interaktionen und Bewertungen zwischen Agenten und konstruiert daraus einen Graphen. Anschließend erstellt er mittels Graphanalysen und Clustering eine abstrahierte Situationsbeschreibung. Wenn die aktuelle Situation vom Ziel abweicht, werden Normen erlassen, um das System zu lenken. Außerdem optimiert der NM das System je nach Situation entweder auf Robustheit oder Geschwindigkeit.

Der entwickelte Mechanismus wird in drei Szenarien evaluiert: Verteilte Sensornetzwerke, offene Grid Computing-Systeme und verteilte Renderingsysteme. Im Sensornetzwerk-Szenario ermöglicht der Ansatz eine Erholung von Angriffen durch bösartige Knoten und erreicht eine Paketzustellungsrate von 95% unter Störung verglichen mit 30% im Referenzexperiment. Allerdings verursacht der Ansatz einen vernachlässigbar geringen Mehraufwand im ungestörten System. Im zweiten Szenario zeigen wir anhand offener Grid Computing-Systeme, dass der NM dauerhafte Bedrohungen wie sich bzgl. ihrer Reputation strategisch verhaltende Agenten erkennen kann. Durch Normänderungen fördert der NM konsistentes Verhalten, um die Angreifer zu isolieren, und das System erholt sich in Folge. Außerdem zeigen wir, dass der NM das System abhängig von der aktuellen Bedrohungslage auf Geschwindigkeit oder Robustheit optimiert. Die Integration von neuen Agenten wird beschleunigt, wenn keine Angriffe beobachtet werden, oder verlangsamt, um Angriffe zu erschweren. Im dritten Szenario zeigen wir anhand verteilter Renderingsysteme, dass unser Ansatz den Stand der Technik übertrifft. Im ungestörten Fall reduziert der NM die Replikation und erhöht den Durchsatz um 90%. Wenn unabhängige Angreifer im System sind, erhöht unser Ansatz den Durchsatz und erreicht eine höhere Korrektheit der Ergebnisse. Kommen konspirierende Angreifer ins System, erreicht der Mechanismus weiterhin höhere Korrektheit und Durchsatz, solange weniger als 50% der Agenten im System Angreifer sind. Erst wenn die Mehrheit der Agenten Angreifer sind, verhält sich die Referenz minimal besser.

Insgesamt ermöglicht unser Ansatz den Entwurf robuster Systeme, welche unter Störungen höhere Leistung erbringen. Im ungestörten Fall impliziert dies einen geringen Mehraufwand, der meist durch Laufzeitoptimierungen kompensiert wird. Zusätzlich schützt der Mechanismus Systeme gegen permanente Angriffe, konspirierende Angreifer und Angriffe auf das Reputationssystem. Wenn negativ emergentes Verhalten auftritt, kann der NM das System durch Normen lenken und eine Erholung herbeiführen.

Schlagworte: Selbstorganisation, Offene verteile Systeme, Multi-Agenten-Systeme

Abstract

Self-organising systems such as Open Distributed Computing or Routing Systems perform their tasks without central control in a distributed manner. However, distributed systems containing autonomous agents are vulnerable to intentional or unintentional attacks by malicious or defective entities because of uncertainty about the behaviour of other entities. Previous solutions such as implicit Trust Communities reduced this uncertainty by introducing trust and reputation to isolate attackers. Furthermore, explicit Trust Communities allow agents to organise to increase robustness and performance. However, due to disturbances or environmental changes, negative emergent behaviour may occur during self-organisation. During a so-called trust breakdown, multiple attackers join the system and cause a temporary overload. Agents will stop accepting jobs from their peers (including the attackers) which in turn will believe that their peers are uncooperative and try to isolate them. As a result, all agents stop to cooperate and every agent is only working on its own. Systems usually do not recover from this state.

In this thesis, a method is introduced to guide self-organisation in Open Distributed Systems. Agents elect a Norm Manager (NM) which monitors the system and guides the system by issuing norms. The NM observes interactions and ratings between agents, performs graph analysis and clustering to gather a high-level situation description. If that situation diverges from the goal function, it will issue norms to guide the system. Furthermore, it can tune the system for either robustness or performance depending on current threats.

The mechanism is evaluated in three application scenarios: distributed sensor networks, open grid computing systems and distributed rendering systems. In the sensor network scenario, the mechanism can recover from attacks by malicious nodes and achieves a packet delivery rate of over 95% during the disturbances compared to 30% in the reference. However, the approach causes a small but neglectable overhead in an undisturbed system. In the open grid computing systems scenario, we show that the NM can identify of permanent threats such as agents which behave strategically regarding their reputation. By changing norms, the NM guides isolation of attackers by enforcing consistent behaviour and, as a result, the system can recover in a distributed manner. Furthermore, we demonstrate that depending on current threats, the NM can tune the system for performance or robustness. Integration of new agents can be facilitated when no attackers are observed or complicated to prevent attacks. In the distributed rendering scenario, we show that our approach outperforms a state-of-the-art solution (BURP). When undisturbed, the NM reduces the replication and thereby improves throughput by about 90%. With independent attackers, our approach always outperforms the throughput of BURP while maintaining better correctness. Furthermore, when colluding attackers join the system, it also outperforms BURP by a large factor in throughput and maintains nearly perfect correctness for up to 50% of attackers. Only when colluding attackers win majority voting (more than 50% of agents are attackers), BURP performs slightly better.

Overall, the approach enables us to build more robust systems which perform better under disturbance. Some overhead occurs when the system is undisturbed but due to runtime optimisations the systems often even achieve higher performance. Furthermore, the approach counters permanent threads, colluding attackers and attacks to the reputation system. In case negative emergent behaviour occurs, the NM guides the system to recovery.

Keywords: Self-Organisation, Open Distributed System, Multi-Agent Systems

Contents

List of Abbreviations

BB	Bad-Behaving agent
BIRCH	Balanced Iterative Reducing and Clustering using Hierarchies
BOINC	Berkeley Open Infrastructure for Network Computing project
BURP	Big and Ugly Rendering Project
CM	Control Mechanism
CPR	Common Pool Resource
DGDS	Dynamic Grouping Distribution Strategy
DGS	Desktop Grid System
DIO	DODAG Information Object
DL	Defeasible Logic
DODAG	Destination-Oriented Directed Acyclic Graph
DODS	Dynamic Ordered Distribution Strategy
DRDS	Dynamic Random Distribution Strategy
DS	Distributed Systems
eTC	explicit Trust Community
ETX	Expected Transmission count
FIFO	First In First Out
HITS	Hypertext-Induced Topic Selection
iTC	implicit Trust Community
MAS	Multi-Agent Systems
MCL	Markov Cluster Algorithm
MTU	Message Transfer Unit
NEB	Negative Emergent Behaviour
NM	Norm Manager
NMAS	Normative Multi-Agent Systems
nTC	normative Trust Community
O/C	Observer/Controller
OC	Organic Computing

OCL	Object Constraint Language
ODCS	Open Distributed Computing System
ODRS	Open Distributed Routing Systems
ODS	Open Distributed Systems
OF	Objective Function
OF0	Objective Function Zero
OGS	Open Grid Systems
PDR	Packet Delivery Rate
RF	Radio Frequency
RPL	Routing Protocol for Low power and Lossy Networks
SES	Simple Exponential Smoothing
SuOC	System under Observation and Control
TARF	Trust Aware Routing Framework
TC	Trust Community
TCM	Trust Community Manager
TDG	Trusted Desktop Grid
TR	Trust Round
TLV	Type-Length-Value
WB	Well-Behaving agent
WSN	Wireless Sensor Network
WU	work unit

List of Figures

List of Tables

1. Introduction

In this chapter, we introduce our system model and application scenarios. Furthermore, we analyse the key challenges and motivate the objectives and success criteria for this thesis.

1.1. Open Distributed Systems and Agents

Distributed Systems (DS) are systems consisting of multiple independent nodes which are represented as a single system to the user [43]. Since components such as shared memory or storage are not universally available between nodes, they have to communicate using messages. Not all kinds of applications perform well on DS [44]. In general, DS belong to one authoritative domain and follow the goal of a single user. Furthermore, all nodes follow the same goal.

Open Distributed Systems (ODS) specifically describe DS consisting of nodes which belong to different authoritative domains. This class of systems is commonly researched in the discipline of Organic Computing (OC) [45]. In contrast to DS, in ODS nodes are autonomous which means that there is no external control over their behaviour. Nodes can, therefore, join or leave the system at any time. We assume at least one distinct user per authoritative domain which sets the goal for the nodes in this domain. Furthermore, nodes can behave incorrectly, intentionally or by error. The technical implementation of a node is up to its user which cannot be punished for malicious behaviour. Generally, nodes will cheat if it is beneficial to reach their goals. As a consequence, there cannot be any assumption of benevolence, and we must consider nodes as rational agents which act on behalf of their user to reach their goal in the best possible way [46]. Since agents act selfishly, high uncertainty about their behaviour exists (when observed externally). Furthermore, agents are considered as black boxes where no introspection is possible.

In ODS, we define a utility function U which measures the utility per node. This value is evaluated locally and ranges from 0 (no utility) to $U_{\max}$ (more is better). The upper limit $U_{\max}$ may be unknown and might depend on the system goal or the environment.

1.1.1. Open Distributed Computing Systems

In Open Distributed Computing System (ODCS), the key requirement for an application running on ODS is that its jobs have to be parallelisable as multiple work units (WUs). In general, there may be dependencies between WUs or jobs. However, in this work, we consider bag-of-tasks applications which contain only independent WUs but the general approach can be extended to include dependencies as well [47].

Reliability is a concern because the probability that at least one component is broken increases with the size of the system. Most traditional algorithms are not designed to deal with node failures. Therefore, jobs are usually separated into multiple idempotent WUs. When all WUs are finished, one of the following approaches is usually applied in DS and ODS (which are both considered in this thesis):

- **Verifying applications** – Results are verified after calculation. This only works with certain applications where validation is significantly cheaper than recalculation.
- **Replicating applications** – Results are calculated multiple times and only accepted when the results match. Since this leads to overhead, the previous approach is usually preferred.

1.1.2. Open Distributed Routing Systems

As another application scenario, Open Distributed Routing Systems (ODRS) consist of multiple nodes which forward data for their peers. Usually, those networks contain a few data sources which are called *root*. The radio range of the autonomous nodes is limited and they decide locally to which peer they forward packets [48–50].

1.2. Challenges of Open Distributed Systems

In this section, we discuss the main challenges when dealing with ODS from a system perspective. A detailed threat model can be found in [C13, 51].

1.2.1. System Openness

The system is considered as open because agents are free to join or leave an ODS at any time. Therefore, we cannot assume that they fear future punishment because they could misbehave and then leave the system before any punishment is enforced. For example, an agent may join the system, exploit it until it is detected, and leave after that. In some application scenarios, an attacker can leave after it was isolated and join with a new identity to regain initial advantages which might be given to new agents.

1.2.2. Black Boxes

In general, the implementation of agents cannot be controlled or introspected. Both the agent and environment executing its code have to be considered as black boxes which can (and will) misbehave, lie about their state and behave selfishly. Therefore, no assumption of benevolence can be made.

1.2.3. Uncertainty and No Assumption of Benevolence

Agents, just like humans, can change their behaviour as a reaction to changing environmental conditions in order to fulfil their goals. This behaviour change leads to uncertainty since it is not predictable. For example, a perfectly behaving agent may start to cheat at any time if this appears advantageous. Similarly, a misbehaving agent may rehabilitate when detected. Furthermore, agents may act as groups and behave differently when dealing with different partners. As a consequence, we cannot assume any benevolence.

1.3. Assumptions

In this work, the following key assumptions are made:

- **System Openness** – Agents can join or leave at any time.
- **Black boxes** – Implementations of agents are unknown and cannot be observed. Also, it is not possible to watch the memory or any internals from the outside.
- **Uncertainty/No assumption of benevolence** – Agents may be broken or malicious. They exploit the system if it aids their goal.
- **Bag-of-Tasks applications** – All work units of one job can be calculated in parallel.
- **Known communication protocol** – All agents know and follow the communication protocol. It is assumed the protocol is deadlock-free and there is a timeout in every state in case an agent leaves the system.
- **Local goal** – Every agent has a local goal. This goal may be unknown to other agents.
- **Global goal** – There is a global goal for an ODS which is usually different from the local agent goals. It is not unusual that local and global goals are contradicting, thus leading to changing/unpredictable agent behaviour.

1.4. Trust and Reputation

As one mechanism to overcome those challenges in ODS, previous works introduced trust and reputation (e.g. [51]). In a first step, nodes rate each other after each interaction and store those experience ratings locally. If a node receives a request from a previously unknown node, it asks trusted peers for their ratings for this particular node and calculates a reputation value for the unknown node. Only cooperative nodes will reach a high reputation, and nodes prefer those over nodes with low reputation. Thereby, attackers are isolated.

Furthermore, after agents made sufficient experience, well-behaving nodes can decide to form a Trust Community (TC) with their trusted peers. Within this organisation, agents agree to cooperate with all other members. If they fail to work for other members of the TC, they will be expelled by the community. Overall, agents gain lower cooperation overhead, higher efficiency and higher robustness when in a TC.

1.4.1. Incentives and Sanctions

To overcome the inherent problems of an open system where no particular behaviour can be assumed, a trust metric was introduced to calculate the reputation. Agents receive ratings for all their actions from their particular interaction partners. This allows others to estimate the future behaviour of a certain agent based on its previous actions. To perform this reasoning, a series of ratings for a certain agent can be accumulated to a single reputation value using the trust metric.

Autonomous agents need to become aware of the expected behaviour in the system. Therefore, we influence the desired actions by norms. These norms are valid for an *Action* in a certain *Context* and, thereby, guide the agents. To enforce the behaviour, they impose a *Sanction* if violated or offer an *Incentive* if fulfilled.

In this scenario, good trust ratings are used as an *Incentive* and, bad trust ratings impose a *Sanction* (agents with higher reputation values have a higher chance to get their work units computed). Based on the norms, agents receive a good rating if they work for other agents and a bad rating if they reject or cancel work requests. As a result, the society isolates malevolent agents and maintains a good system utility in most cases.

1.5. Negative Emergent Behaviour (NEB)

Emergent behaviour as a consequence of self-organised interactions among distributed agents can result in positive and negative effects [C8, W30, W32]. Establishing implicit Trust Communities (iTCs) (see Section 1.7.1) by increasing cooperation with other well-trusted agents and consequently isolating malicious agents to a certain degree is considered as positive emergent behaviour in this context. In contrast, Negative Emergent Behaviour (NEB) typically impacts the overall system performance and consequently needs to be countered. In the following, we describe two scenarios to illustrate the impact of such effects.

1.5.1. Trust Breakdown

As mentioned in Section 1.3, agents are free to join ODS at any time. In case of a potentially large group of malicious agents joining the system simultaneously, we can observe a negative emergent effect with respect to the actual trust level of currently participating agents. For instance, a group of colluding *Freeriders* (see Section 1.6.2 for a definition of this stereotypical behaviour) will load the system with additional work packages while simultaneously rejecting to work for others. Consequently, benevolent agents will also reject work packages issued by the *Freeriders* (following a tit-for-tat concept known from game theory, where an agent first cooperates and then replicates the opponent's previous action [52]). As a result, we can observe numerous bad ratings for both benevolent and uncooperative agents. In sum and on average, the trust levels will drop drastically resulting in a system state where agents do not trust each other any more. We consider such a situation as a disturbed system state; this emergent situation is called *trust breakdown* [53, 54].

Under normal conditions, i.e. when a limited set of *Freeriders* without synchronised and colluding behaviour is part of the system, detection is fast and does not impact the functioning of the trust system. When agents estimate that an interaction partner is uncooperative, i.e. a *Freerider*, they simply stop to cooperate (neither submitting work to nor accepting it from this agent). In contrast, the emergent *trust breakdown* behaviour is the aggregated result of many simultaneous and synchronised interactions – which is not apparent from the single interaction itself.

1.5.2. Overload Situations

A second negative emergent behaviour can be observed in overload situations. In case of high resource demands, agents might distribute significantly more work packages to the system than can be processed in parallel. Consequently, queues fill quickly and agents experience very long waiting times until results are available. Since agents will receive no benefit from participating in the grid any more, they might leave the system and process their work on their own. Hence, we observe negative emergent behaviour that disturbs the grid or even prohibits it from functioning as desired.

A possible strategy to counter such a situation is to restrict the submit ratio, i.e. urging agents to submit less work packages than they process on their own for a certain period. This is assumed to balance the load during operations and therefore avoid peak conditions that motivate benevolent agents to leave the system.

1.6. Application Scenarios

As exemplary ODS, we consider three application scenarios which are introduced in the following. All three application match our assumptions (see Section 1.3) and expose different types of NEB (see Section 1.5). We will use those applications later in the evaluation.

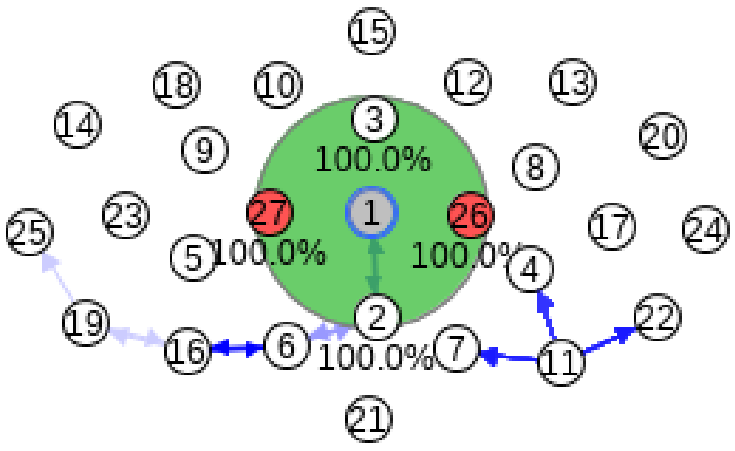
1.6.1. Distributed Low-Power Sensor Networks

Sensor networks consist of numerous distributed nodes which can only communicate locally. Nodes sense locally and send data to a central root node which only few nodes can reach directly because the radio range is limited (i.e., IEEE 802.15.4 [55]). Therefore, data is forwarded by peers to the next node. “The Routing Protocol for lousy and Low-power networks” (RPL; [56]), an IPv6 Routing Protocol, is used to find the best path to the *root* node. “The best path” is defined using an objective function and generally tries to minimise energy consumption (minimal number of transmissions) and to maximise the Packet Delivery Rate (PDR) of data packets. Exemplary applications running on top of sensor networks are described in [48–50].

In Figure 1, we show a sensor network consisting of 27 nodes. If the attacking nodes just failed, RPL would choose the depicted blue paths. However, if they misbehaved, the surrounding nodes would still send them data which would never reach root.

Standard protocols from the domain of mobile ad-hoc networks (i.e., AODV [57] or OLSR [58]) are not applicable due to the limitations of the nodes mentioned above. RPL is a distance vector routing protocol designed for the limitations given by the hardware and the special environmental challenges in Wireless Sensor Networks (WSNs). To route packets, RPL constructs a DODAG. A DODAG is an appropriate routing assumption for WSNs,

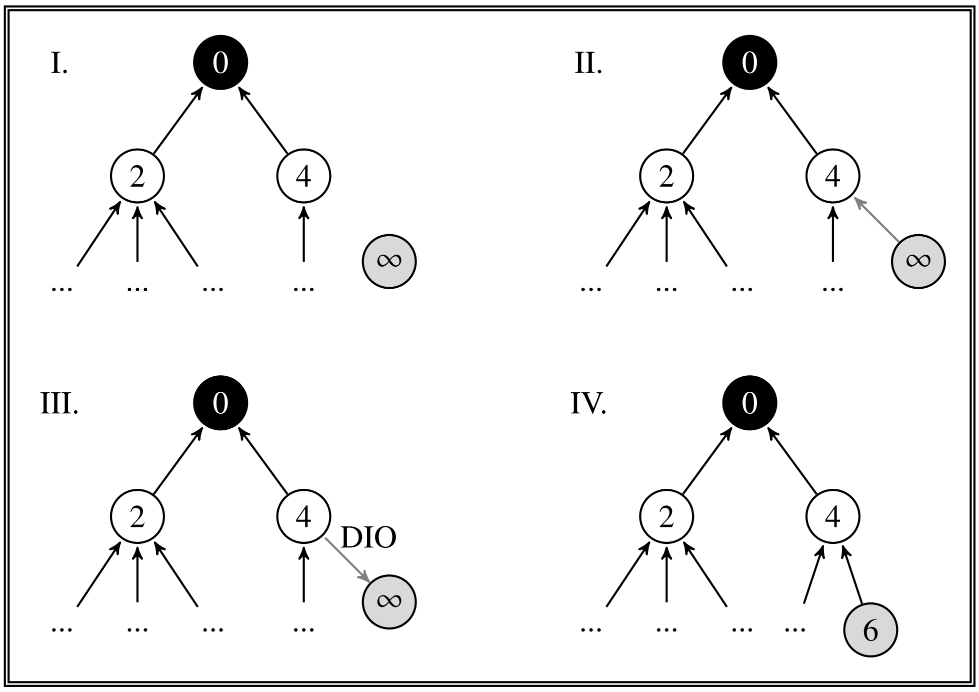
Figure 1: An exemplary sensor network consisting of 27 nodes which includes one root (1; grey) and two attackers (26, 27; red). All other nodes behave well. The radio range of root is shown in green (Nodes 2, 3, 26 and 27 will receive the signal at 100% strength). Other nodes have a similar range. An exemplary routing path is shown in blue.



because most of the traffic is directed to the *sink* node. In RPL, this node is called *root*. A network can contain more than one RPL instance; each instance has its own goal and may consist of more than one DODAG to fulfil this goal. There is only one *root* per DODAG. The process of adding a node to a DODAG is shown in Figure 2. To route a packet to the *root*, all intermediate nodes forward the packet to their parent nodes until the *root* is reached. Nodes periodically communicate their rank by sending DODAG Information Object (DIO) messages. Although most packets are sent to the *sink*, some packets need to be sent from the *sink* to another node. To be able to handle these cases, RPL has the ability to gather all nodes forwarding a packet to the *root*. Using this information, RPL is able to store the routing path into a packet and route it down to the desired node. However, this mode is optional and may be disabled to save memory on the nodes.

Choosing a parent in the DODAG is a task performed by each node on its own. The decision is influenced by different metrics, i.e. the required energy to transmit a packet to the next node. Furthermore, constraints such as the maximum number of forwards can influence the structure of the resulting DODAG. From all this information, the Objective Function (OF) generates the *rank* of a node. The *rank* influences the position of a node in the DODAG. Thereby, the *root* always has a *rank* of 0. Only nodes with a lower *rank* can be parents of a

Figure 2: The process of adding a node to a RPL DODAG. The new node (gray) has an initial rank of unlimited. It sends an announcement message. The neighbouring nodes answer with a DIO containing their rank and routing information. Based on that, the new node will choose its rank. The root node is shown in black on top with rank 0.



node. To forward a packet, a node does not necessarily take the parent with the lowest *rank*. This decision is left to the OF as well. The ability to construct an application-specific routing graph is a big advantage of RPL compared to AODV [57] or OLSR [58].

Due to the openness of distributed sensor networks numerous attacks are possible. Besides failures, nodes can misbehave and act as blackholes or sinkholes. RPL can only cope with failing nodes, and the sender will not notice this failure because acknowledgement packets are not sent in RPL. Even a single malicious node can blackhole most of the network and every other node has to recognize it which will rarely happen. Furthermore, groups of collaborating attackers can sinkhole the complete network without much effort and the network will never recover.

1.6.2. Open Grid Systems (OGS)

OGS consist of numerous heterogeneous nodes which can join or leave at any time. Every node is considered as an agent which acts on behalf of its user and participates in the cooperative network to get its jobs calculated faster. Agents act as submitter to distribute their work and as worker to reciprocally work for other submitting agents. As utility function U, open grid systems use the speed-up σ (1) which captures the gain from distributing work into the grid. τ_{self} is the time which the agent would need to perform the work on its own. Similarly, $\tau_{\text{distributed}}$ is the time needed when distributing the work which is determined by the slowest peer p in the list of parterns P because all work units are distributed at the same time (2).

$$\sigma := \frac{\tau_{\text{self}}}{\tau_{\text{distributed}}} \tag{1}$$

$$\tau_{\text{distributed}} := \max_{p \in P} \left\{ t_{\text{communication},p} + t_{\text{computation},p} \right\} \tag{2}$$

Consequently, agents will leave the system if they do not gain sufficient speed-up from participating (i.e. their speedup falls below 1). We consider the following five stereotypic agent-behaviours in our system:

1. *Adaptive Agents* – These agents are cooperative. They work for other agents who earned high reputation in the system. How high the reputation value has to be generally depends on the estimated current system load and how much the input queue of the agent is filled up.
2. *Freeriders* – Such agents do not work for other agents and reject all work requests. However, they ask other agents to work for them. This increases the overall system load and decreases the utility for well-behaving agents.
3. *Egoists* – These agents only pretend to work for other agents. They accept all work requests but return fake results, which wastes the time of other agents. If results are not validated, this may lead to errors. In any case it lowers the utility of the system.
4. *Cunning Agents* – These agents behave well in the beginning but may change their behaviour later. Periodically, randomly, or under certain conditions they behave like *Freeriders* or *Egoists*. This is hard to detect and may lower the overall system utility.
5. *Altruistic Agents* – Such agents will accept every job. In general, this behaviour is not malicious and increases the system performance. However, it hinders isolation of bad-behaving agents and impacts the system goals because Altruistic Agents will continue to work for attackers even when those are known.

In OGS certain threats arise (as described in Section 1.6.2): Freeriders do not work for others but try to submit their work anyway. Furthermore, Egoists only pretend to work for others and return fake results. Additionally, colluding attackers pretend to work for each other and issue fake ratings. Finally, Cunning Agents change their behaviour over time.

Adaptive Agents are referred to as well-behaving (WB) agents. *Freeriders* and *Egoists* are considered as bad-behaving (BB) agents. *Cunning Agents* alternate between both behaviours and, therefore, cannot be clearly classified. *Altruistic Agents* can be seen as well-behaving, but they also work for bad-behaving agents. Thereby, they decrease the fairness in the system and in that context may also be considered as bad-behaving. Finally, bad-behaving agents might collude to further exploit the system. This means, they implicitly avoid assigning work to each other since they are mutually aware of their exploitation strategy.

To cope with those attackers, we consider the TDG which is an open grid system implementation using trust and reputation as introduced above. For more details see [51]. The TDG supports *Verifying applications* and *Replicating applications*. However, for simulation purposes it can be run with pure virtual applications.

Attacks can happen when malicious agents join the system at startup or during runtime. In this scenario, malicious agents distribute their work, and as a consequence the utility for well-behaving agents decreases since they had no previous experiences with the new agents and are consequently not aware of their malicious behaviour. However, those agents quickly receive bad ratings such that their reputation in the system is reduced. At this point, other agents stop to cooperate with these increasingly isolated agents. Thus, we try to minimise the impact and duration of these disturbances, but they still decrease the system utility [59]. This often leads to NEB such as a *trust breakdown* (see Section 1.5).

1.6.3. Distributed Rendering

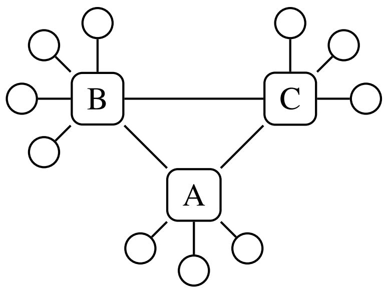


Figure 3: Three studios (A,B,C) with multiple workers

Another application scenario is a distributed rendering application as payload on top of a volunteer-based, self-organised grid system (Figure 3). The file size of an uncompressed film is usually very large. To estimate the required network bandwidth and computing power for distributed rendering, we analysed the raw 4K footage of the motion picture *Big Buck Bunny* from the *Blender Foundation*, which is publicly available. The film has a frame rate of 60 frames per second due to stereoscopic 3D and consists of 79781 frames [60]. Despite the size of the whole film project, each raw frame has a size between 10 and 20 MB, which has to be transmitted to the participants of the network. The workers also need the rendering environment, which differs among every scene in the film. For example, *Big Buck Bunny* is divided into 120 scenes, and the compressed scene rendering environments have a size between 1 MB and 60 MB. However, the submitter can restrict the worker to only render frames from one scene. Therefore, the worker only needs to load this one scene to be able to render all frames from it. This leaves the requirements on bandwidth to be small compared to the computational requirements. These are ranging “from a few hours to several weeks of CPU time” per frame [60]. For comparison: Another animation studio claims, that they require 11.5 hours to render a frame on average for their animated films [61].

Figure 3 shows the concept of multiple film studios which participate in the system and provide multiple workers each. Agents working on behalf of a studio act as a proxy to the internal workers in the worker role. We call them *Proxy Worker*, which behave like a single, fast worker to the grid. In the submitter role, agents can simply send jobs to the *Proxy Worker* without knowing about the internal workers. This scenario runs as *Replicating application* on top of the TDG mentioned in Section 1.1.1.

1.7. Agent Organisations

Agents in ODS can cooperate and form implicit or explicit organisations to defend against attacks. We introduce two types of Trust Communities (TCs) in the following.

1.7.1. Implicit Trust Communities (iTCs)

Implicit Trust Communities (iTCs), as developed by Klejnowski [51], are the simplest form of organisation in ODS. In this section, we present the mechanism and the resulting behaviour. Then, we describe potential application scenarios and limitations which apply to this approach.

Agents locally rate each other after every interaction and calculate the reputation based on the aggregated ratings for all other agents. Therefore, iTCs constitute a fully decentralised solution. Ratings can have values between -1 for a very bad experience and 1 for a very good

Figure 4: A trust graph with agents as nodes and trust relationships as edges. The size of a node corresponds to the reputation of an agent and the thickness of an edge to the trust value. The graph is clustered using a Force Algorithm [62] which pushes nodes apart and contracts by the thickness of edges. Visually, nodes with high reputation form the centre and are considered to be within the iTC. Attackers with low reputation are isolated at the perimeter. Groups of agents are coloured (not relevant here) [O42].

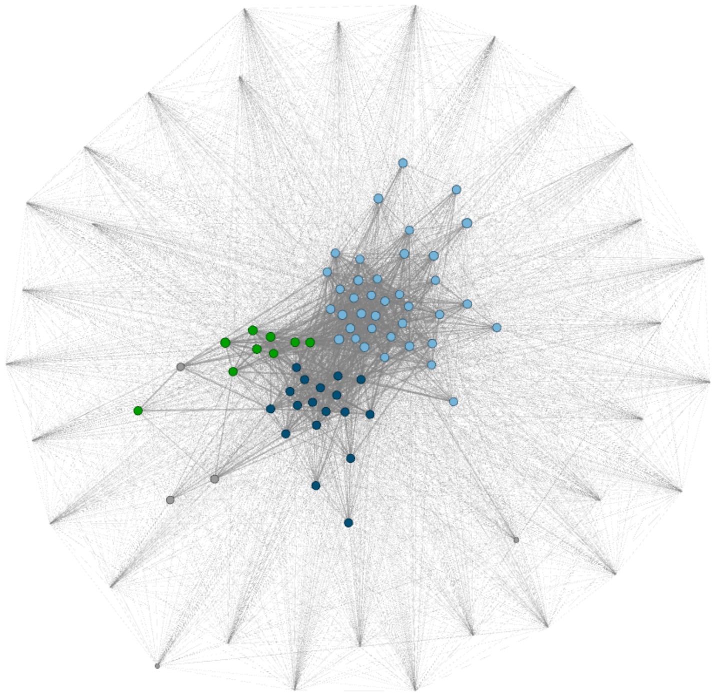
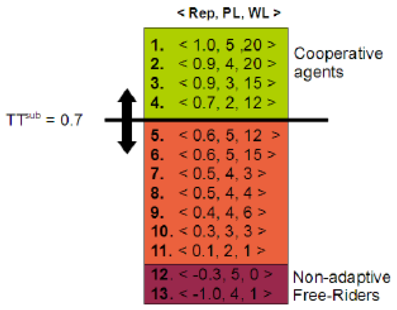


Figure 5: The local trust values for all peers of an agent ranked by their reputation (Rep; PL and WL are not important here). All agents above TT^{sub} are considered as cooperative agents. Agents below TT^{sub} and above zero are undecided and agents below zero are considered untrustworthy. Agents will prefer to work with other cooperative agents [63].



experience. Consequently, the aggregated reputation ranges from -1 for very low to 1 for very high. Agents then sort their peers by reputation and consider all agents above a certain threshold TT^{sub} to be in the iTC. They may also cooperate with agents with a reputation smaller than TT^{sub} but higher than zero if no other cooperation partners can be found. However, they will refuse to work for agents with negative reputation and, thereby, isolate those attackers (see Figure 5). Consequently, agents have a strong incentive to cooperate with WB peers because those peers will reciprocally work for them, too [59].

Using iTCs, various attacking behaviours can be counteracted. In general, all non-cooperative behaviour which results in bad experience ratings can be detected. After an attack by a group of agents the system starts a self-organised recovery process and isolates the attackers once their reputation decreased. This especially includes *Freeriders* and *Egoists* (see Section 1.6.2) because their behaviour directly results in a low reputation. Furthermore, *Cunning Agents* are temporarily detected and isolated.

In Figure 4, we depict an agent community as a graph. Edge thickness represents aggregated trust, and node size corresponds to the reputation of the agent. WB agents form the centre of the graph, and attackers are isolated at the border. In practice, iTCs perform very well in the sensor networks application scenario (see Section 1.6.1). Nodes can only communicate locally with a small number of peers. Therefore, distributed self-organised iTCs isolate attackers without much overhead. Furthermore, iTCs are used in OGS and protect them against most non-colluding attackers.

Since iTCs organise themselves in a fully distributed way, an attacker has to be detected by each agent separately which may take a while. Therefore, recovery speed is low when dealing with larger groups of attackers. Another limitation is posed by collusion attacks where multiple agents agree to exploit the system as a group. Those agents may only pretend to work for each other, assign fake ratings and gain a fake positive reputation. When this attack is organised correctly, it cannot be detected in a decentralised way by an iTC because attackers can permanently generate more good ratings than they receive bad ratings which are given by non-attackers.

1.7.2. Explicit Trust Communities (eTCs)

To overcome some limitations of iTCs, we allow agents to form temporary organisations called explicit Trust Communities (eTCs). They elect a manager which monitors compliance to the rules within the community.

To counteract the limitations of iTCs, we add an explicit membership function in eTCs. Agents within an iTC can decide to form an eTC with trusted peers and ask those peers if they want to join. If a sufficient number of members is found, they elect an temporary Trust Community Manager (TCM) as master which manages membership. Within an eTC, agents can omit additional safety mechanisms such as replication of jobs which decreases the overhead necessary in ODS. Agents agree to work for any other member in the eTC when joining. If an agent misbehaves, other members can complain and the TCM will expel it from the community. During the lifetime of an eTC, the TCM might ask other agents to join if they proved sufficiently trustworthy to maintain an adequate member count. If the eTC becomes too small, it will eventually dissolve because maintaining the TCM would induce too much overhead while creating little value.

In Figure 6, we show the overall system structure of an eTC within an ODS. Agents on the left form an iTC (red and blue) while some agents decide to join the eTC on the right (yellow and orange). The TC members elect a TCM (orange) and prefer to work within the community because of lower overhead. Additionally, in Figure 7, we show a graph from the OGS scenario with four eTCs marked with different colours. Agents prefer to work within the eTCs and maintain a high reputation.

Figure 6: System structure of a system consisting of an iTC on the left (red and blue) and an eTC on the right (yellow and orange). The TCM is depicted in orange and eTC members in yellow. Agents prefer to work within the eTC but are also free to work with agents outside the eTC [51].

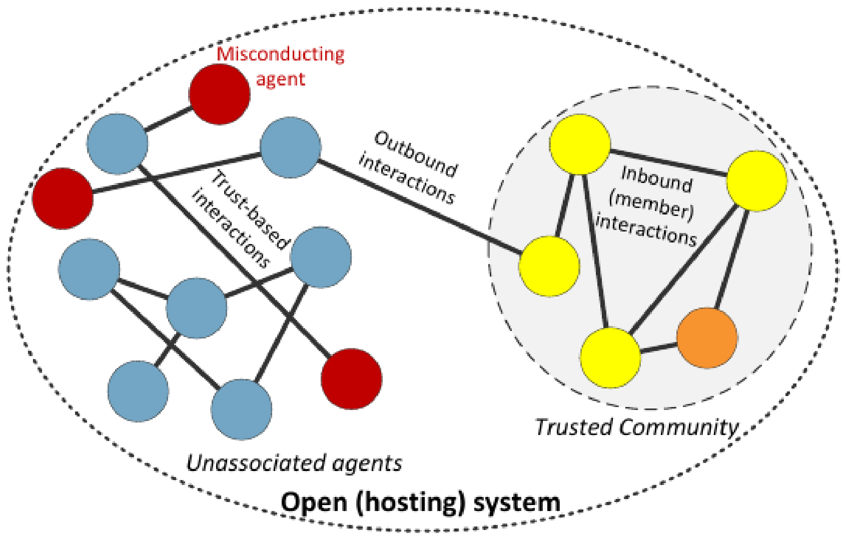
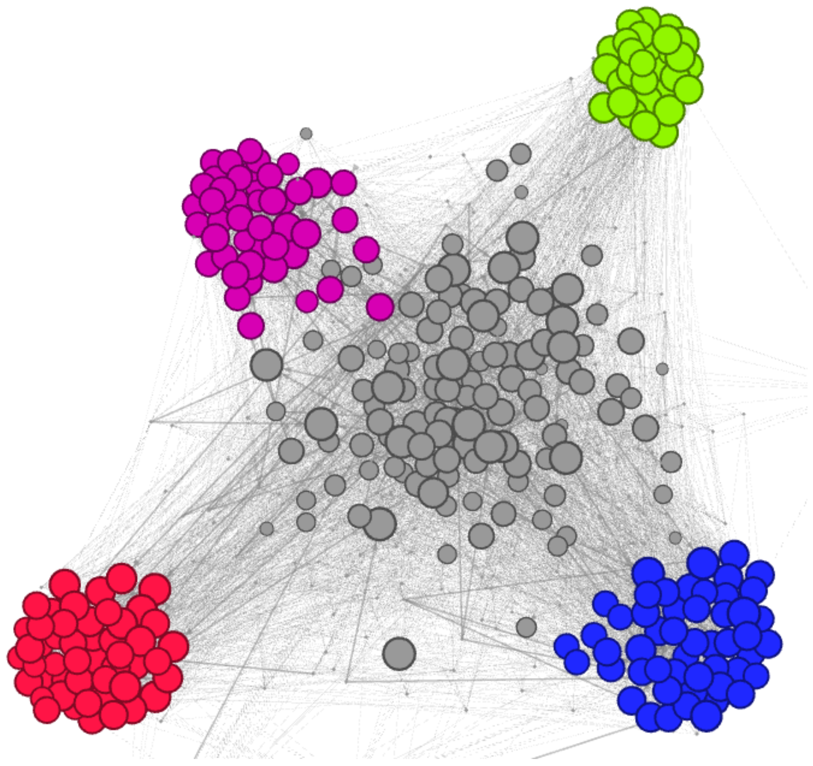


Figure 7: Graph from an experiment in an OGS scenario. The size of a node corresponds to the reputation of an agent and the thickness of an edge to the trust value. The graph is clustered using a Force-Algorithm [62] which pushes nodes apart and contracts by the thickness of edges. Four eTCs have formed (coloured nodes), and an iTC is shown in the center. Attackers are located between the eTCs as very small nodes (gray) with low reputation [O41].



When an attack starts after agents formed an eTC, they are mostly not affected by the attack as long as the attacker is not inside the eTC. If attackers are mostly outside of the eTCs, the impact will be much lower than with only iTCs. Even when attackers are within eTCs, they will be expelled, and the self-organised recovery is much faster. In general, the robustness of the ODS increases and the system can deal with larger attacks. Futhermore, overhead is reduced by skipping safety measures within the eTC. To minimise organisation overhead, the TCM adjusts the member count to be large enough to provide less safety overhead but sufficiently small to keep communication overhead at a minimum.

In the OGS scenario, eTCs provide advantages when large-scale attacks occur or when attackers are colluding to exploit the system (see Section 1.6.2). We do not apply eTCs to sensor networks because their nodes have too few peers to benefit from eTCs.

Generally, an agent society gains robustness and performance from eTCs. However, eTCs can only form in relatively friendly environments. Especially, in the presence of NEB (see

Section 1.5), agent do not mutually trust each others and eTCs cannot form.

Similarly, during collusion attacks, where attackers coordinate and issue fake positive ratings to each other, attackers will be able to join eTCs. As a consequence, those eTCs will usually dissolve quickly. Another limitation surfaces when overload or saturation occurs because agents are no longer able to accept any work from their peers within the eTC.

1.8. Problem Statement

To cope with NEB in ODS, we extract the core objectives for this thesis and define success criteria to measure the effectiveness of different techniques.

1.8.1. Objectives

In ODS, certain situations (such as NEB or a *trust breakdown*) cannot be detected with local knowledge only. However, to maintain a stable and robust system agents need to detect long-term threats (e.g. posed by colluding agents) and take countermeasures as a collective. Therefore, the goal of the work is to develop a mechanism for a collective of agents in an ODS to detect and mitigate NEB. Thus, the mechanism should allow agents to keep their autonomy. Instead, we are searching for new mechanisms which are able to and influence the behaviour of the (semi-autonomous) nodes through indirect guidance only because agents cannot be forced to perform an action in open systems. We want to use norms as a basic mechanism for this purpose.

More specifically, the overall goal is to maximise the utility of the well-behaving agents in the system and the system in general. This especially applies during and after disturbances. However, the cost of the mechanism (in terms of utility) should also be minimal in case of no disturbances. Furthermore, we aim to generally increase the active robustness of the system by introducing active mechanisms which are only activated when a certain disturbance occurs and was detected. Long-term or permanent disturbances are particularly interesting because recovering from those will largely benefit our main goal.

1.8.2. Success Criteria/Metrics

To measure success we observe the utility of a system which is application-specific. Generally, the overall goal should be to maximise the utility. When disturbances do not occur, reasonable optimisation should be performed and overhead for security measurements should be reduced.

In general, we want to see only minimal (or no) overhead (measured in terms of utility decrease) when the system is operating normally.

When a disturbance occurs, we expect the system to detect this event and start an active recovery mechanism if needed. Later, it should disable the mechanism if no longer necessary. Thereby, we should observe a smaller utility degradation when compared to a reference experiment without the developed mechanism. Furthermore, if no suitable active recovery mechanism is available we expect the utility to not become worse.

Specifically, we expect the system to recover from permanent attacks which are not observable from a local perspective of an agent. Those attacks, which can be detected from the system-level only, should be identified and security measurements should be enabled. This includes special tailored attacks to the reputation system and secondary effects such as NEB. Colluding attackers may attack the reputation system by assigning fake ratings and trigger NEB when successful. Ideally, we can prevent the attack to the reputation system. However, if that succeeds, we should be still able to recover from the resulting *trust breakdown*.

To summarise this section, we aim at:

- **Utility Maximisation** – Utility should be maximal when no attacks occur. Thereby, the overhead introduced by the measures should be minimal when no attacks occur and optimisations should be triggered.
- **Fast Recovery from Attacks** – When attacks occur, they should be detected and security measurements should be enabled. This should accelerate recovery from such events.
- **Detection and Mitigation of Permanent Threats** – The mechanism should be able to detect and mitigate permanent threats without much overhead when those do not appear.
- **Detection and Recovery from Attacks to the Reputation System** – Attacks to the reputation system are generally possible but should be detected and mitigated.
- **Recovery from NEB** – The system should be able to recover from NEB by encouraging cooperation and, if necessary, relaxing constraints.

1.9. Outline of this Thesis

This thesis is structured as follows: In Chapter 1, we motivated the challenge of ODS and introduced our three application scenarios. Additionally, we defined the objectives and success criteria. Afterwards, in Chapter 2, we investigate related work and discuss previous work. We introduce previous work from the area of trust and reputation, ODS and normative multi agent systems. Furthermore, we review approaches from all application scenarios and rate them according to our criteria. Then, we present our approach to introduce a higher-level Norm Manager in Chapter 3 which observes and guides the system during runtime. First, we introduce the observer part which monitors the network structure, communication and trust relationships. Based on this it builds and analyses graphs to assess agent behaviour and find groups of colluding agents. Afterwards, the controller compares the system state to thresholds and changes norms to defend or optimise the system. We evaluate our approach in all three application scenarios in Chapter 4 and show that it outperforms state-of-the-art systems. Thereafter, in Chapter 5, we discuss the results regarding our success criteria and goals. Finally, we conclude and give an overview about future work.

2. Previous and Related Work

In this section, we discuss previous and related work. First, we describe the state-of-the art for the techniques used. Afterwards, we analyse if related approaches could be used to solve our challenge. Finally, we present an overview and show that the state-of-the-art systems do not meet our criteria.

2.1. Previous Work

Our approach uses or extends the following techniques. However, on their own they are not suited to solve our challenge.

2.1.1. Open Grid Computing Systems

OGS are used to share resources between multiple administrative authorities. The *ShareGrid* Project in Northern Italy is an example for a peer-to-peer-based system [64]. A second approach is the Organic Grid, which is peer-to-peer-based, but with decentralised scheduling [65]. Compared to our approach, these approaches assume that there are no malicious parties involved and each node behaves well. Another implementation with a central tracker is the Berkeley Open Infrastructure for Network Computing project (BOINC) [66].

All those systems solve a distributed resource allocation problem. Since work units can be computed faster when agents cooperate, they reward and, thus, maximise cooperation. Additionally, a high fairness value ensures equal resource distribution (cf. [67–69]).

We model our grid nodes as agents. Agents follow a local goal which differs from the global system goal [70]. We consider agents as black boxes which means that we cannot observe their internal state. Thus, their actions and behaviour cannot be predicted [71]. Our Trusted Desktop Grid supports Bag-of-Tasks applications [72].

A classification of Desktop Grid Systems (DGSs) can be found in [73]. A taxonomy can be found in [74]. It is emphasised there that there has to be some mechanism to detect failures and malicious behaviour in large-scale systems. Nodes cannot be expected to be unselfish and well-behaving.

2.1.2. Trust and Reputation

In contrast to other state-of-the-art works, we do not assume the benevolence of the agents [46]. To cope with this information uncertainty, we introduced a trust metric. Trust has been discussed in the domain of philosophy [75], psychology [76] or sociology [77]. A review about computational trust can be found in [78]. A general overview about trust in Multi-Agent Systems can be found in [53].

Ramchurn et al. [79] described an early trust metric. Another implementation of trust in a DGS was evaluated in [80]. More recently, Klejnowski [51] proposed a trust metric and reputation system for ODS. However, the metric has some short-coming when rational agents use it for reasoning. In some cases, agents will not perform desired actions because it would decrease their reputation. We will extend this metric in Section 3.2.

2.1.3. Norms

Norms are similar to laws and can be expressed in deontic logic and argumentation. Individuals can reason based on these norms. Since there are multiple actions available, they may use additional factors or preferences [81]. Other approaches use Defeasible Logic (DL) to efficiently model [82] and reason [83] about norms. They separate facts and rules, which can be strict rules, defeasible rules, and exceptions from defeasible rules (called defeaters). To resolve conflicts between two rules reasoning about the same action, priorities can be specified [84]. All reasoning can be performed in linear time and is stable even when norms are not consistent [85].

We base our norm format on Urzică and Gratie [86]. The authors developed a model for representing norms using context-aware policies with sanctions. They consider reputation when making decisions based on norms. We use a conditional norm structure as described in Balke et al. [87]. Most of our norms can be characterised as “prescriptions” based on [88], because they regulate actions. Our norms are generated by a centrally elected component representing all agents which classifies them as an “r-norm” according to Tuomela and Bonnevier-Tuomela [89]. By using norms, our agents can reach agreements and express commitments [89]. However, the agents can still violate such commitments and risk a sanction. Thereby, the agents stay autonomous. In Hollander and Wu [90], the authors present a norm lifecycle including norm creation, enforcement, and adaption which is revisited in our approach.

2.1.4. Normative Multi-Agent Systems

This work is part of wider research in the area of norms in Multi-Agent Systems (MAS). However, we focus more on improving system performance by using norms than researching the characteristics of norms [91]. Normative Multi-Agent Systems (NMAS) are used in multiple fields: e.g. Governatori and Rotolo [92] focus on so-called policy-based intentions in the domain of business process design. Agents plan consecutive actions based on obligations, intentions, beliefs, and desires. Based on DL, social agents reason about norms and intentions.

In Artikis and Pitt [93], the authors present a generic approach to form organisations using norms. They assign a role to agents in a normative system. This system defines a goal, a process to reach the goal, required skills, and policies constraining the process. Agents directly or indirectly commit to certain actions using a predefined protocol. Agents may join or form an organisation with additional rules.

The *normchange* definition [94] describes attributes, which are required for NMAS. Ten guidelines for implementation of norms to MAS are given. We follow those rules in our system.

When norms are involved, agents need to make decisions based on these norms. Conte et al. [95] argue that agents have to be able to violate norms to maintain autonomy. However, the utility of certain actions may be lower due to sanctions.

According to Savarimuthu and Cranefield [96], NMAS can be divided into five categories: norm creation, norm identification, norm spreading, norm enforcement, and network topology. We use a leadership mechanism for norm creation and norm spreading. For norm identification, we use data mining and machine learning. For norm enforcement, we use sanctioning and reputation. Our network topology is static.

Related approaches use NMAS for governance or task delegation in distributed systems [97]. However, we focus on improving technical systems and their approach is not directly applicable.

2.1.5. Tragedy of the Commons

Our scenario is similar to management of Common Pool Resources (CPRs). According to game theory, this leads to a *tragedy of the commons* [98]. However, Ostrom [99] observed cases where this did not occur. She presented eight design principles for successful self-management of decentralised Enduring Institutions. Pitt et al. [100] adapted these to NMAS:

1. *Clearly defined boundaries: those who have rights or entitlement to appropriate resources from the CPR are clearly defined, as are its boundaries.* This means that ownership for all resources must be clearly defined.
2. *Congruence between appropriation and provision rules and the state of the prevailing local environment.* Norms or rules should discourage over-usage by imposing sanctions and encourage cooperation/sharing of those resources.
3. *Collective choice arrangements: in particular, those affected by the operational rules participate in the selection and modification of those rules.* Agents participating in the system should have an influence on the rules.
4. *Monitoring, of both state conditions and appropriator behaviour, is by appointed agencies, who are either accountable to the resource appropriators or are appropriators themselves.* The monitoring instance should be elected/selected by the users of the CPR which could be one of the users.
5. *A flexible scale of graduated sanctions for resource appropriators who violate communal rules.* Sanctions should be gradual and users should not instantly be expelled (or death sentenced) to maintain a healthy community.
6. *Access to fast, cheap conflict resolution mechanisms.* In case of conflicts there should be a fast and cheap resolution mechanism. The mechanism can either decide about sanctions or change rules.
7. *Existence of and control over their own institutions is not challenged by external authorities.* Outside institutions should fully delegate the authority for the CPRs and not intervene.
8. *Systems of systems: layered or encapsulated CPRs, with local CPRs at the base level.* Hierarchical systems or institutions should be used and authority should be delegated to reduce overhead.

Klejnowski [51] implemented most of those principles in the previously mentioned TDG scenario with eTCs. However, *Collective choice arrangements*, *Monitoring* and *Graduated Sanctions* are very limited. In this dissertation, we will extend those implementations.

2.1.6. Robustness

The term *robustness* in literature is widely used with different meanings, mostly depending on the particular context or underlying research initiative. Typical definitions include the ability of a system to maintain its functionality even in the presence of changes in their internal structure or external environment (sometimes also called *resilient* or *dependable* systems) [101], or the degree to which a system is insensitive to effects that have not been explicitly considered in the design [102].

Especially when considering engineering of information and communication technology-driven solutions, the term "robust" typically refers to a basic design concept that allows the system to function correctly (or, at the very minimum, not failing completely) under a large range of conditions or disturbances. This also includes dealing with manufacturing tolerances. Due to this wide scope of related work, the corresponding literature is immense, which is e.g. expressed by detailed reports that range back to the 1990s [103]. For instance, in the context of scheduling systems, robustness of a schedule refers to its capability to be executable – and leading to satisfying results despite changes in the environment conditions [104]. In contrast, the systems we are interested in (i.e. self-adaptive and self-organising OC systems [45]) typically define robustness in terms of fault tolerance (e.g. [105]).

In computer networks, robustness is often used as a concept to describe how well a set of service level agreements are satisfied. For instance, Menasce et al. explicitly investigate the robustness of a web server controller in terms of workloads that exhibit some sort of high variability in their intensity and/or service demands at the different resources [106].

More specifically and beyond these general definitions of the notion of robustness, we shift the focus towards quantification attempts. Only a few approaches known in literature aim at a generalised method to quantify robustness; in a majority of cases, self-organised systems are either shown to perform better (i.e. achieve a better system-inherent utility function) or react better in specific cases (or in the presence of certain disturbances), see e.g. [107, 108]. In the following, we discuss the most important approaches to robustness quantification.

In the context of OC, a first concept for a classification method has been presented in Schmeck et al. [109]. Here, the idea is (as in our approach) to take the system utility into account. Based on a pre-defined separation of different classes of goal achievement (i.e. distinguishing between target, acceptance, survival, and dead spaces in a state space model of the system S), the corresponding states are assigned to different degrees of robustness. Consequently, different systems are either strongly robust (i.e. not leaving the target space), robust (i.e. not leaving the acceptance space), or weakly robust (i.e. returning from the survival space in a defined interval). In contrast to our method, a quantitative comparison is not possible. In particular, this robustness classification does not take the recovery time into account.

Closely related is the approach by Nafz et al. presented in [110], where the internal self-adaptation mechanism of each element in a superior self-organising system has the goal to keep the element's behaviour within a pre-defined corridor of acceptable states. Using this formal idea, robustness can be estimated by the resulting goal violations at runtime. Given that system elements have to obey the same corridors, this would also result in a comparable metric. Recently, the underlying concept has been taken up again to develop a generalised approach for testing self-organised systems, where the behaviour of the system under test has to be expressable quantitatively [111]. However, this depends on the underlying state variables and invariants that are considered – which might be more difficult to assess at application level (compared to considering the utility function in our approach).

In contrast, Holzer et al. consider a node in a network as a stochastic automaton and model the nodes' configuration as a random variable [112]. Based on this approach, they compute the level of resilience (the term is used similarly to robustness in the following) depending on the network's correct functioning in the presence of malfunctioning nodes that are again modelled as stochastic automatons [113]. In contrast to our approach, this does not result in a comparable metric and limits the scope of applicability due to the underlying modelling technique.

From a multi-agent perspective, Di Marzo Serugendo et al. approached a quantification of robustness using the accessible system properties [114]. In general, properties are assumed to consist of invariants, robustness attributes, and dependability attributes. By counting or estimating the configuration variability of the robustness attributes, systems can be compared with respect to robustness. Albeit the authors discuss an interesting general idea, a detailed metric is still open.

Also in the context of MAS, Nimis and Lockemann presented an approach based on transactions [115]. They model a MAS as a layered architecture (i.e. seven layers from communication at the bottom layer to the user at the top layer). The key idea of their approach is to treat agent conversations as distributed transactions. The system is then assumed to be robust, if guarantees regarding these transactions can be formulated. This requires technical prerequisites, i.e. that the states of the participating agents and the environment must be stored in a database – this serves as basis for formulating the guarantees. Obviously, such a concept assumes hard requirements that are seldom fulfilled, especially not in open, distributed systems where system elements are not under control of a centralised element and might behave unpredictably.

2.2. Related Approaches

In this section, we analyse state-of-the-art approaches according to our criteria (see Section 1.8.2). We structure this section along our application scenarios and discuss related solutions. Afterwards, we present a matrix which rates all techniques according to our criteria.

2.2.1. Low-Power Sensor Networks

Low-power sensor networks (as described in Section 1.6.1) try to optimise two metrics: PDR should be maximal as primary utility and power consumption as secondary utility (which is measured as the number of transmission) should be minimal. RPL [56] achieves this goal for undisturbed networks. It can handle node failures and will route around them. However, it has no means to handle intentional or unintentional misbehaviour. Therefore, attackers can easily create a sinkhole to drop the complete traffic of the network.

Ganeriwal et al. [116] implemented reputation in sensor networks. Every node rates its neighbours and malfunctioning nodes can be avoided. This allows identifying simple misbehaviour and isolating those attackers. However, relay or wormhole attacks cannot be prevented. Furthermore, colluding attackers might easily trick the system and could probably cause NEB which cannot be covered. Also, attacks to the reputation system are possible by colluding attackers.

Another approach was presented by Leligou et al. [117]: They propose to use active traffic sniffing to defend against attacks and integrate encryption to fight most attacks. However, active sniffing costs a lot of energy, because the CPU has to read all packets send over the air. Additionally, this approach only ensures that packets are forwarded. It cannot assure that the next hop actually exists and, thereby, those packets may not be delivered to the *root*.

Zhan et al. [118] presented a Trust Aware Routing Framework (TARF), which can fight most attacks without much overhead. They use a very simple trust metric, but no reputation. However, their routing is only partly energy-aware and mostly optimises for PDR. Therefore, after temporary disturbances, TARF will stick with suboptimal routes. Furthermore, permanent threats such as periodic attacks cannot be successfully mitigated.

2.2.2. Open Grid Systems

OGS try to optimise either throughput or speedup. Traditional cluster managers (such as TORQUE [119]) only consider closed systems in a single administrative domain. TORQUE will optimise the throughput or speedup (depending on job settings) and can handle simple node failures. However, misbehaving nodes can easily trick the system and will not be detected or become isolated. Furthermore, the scheduler is a single point of failure and does not scale for large distributed systems. Therefore, TORQUE is not usable in open distributed systems. HTCondor [120] is a scheduler for open grid systems which is decentralised. However, HTCondor assumes a trust relation between users and the workers. Therefore, all kinds of attacks by the worker are possible because HTCondor does not try to mitigate those. Similarly, JoSchKa [121] is another approach for desktop grid systems. It runs a central tracker which distributes work and schedules retries on failed nodes. However, it does not perform any validation of results and can be attacked easily. Additionally, the tracker poses a single-point of failure.

The Trusted Desktop Grid (TDG) [51] is a system which implements trust and reputation (as described in Section 1.6.2). It supports multiple types of applications which can be either verifying or replicating. Agents within the system can decide to join an explicit Trust Community (as described in Section 1.7.2), which shields them from certain attacks. The TDG is robust against attacks once eTCs have formed and can even recover from a limited amount of NEB it that case. However, when the system is constantly under disturbance, no eTCs will form. In this case, it cannot recover from NEB and shows poor robustness.

Another approach is the Berkeley Open Infrastructure for Network Computing project (BOINC) [66, 122] which aims to distribute work among volunteers running a client on their desktop. They specifically expect users to cheat and, therefore, implement replication with a fixed replication factor and majority voting. Thereby, there is always a large overhead because of duplicate processing. Additionally, colluding attackers might also trick this replication by returning the same fake results. Famous applications running on top of BOINC are SETI@home [123] or Einstein@home [124]. The Big and Ugly Rending Project (BURP [125]) performs distributed rendering (see Section 1.6.3) on top of BOINC. In practical cases, they claim that they had to reprocess significant portions of the films because of incorrect results (see [60]).

2.2.3. Comparison

In Figure 8, we list all related approaches and rate them according to our criteria (see Section 1.8.2). None of the approaches fulfils all of them. The TDG is the most promising approach since it matches the criteria best. Thus, we decided to extend it to fit our requirements in the next chapter.

Figure 8: Comparison of related work according to our criteria.. ✓= criterion fulfilled, (✓)= partially fulfilled, ✗= not fulfilled or does not apply (i.e. no reputation system exists)

	Utility Maximisation	Fast Recovery from Attacks	Detection and Mitigation of Permanent Threats	Detection and Recovery from Attacks to the Reputation System	Recovery from NEB
RPL [56]	✓	✗	✗	✗	✗
Ganeriwal et al. [116]	✓	(✓)	(✓)	✗	✗
Leligou et al. [117]	✗	(✓)	(✓)	✗	✗
TARF [118]	✓	✓	(✓)	✗	✗
TORQUE [119]	✓	✗	✗	✗	✗
TDG [51]	✓	✓	(✓)	(✓)	(✓)
HTCondor [120]	✓	✗	✗	✗	✗
JoSchKa [121]	✓	✗	✗	✗	✗
BOINC [66, 122]	✗	(✓)	(✓)	✗	✗

3. Approach

In this chapter, we present our solution concept in more detail. We introduce details of the trust/reputation system and our norm implementation. Then, we describe our higher-level Norm Manager (NM) and its components.

3.1. Solution Concept

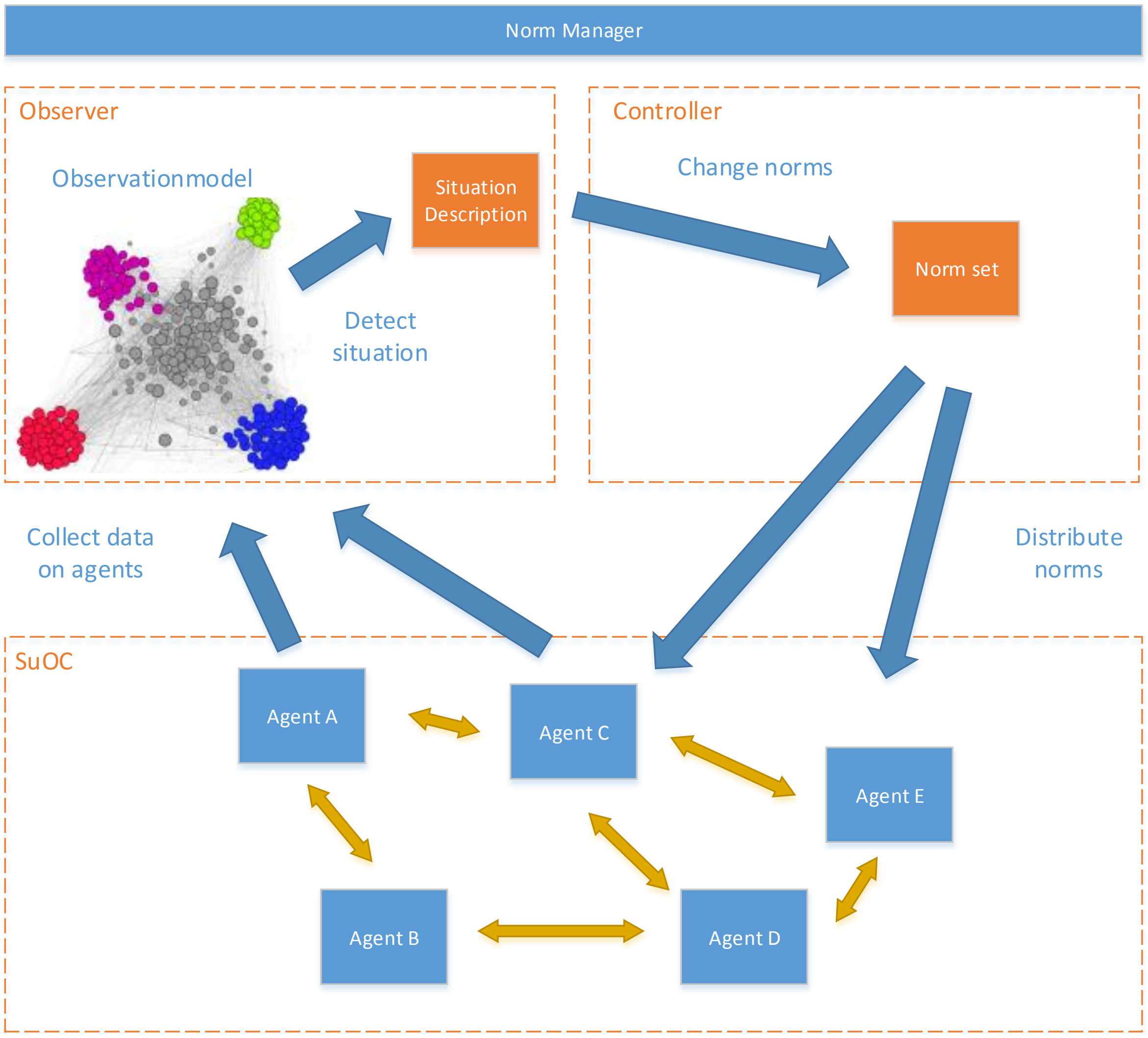
To detect NEB and permanent threats in ODS, we propose a higher-level Observer-Controller (O/C; see [45]) which guides the system out of the situation. Those situations are usually not locally observable from an agent's perspective. Therefore, the observer constantly observes interactions in the system to detect NEB such as a *trust breakdown* or colluding attacks. When such a situation is recognised, the controller will take actions to guide the system out if it. However, because of the openness in ODS, the controller cannot rely on compliance when it commands a black box agent to perform a certain action. Nevertheless, it can issue norms which are enforced in a distributed manner by the agents as long as the majority adheres to them.

In Figure 9, we show the overall solution concept of the higher-level O/C. The System under Observation and Control (SuOC) on the bottom consists of autonomous agents which interact. Most agents cooperate but some also might participate in an attack. The community elects a higher-level O/C which observes the interactions between agents. Based on communication patterns and statistics, it assesses the current situation and creates a situation description. If NEB or permanent attacks are detected, the observer issues or changes norms to guide the SuOC out of the situation. Norms are enforced among the agents in a distributed manner.

In the remainder of this chapter, we present the solution concept: First, we introduce trust, reputation and norms into the SuOC to allow autonomous but guided decision making. Norms are used to define current desirable behaviour. Outside of attacks, they guide agents to optimal global system performance and they discourage actions which might be harmful during NEB. To enforce norms in a distributed manner, agents rate transactions using a trust metric and use reputation to choose cooperation partners.

Afterwards, we present an approach to monitor an ODS which consists of black box agents by observing communication and utility. We introduce a formal robustness definition to assess the global utility for the system before, during, and after disturbances. To evaluate the behaviour inside the system, we gather observable information from the reputation system,

Figure 9: A System under Observation and Control (SuOC) consisting of autonomous agents is observed by the Norm Manager. The observer aggregates externally observable information about the system and passes a situation description to the controller which changes norms accordingly.



sample ongoing communication and represent them as graphs. Using those graphs, we present a solution to estimate the level of self-organisation in the system which is used in the O/C to determine if a disturbance is happening. Furthermore, the observer applies clustering to find similar-behaving and cooperating agents. The former ones are used to determine the composition of the system which is exploited later when choosing norm changes in the controller. Additionally, this information can be used to detect non-colluding attackers. The latter identifies groups of agents which are then rated to decide if they are groups of colluding attackers or groups of cooperating agents (e.g. eTCs; see [51]).

Finally, we focus on the controller part of the O/C which selects norms when an attack or disturbance was detected and quantified by the observer. In general, the controller can increase sanctions to discourage actions or intensify incentives to encourage certain other actions. We analyse which strategy is preferable and what the limits of both approaches are. Furthermore, we discuss the trade-off between performance and robustness which occurs in most situations and elaborate how and when agents are limited in their autonomy when using this approach.

3.2. Trust and Reputation

To overcome problems of ODS where no particular behaviour can be assumed, we introduce a trust metric. Agents receive ratings for all their actions from their particular interaction partners. This allows others to estimate the future behaviour of a certain agent based on its previous actions. To perform this reasoning, a series of ratings for a certain agent can be accumulated to a single reputation value using the trust metric.

To use trust in ODS, the represented ratings should have a well-defined semantic. In general, a rating r is either good or bad. Neutral ratings have no impact and, therefore, can be omitted. In the following, we consider a rating with a value above/below 0 as good/bad rating and define r_+/r_- as a set of all good/bad ratings (4, 5). In all cases, $\mathfrak{T}(R) = 0$ is considered as a neutral reputation.

An obvious implementation would be to store all ratings r which are received by an agent in a First In First Out (FIFO) queue called R_n (with n as the generation), containing up to k elements (R_0 will be empty in the beginning; as proposed by [51]). The trust metric $\mathfrak{T}(R)$ calculates the *reputation* τ based on R_n (3). In the FIFO queue R_n, recent ratings are in the end of the queue and move to the front when more ratings are added. Formally, a function

$\mathfrak{A}(R_n)$ is defined to add an entry r to R_n (6).

$$r \in \mathfrak{R}, R_n \in \mathfrak{R}^k, n, k \in \mathbb{N} \tag{3}$$
$$\mathfrak{R}_+ := \{r : r > 0 \wedge r \in \mathfrak{R}\} \tag{4}$$
$$\mathfrak{R}_- := \{r : r < 0 \wedge r \in \mathfrak{R}\} \tag{5}$$
$$R_{n+1} := \mathfrak{A}(R_n, r) \tag{6}$$
$$\mathfrak{T} : \mathfrak{R}^k \rightarrow [-1, 1] \tag{7}$$
$$\tau := \mathfrak{T}(R_n) \tag{8}$$
$$\tag{9}$$

If one agent performs an action A which is preferred, it receives a more positive rating. E.g., A may have a value of 0.9 while a less desirable action B may get a value of 0.45 assigned. If an agent performed some unwanted action C with a value of -0.9, it would either need to perform A once or B twice to gain a neutral reputation of 0.

Unfortunately, this only works for combinations of negative and positive ratings. Intuitively, we would assume that an agent may gain the maximal reputation by either performing a few very desirable actions or several positive, but less desirable actions. However, this is not the case for $\mathfrak{T}^c(R_n^c)$:

$$R_1^c := (1, 1, 1) \tag{10}$$
$$\mathfrak{T}^c(R_1^c) := \frac{1 + 1 + 1}{3} = 1 \tag{11}$$
$$R_2^c := (1, 1, 1, 0.5, 0.5, 0.5) \tag{12}$$
$$\mathfrak{T}^c(R_2^c) := \frac{3 \cdot 1 + 3 \cdot 0.5}{6} = 0.75 \tag{13}$$

In R_1^c (10), we consider an agent which received three perfect ratings. In R_2^c (12), the same agent received three additional medium positive ratings which were added using Equation (6) (considering $k \geq 6$). As expected, R_1^c results in a perfect reputation $\mathfrak{T}^c$ of 1 (11). We would assume that this is also valid for R_2^c. Additional positive ratings should not decrease the reputation. Unfortunately, this is happening with $\mathfrak{T}^c$ and the reputation decreases to 0.75 (13).

Thus, we consider a rational agent which gained a perfect reputation of 1. Since it behaves rationally, it will not select any action which will decrease its reputation. Unfortunately, with the previous metric $\mathfrak{T}^c$, it can no longer select any action with a rating other than 1. However, all ratings with a value higher than 0 are considered as good and, thus, desirable. Therefore, we require a metric which will not decrease the reputation if an agent receives a good rating.

To allow fully rational agents, a representation of trust should be able to handle ratings of different intensities. I.e., if an agent has to choose from two possible desirable actions where one is favoured, we want to represent this fact in the resulting trust ratings r_1 and r_2. If r_1 is more favourable than r_2, we expect the resulting reputation to increase more after receiving r_1 than after r_2 unless the reputation is already maximised (R1). The same should symmetrically apply to negative ratings. Additionally, we expect the reputation to increase for every positive rating until it reaches its maximum (R2). Again, the exact opposite should apply for negative ratings.

Our main motivation for R2 is a scenario where an agent can mandate to perform a desirable action which has a low priority (e.g., it is not required to do something). Based on previous metrics, if the agent already has a high reputation (i.e., received many good ratings), its reputation would suffer. Therefore, the agent will choose not to accept the job.

$$\begin{aligned}\forall r_1, r_2 \in \Re_+ : \mathfrak{T}(\mathfrak{A}(R_n, r_1)) > \mathfrak{T}(\mathfrak{A}(R_n, r_2)) \vee \\ \mathfrak{T}(\mathfrak{A}(R_n, r_2)) = 1, \\ r_1 > r_2 \Rightarrow |\Re_+| > 1\end{aligned} \tag{R1}$$

$$\forall r \in r_+ : \mathfrak{T}(\mathfrak{A}(R_n, r)) > \mathfrak{T}(R_n) \vee \mathfrak{T}(R_n) = 1 \tag{R2}$$

In [C15], we developed a trust metric which leverages *Simple Exponential Smoothing* (SES) [126]. This method can be seen as an advanced version of a rolling average, but it does not have to remember historic values. Therefore, R^s is no longer considered as a FIFO queue (15). As a side effect, newer ratings have more impact to the resulting value, which is desirable for many applications (i.e., this implements some kind of forgiveness [127]).

Those properties allow us to create an advanced metric based on R_n^w, which (i) fulfils our requirements and (ii) requires less memory. Obviously, SES can be used to reduce the memory usage of R_n^c down to just one float value. However, to use it on R_n^w, we need to store the weight of positive p_1 and negative p_2 ratings separately as a tuple (p_1, P_2). Based on the principle of SES, we also define $\mathfrak{A}^s(R_n^s, r)$ to modify the weight of either positive or negative ratings. However, both sums are multiplied by α or $(1-\alpha)$ to keep the values proportional (16). The value of α controls the influence of a new rating. A typical value for α would be between 0.95 and 0.999, depending on how many old values should influence the trust value (similar to k in the previous metrics). Calculating the trust values works by subtracting the negative weight from the positive weight and normalise the result by the total weight (see Equation (17); similar to $\mathfrak{T}^w(R^w_n)$ with the difference that the sums for positive and negative ratings are already precomputed).

$$\mathfrak{R}^s := [-1, 1], r^s \in \mathfrak{R}^w \tag{14}$$

$$R^s := [0, 1]^2 \tag{15}$$

$$\begin{aligned} R^s_{n+1} &:= \mathfrak{A}^s(R^s_n, r) \\ &:= \begin{cases} (p_1 \cdot \alpha + (1-\alpha) \cdot r, p_2 \cdot \alpha) & r > 0, (p_1, p_2) \in R^s_n \\ (p_1 \cdot \alpha, p_2 \cdot \alpha - (1-\alpha) \cdot r) & r < 0, (p_1, p_2) \in R^s_n \\ R^s_n & \text{otherwise} \end{cases} \end{aligned} \tag{16}$$

$$\tau^s := \mathfrak{T}^s(R^s_n) := \frac{p_1 - p_2}{p_1 + p_2}, (p_1, p_2) \in R^s_n \tag{17}$$

We will use this metric throughout this thesis because it allows rational behaviour in all situations and additionally uses very little memory. As part of this thesis, we evaluated more metrics which can be found in Appendix A.

3.3. Norms

Since agents are considered as black boxes they cannot be controlled directly from the outside. Each agent behaves autonomously and selfishly. However, we want to influence the system to optimise and make it more robust. Therefore, we introduce norms to change the incentives and sanctions for all agents.

In ODS, different attacks by malevolent agents can occur (for instance, the aforementioned *trust breakdown*). We implemented various counter and security measures to maintain a good utility for well-behaving agents. However, most of these measures appear with some attached costs. Although we do not benefit from those mechanisms under normal conditions, they are essential under attack or at least lead to a significantly faster recovery from attacks. Additionally, we can configure our reputation system and change the effect of ratings. This may increase or decrease robustness, but it also influences how fast new agents are integrated into the system. Offering larger incentives leads to a faster system start-up and a better speedup when well-behaving agents join the system. However, it also gets easier to exploit the system for malevolent agents.

In ODS, a variety of different parameters exist which influence the system behaviour. They must be set before system start. For example, they enable or disable security measures or change the influence of a rating to the reputation system. Some settings result in a better utility when no attacks occur, but lead to a higher impact on the performance in case of the system being under attack. There is no global optimal value for most of these scenarios. The ideal value or setting depends on the current situation.

To obtain the best overall performance, we need to change these parameters and settings during runtime according to the current situation. However, we cannot detect global system states, such as the *trust breakdown* or overload situations from the local viewpoint of an agent. It is also not possible to influence agents directly since they are autonomous. There needs to be a higher-level instance which can detect the current system state and consequently guide the agent's behaviour through indirect influences. The assessment of the current situation is presented in the remainder. Guiding agents is realised by norm-based control, which is briefly outlined in the following.

A norm is a rule with a sanction (such as laws in a society) or an incentive if certain conditions are met or violated. We formalise a norm as a triple:

$$\begin{aligned}
\text{Norm} &:= \langle \text{Evaluator}, \text{Action}, (\text{Policy}_1 \dots \text{Policy}_n)\rangle \\
\text{Policy} &:= \langle \text{Context}, \text{Sanction}\rangle \\
\text{Norm} &:= \langle \text{Evaluator}, \text{Action}, (\langle \text{Context}, \text{Sanction}\rangle, \langle \text{Context}, \text{Sanction}\rangle \dots)\rangle
\end{aligned}$$

The *Evaluator* is either the worker or submitter part of an agent. Both have different *Actions* they can perform. The following list names example actions from the TDG scenario for both components.

1. *Worker*
   a) AcceptJob(A_w, A_s): Agent A_w accepts a job from agent A_s
   b) RejectJob(A_w, A_s): A_w rejects a job from A_s
   c) ReturnJob(A_w, A_s): A_w returns the correct calculation for job to A_s
   d) CancelJob(A_w, A_s): A_w cancels job of A_s
2. *Submitter*
   a) AskForDeadline(A_s, A_w): A_s asks worker A_w for the deadline for a job

b) GiveJobTo(A_s, A_w): A_s asks worker A_w to do a job

c) CancelJob(A_s, A_w): A_s cancels a job A_w is working on

d) ReplicateJob($copies$): Copies a job multiple times and distributes them via GiveJobTo()

A norm may contain multiple *Policies* that consist of a *Context* and a *Sanction*, which can also be an incentive. The *Context* contains one or multiple conditions which must be true to trigger a certain *Sanction*. Since all agents want to achieve a maximal speedup, it is not possible to give a direct reward to an agent. We can only increase or decrease the speedup indirectly by varying the reputation of an agent. The *Sanction* may also influence more indirect values, which can influence the success of an agent. The following list summarises the possible interventions in terms of sanctions and incentives:

1. *Incentive*
   a) Reputation is increased
   b) Monetary incentives
2. *Sanction*
   a) Reputation is decreased
   b) Loss of monetary incentives

Figure 10 shows an exemplary norm which is used in the TDG in extended Object Constraint Language (OCL) format [128]. In this example, the norm formulates that a *Worker* should always finish a job and will receive an incentive which is stronger when the requester has a low reputation. Otherwise, the working agent will receive a sanction.

Agents in the ODS need to be able to understand the currently valid norms, which enables them to consider sanctions and incentives in their decision making. This allows them to follow short- or long-term strategies based on these norms. Since the agents are autonomous and free to obey or ignore these norms, the system still needs to enforce the sanctions and give incentives to agents [86]. To resolve conflicts which can arise when agents get falsely accused of norm violations, a mechanism based on Ostroms's eight principles for building enduring institutions is used [100] (see Section 2.1.5). Additionally, our reputation system implements forgiveness which allows the system to tolerate certain amounts of incorrect ratings caused by wrong accusations [129]. Furthermore, agents can get accused of norm violation when they are overloaded. To overcome this issue, we use the following conflict resolution mechanism: If challenged, the agent needs to prove that it is working by naming the submitters (to a third party) for which it is actually working.

Figure 10: Norm used in the evaluation constructed as invariant in extended OCL syntax [W38] which contains three sections: condition, incentive and sanction. The condition section describes the context and how the agent is supposed to behave. If the condition holds, the agent may receive an incentive as imposed by the incentive section which can contain multiple conditional statements. In the opposite case, the sanction section imposes sanctions. In this case, it only contains one entry which always applies but it can also contain multiple conditional entries. The norm sanctions cancelling work and rewards completing work. In addition, working for submitters with lower reputation generates a stronger positive rating.

$$\left.\begin{array}{l}\textbf{context}\ \overbrace{\text{Worker::returnJob()}}^{\text{Target/Role}}\ \textbf{inv}\text{:}\\ \quad\underbrace{\texttt{self.returnJob(job, requester)} = \texttt{true}}_{\text{Postcondition}}\end{array}\right\}\text{Condition section}$$

$$\left.\begin{array}{l}\textbf{incentivised}\\ \quad\textbf{if}\ \underbrace{\texttt{requester.reputation} < \text{highReputationThreshold}}_{\text{Sanction Condition}}\\ \qquad\textbf{then}\ \underbrace{\text{self.reputation += highWorkDoneIncentive}}_{\text{Incentive}}\\ \quad\textbf{if}\ \underbrace{\texttt{requester.reputation} \geq \text{highReputationThreshold}}_{\text{Sanction Condition}}\\ \qquad\textbf{then}\ \underbrace{\text{self.reputation += lowWorkDoneIncentive}}_{\text{Incentive}}\end{array}\right\}\text{Incentive section}$$

$$\left.\begin{array}{l}\textbf{sanctioned}\\ \quad\underbrace{\text{self.reputation += -cancelSanction}}_{\text{Sanction}}\end{array}\right\}\text{Sanction section}$$

3.4. Higher-Level Observer

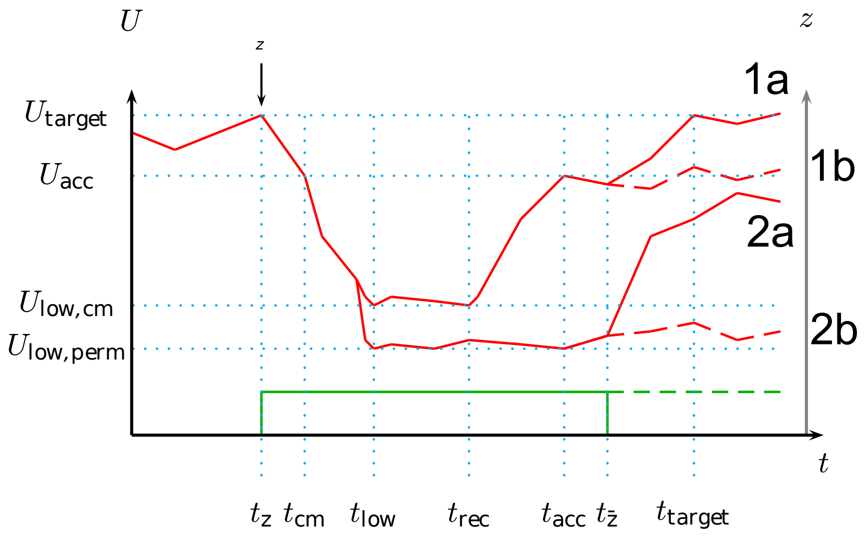
As discussed in Sections 1.5 and 3.1, ODS can suffer from NEB. We exemplify such an unwanted system state with the *trust breakdown* scenario in Section 1.5. To counter the negative effects in these situations and to further guide the self-organised system behaviour, we propose to establish a system-wide normative control loop. This control loop consists of an observer and a controller part. The observer derives a situation description from monitoring interactions among agents, while the controller issues norms as response to conditions where actions are required to prevent NEB.

In Figure 9, we present our concept of the NM, which uses the common O/C pattern [45]. The complete control loop implemented by the O/C component helps to mitigate effects of attacks to the TDG and allows a better fulfilment of the system goals. Thereby, it defines an intelligent control mechanism working at system level. However, if the additional NM fails, the system itself is still operational and can continue to run (this refers to the desired OC characteristic of *non-critical complexity* [130]). When the system has recovered, the NM can start to optimise the system again. Agents will elect a leader which runs the NM in a process similar to the election of a TCM (see Section 1.7.2).

3.5. Attacks, Recovery and Robustness

We assume that it is generally feasible to measure the utility over time (at least from the point of view of an external observer). However, it is often hard to quantify the strength of a disturbance z. Therefore, depending on the application, an estimation of z is required for our model. Furthermore, the system has to know a target utility value U_{target} (maybe the highest possible utility) and an acceptable utility value U_{acc} (a minimal value where the system is still useful to the user). The goal of the robustness model is to quantify the response of a system to a certain disturbance in terms of its utility and compare it to other disturbances or systems.

Figure 11: Utility degradation over time. At t_z a disturbance with strength z occurs. The utility U drops. When it reaches U_{acc} (i.e. the system state vector leaves the acceptance space, see [109]), a control mechanism (CM) is activated for (1) which only decreases to $U_{\text{low,cm}}$. At t_{rec}, recovery starts and (1) passes U_{acc} at t_{acc}. For (2) no CM is activated and utility drops to $U_{\text{low,perm}}$ and no recovery occurs. When the attack ends (a), the utility recovers U_{target} eventually. If the attack prevails or did permanent damage to the system (b), utility may stay at a lower value.



In Figure 11, we exemplarily show a typical utility function $U(t)$. In the beginning, U equals the target value U_{target}. At time t_z, a disturbance of strength z occurs (displayed in green) and the utility (red) decreases. Once it drops below the acceptance threshold U_{acc} at t_{cm}, a control mechanism (CM) starts to intervene. At t_{low}, U reaches $U_{\text{low,perm}}$ without an effective recovery mechanism and $U_{\text{low,cm}}$ if a CM is acting against the impact of the disturbance. With a CM, U starts to recover at t_{rec} and passes U_{acc} at t_{acc}. However, without a CM, U does not recover.

We can differentiate between two classes of behaviour during an attack:

1) An effective CM is started and the system recovers to at least U_{acc}.
2) The system does not recover during attack (or during a disturbance).

These two classes are represented by the curves marked as 1a and 1b for class 1, and with 2a and 2b for class 2 in Figure 11. Furthermore, we observe two different types of behaviour when the attack ends at $t_{\overline{z}}$. Either (a) the utility reaches the same value as before the attack (here U_{target} at t_{target}; solid red line in Figure 11), or (b) it stays at the same level as during the attack (dashed line). In total, this results in four different stereotypes of behaviour:

1a) The system S recovers during the attack to $U \geq U_{\text{acc}}$ and returns to $U \geq U_{\text{target}}$ when the attack ends. This is a *strongly robust system.*

1b) S recovers during the attack to $U \geq U_{\text{acc}}$ and stays there when the attack ends. This is a *weakly robust system.*

2a) S does not recover during the attack but returns to the previous value after the attack ends. Such a system shows just a certain ”elasticity”, we call it *partially robust*.

2b) S does not recover during the attack, it stays at the low utility level $U_{\text{low,perm}}$. Such a system is not robust at all.

While important, this only serves as a classification of different typical behaviours. However, to compare the effect of different disturbances z or different CMs, we have to quantify the utility degradation. We define the utility degradation D_U as the area between a baseline utility (U_{baseline}) and $U(t)$ for the time when $U \leq U_{\text{acc}}$ (see Equation (18)):

$$D_U := \int_{t_z}^{t_{\overline{z}}} (U_{\text{baseline}}(t) - U(t)) \, \mathrm{d}t \tag{18}$$

First, we need to define a baseline U_{baseline} for the measurement. This can be either hypothetical by using U_{target} or U_{acc} (in Figure 11) or we can run a reference experiment (as we will show later in Scenario 2, see Section 1.6.2). In order to maximise the system robustness, we have to minimise D_U. A system which never drops below U_{baseline} apparently has maximal robustness. We define the robustness during attack R_a as when U_{target} is reached at t_{target}:

$$R_a := \frac{\int_{t_z}^{t_{\overline{\mathrm{z}}}} U(t)\,\mathrm{d}t}{\int_{t_z}^{t_{\overline{\mathrm{z}}}} U_{\text{baseline}}(t)\,\mathrm{d}t}$$

For each CM to be compared, we measure a utility degradation D_U. This allows comparing the behaviour of two CMs during an attack (such as cases (1) and (2) from above) or the effectiveness of one CM for different attacks.

Furthermore, when the attack ended, we can measure the long-term utility degradation $D_{U,\text{long\_term}}$ which is the area between U_{baseline} and the actual U for the time after $t_{\overline{\mathrm{z}}}$. This allows us to compare the cases (1a/2a) and (1b/2b) from above. Hence, the long-term robustness R_l is defined as:

$$R_l := \frac{\int_{\overline{\mathrm{z}}}^{t_{\text{target}}} U(t)\,\mathrm{d}t}{\int_{\overline{\mathrm{z}}}^{t_{\text{target}}} U_{\text{baseline}}(t)\,\mathrm{d}t}$$

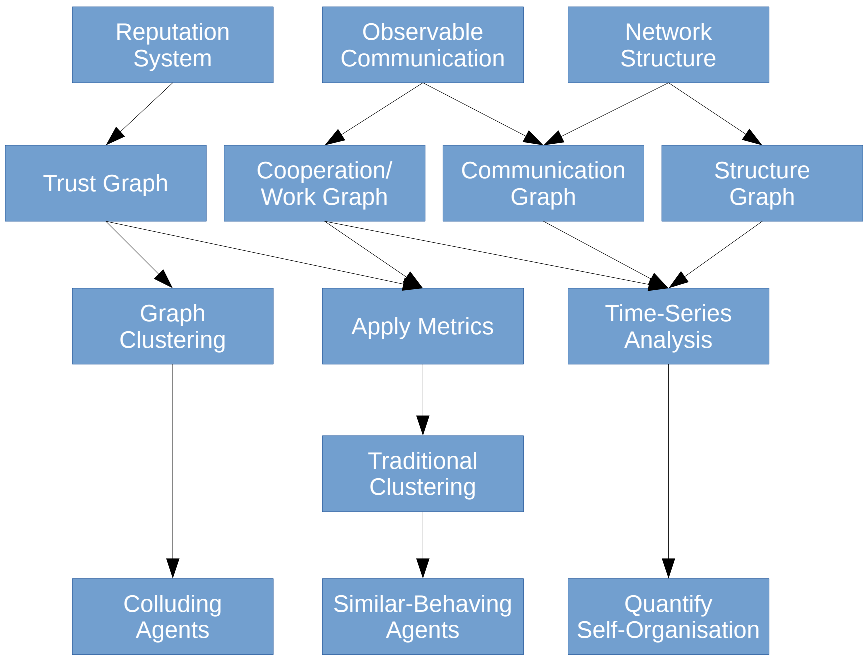
In cases where $U(t)$ never reaches U_{target} again, t_{target} is ∞ (also U_D is ∞) but we can calculate the open integral:

$$t_{\text{target}} = \infty \to R_l := \frac{\int_{\overline{\mathrm{z}}}^{t_{\text{target}}} U(t)\,\mathrm{d}t}{\int_{\overline{\mathrm{z}}}^{t_{\text{target}}} U_{\text{baseline}}(t)\,\mathrm{d}t} = \lim_{t\to\infty} \frac{U(\infty)}{U_{\text{baseline}}(\infty)}$$

See 2b in Figure 11 for an example. For instance, a permanent *trust breakdown* can cause such a disturbance in the TDG.

In all normal cases, R is assumed to be in the interval [0, 1]. It can never be negative and will only be larger than 1 if $U(t)$ improves through the disturbance which occurs only under very specific conditions (i.e. the system settled in a local optimum before the disturbance occurs, and is able to leave this local optimum as a result of the CM's intervention). Even if the utility $U(t)$ never recovers, we get an asymptotic robustness value. However, values of R_l with $t_{\text{target}} = \infty$ are not comparable to values for R_l with $t_{\text{target}} \neq \infty$ because the open integral would be always 1 in this case. In [B5, C7], we validated this approach against different application scenarios.

Figure 12: Concept of observer component as a work flow.



3.6. Observer – Assessing the Situation

In Figure 12, we present our concept for the observer. Several observable sources exist in ODS: i) the reputation system is available to all nodes and can be queried at very low costs (see Section 3.8); ii) the structure of the network is generally known (e.g. fully-meshed for OGS) or can be easily observed (e.g. in the sensor networks example, it is part of the routing information) (see Section 3.7.1); iii) furthermore, communication can be observed, although with some overhead (see Section 3.7).

In a first step, we create graphs based on those observations. In all cases nodes are agents. In Section 3.7.1, we describe the *Structure Graph* which is based on the network structure. Then, we introduce the *Communication Graph* as the subset of the *Structure Graph* where communication happened in a certain period (see Section 3.7.2). Furthermore, we define the *Work/Cooperation Graph* containing edges between agents which cooperate or work for each other (see Section 3.7.3). In Section 3.8, we depict the *Trust Graph* where edges represent trust relationships between agents.

Next, the NM calculates common graph metrics (see Section 3.9) for every node in the *Trust Graph*. Using statistics, the global system state is determined. Based on this metric, we use clustering algorithms to find groups of similar-behaving agents (see Section 3.14). By further classifying these groups, the Observer achieves an even better understanding about potentially happening attacks. Similarly, graph clustering is performed on the *Trust Graph* and *Work/Cooperation Graph* to find groups of colluding agents (see Section 3.13). Those groups can be TCs or groups of attackers which is determined using graph metrics and statistics. Furthermore, in Section 3.10.3, we perform a time-series analysis over *Communication Graph*, *Structure Graph* and *Work/Cooperation Graph* to assess the degree of self-organisation in the system In the end, the observer is able to classify whether the system is under attack, categorise the type of the attack, and rank attacks according to their severity. This is accompanied by an estimation of how accurate this information is.

3.7. Observing Communication

In order to measure the degree of self-organisation, three different *Communication Graphs* $G_i(p) = (V, E_i)$ are built. Each graph is generated for a period p between t_1 and t_2. In all graphs, the set of vertices V represents the same set of agents in the system (i.e. each agent represents an autonomous subsystem). However, the edges E_i describe different kinds of communication in the system. In G_p, they represent possible communication paths. In G_c, edges are actual communication, and in G_a, edges describe cooperation or work. We will use those graphs in Section 3.10.3 to measure the degree of self-organisation.

3.7.1. Building a Network Structure Graph

In the first graph G_p, each possible communication path represents an edge in E_p. The existence of a possible path requires a pair of agents a and b to reach each other directly without any relaying of third parties. A communication path exists when a could send a message and b would receive it with a very high probability. It is important to note that a initiates this transmission. In the same way b should be able to send a a reply.

In general, the existence of a communication path between a pair of agents is not observable. However, this fact is known for most systems or can be very accurately approximated. There exist two major types of G_p: the easiest one is fully-meshed where every pair of agents can communicate directly. This is typically known at design-time and constant over the runtime; the other case are types of G_p where communication is defined by locality. In some cases, the agents may also change their location and, thereby, the communication paths in E_p also change.

We consider G_p to be an undirected graph since most communication paths are bidirectional by their nature. However, this may not hold for all possible distributed systems and, therefore, can be considered as a limitation of this approach.

3.7.2. Building a Communication Graph

The second graph G_c represents the observed communication. The edges E_c in graph G_c represent a subset of E_p, because when a message was received there has to be a communication path in E_p. Obviously, G_c contains directed edges. Additionally, edges are also weighted by the number of messages sent from agent a to b. It is assumed that communication is confined to the necessary messages, i.e information is not transmitted redundantly. Moreover, broadcasts are generally omitted unless they contain specific information for the receiver, e.g. in wireless systems all messages are broadcasts but contain a receiver field and, thereby, are irrelevant for all agents but the receiver. Also, we consider communication to be only control information and not payload (i.e. in a self-organising routing system the actual payload traffic is not interpreted as communication).

3.7.3. Building a Cooperation and Work Graph

In the third graph G_a, edges in E_a represent mutual stable relationships between two agents. In general, relationships are temporary. They are initiated by negotiation over communication messages. E.g. agent a may ask b to cooperate for a certain task. If b agrees (and expresses this via a message) a *mutual relationship* or *agreement* was created. Afterwards, both partners may cancel this relationship by sending messages. Additionally, a relationship can also end at a certain time known to both parties or when certain conditions are met (e.g. the task is finished).

Edges in E_a are undirected since relationships are required to be mutual. We assume it to be irrelevant to measure multiple concurrent relationships between a pair of agents. Therefore, either one or none edges exist. $E_a(p)$ is not exactly a subset of $E_c(p)$ for every period p: A relationship between a and b may have been negotiated in a previous period p_{-1}, therefore, edges in $E_c(p_{-1})$ and $E_a(p_{-1})$ exist. In the next period p_0, those relationships may still be in place and, therefore, the edges in $E_a(p_0)$ exist. However, a and b may not communicate in period p_0 and, hence, there may be no edges in $E_c(p_o)$. Thus, when considering E_a and E_c for the whole runtime of the system, E_a is a subset of E_c. Accordingly, relationships negotiated outside the system boundaries or defined at design-time cannot be taken into account.

In this thesis, we consider a relationship to be undirected. Nevertheless, cases exist where agent a agrees to help b with a certain task but b does not help a (i.e. because currently a has no task). Therefore, those relationships would be directed. However, we assume that b would also work for a in most cases (see [J2] for a validation against multiple scenarios).

3.8. Observing the Reputation System and Building a Trust Graph

To analyse the system state, we build a *Trust Graph* $G = (V, E)$. We add all agents as nodes V. Afterwards, we fetch the trust relationships $T(a, b)$ between the agents a and b from the reputation system and add them as edges E to the graph. A trust relation defines how trustworthy agents rate each other within an interval between 0 (not trustworthy at all) and 1 (fully trustworthy). Agents fully trust themselves and, therefore, $T(a, a)$ is always 1 (self-trust). The weight of the edge e_{ij} which connects v_i and v_j represents the amount of trust between the connected agents a_i and a_j.

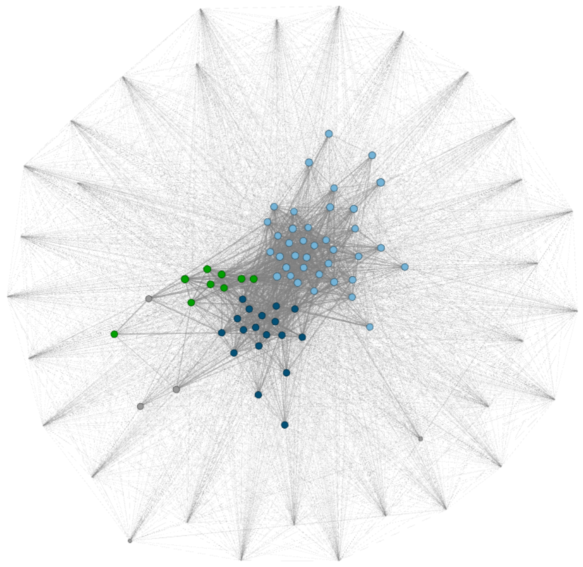
Figure 13 illustrates an exemplary *Trust Graph* from one TDG scenario. It has been graphically arranged using a Force Algorithm [62]. During the layout nodes push apart and edges between nodes attract nodes. Since edges represent trust, agents trusting each other are place next to each others.

Thereby, in the centre of the graph, agents trust each other and work together. We call this the *core* of the network. All isolated agents are located at the border of the graph and are only weakly connected. The graph is a complete graph, because there exists a trust relation between every pair of agents. However, we omit all weak edges below a certain threshold to improve performance and to allow for a better visualisation. This also allows us to apply algorithms which ignore the edge weight.

We decided to create directed and undirected *Trust Graph*s to be able to use more metrics. In general, trust is not necessarily mutual. For instance in the TDG scenario, a *Freerider* F may trust an *Adaptive Agent* A, because F had initial reputation and A worked for F. But most certainly, later on A does not trust F, because F refuses all work requests of A. This leads to a directed graph with two edges between every pair of nodes. However, after filtering out weak edges, there may only be zero or one edge left.

However, some algorithms only work on undirected graphs. Since trust is not mutual, we need to combine both edges for those algorithms. We decided to use the minimum of weights, because only two well-behaving agents mutually trust each other. However, with this approach, we loose asymmetrical trust relationships. Therefore, the undirected graph can be

Figure 13: Simple *Trust Graph* in TDG arranged by Force Algorithm [62]. Nodes are agents and edges are trust relationships. The size of a node represents its reputation.



used to differentiate between well-behaving and not well-behaving agents. In contrast, the observer can distinguish between different types of misbehaviour using the directed graph.

3.9. Graph Metrics

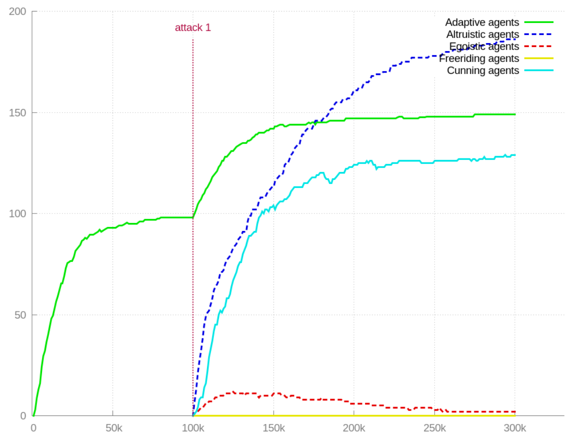
To evaluate how an agent behaved in the past, we analyse incoming and outgoing edges in the *Trust Graph*. For instance, *Freeriders* will have very few incoming connections because they did not work for others. In contrast, *Adaptive agents* will have similar incoming and outgoing trust edges. Therefore, we apply metrics from graph theory to the problem. We select a subset of metrics based on [131] from the literature.

3.9.1. Prestige

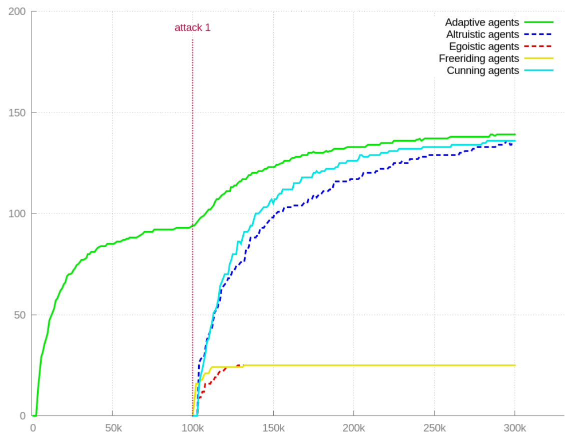
The *Prestige* allows us to rate how altruistic an agent behaves. As shown in Figure 14, *Altruistic agents* get the highest value followed by *Adaptive Agents* and *Cunning Agents*. The values for *Freeriders* and *Egoists* are near zero. This metric counts the incoming edges for a node in a directed graph (see Equation (19) and Figure 15a). In our system, every incoming edge represents trust by another agent which causes the effect above.

$$\text{Prestige}_i := |\{v_j : e_{ji} \in E\}| \tag{19}$$

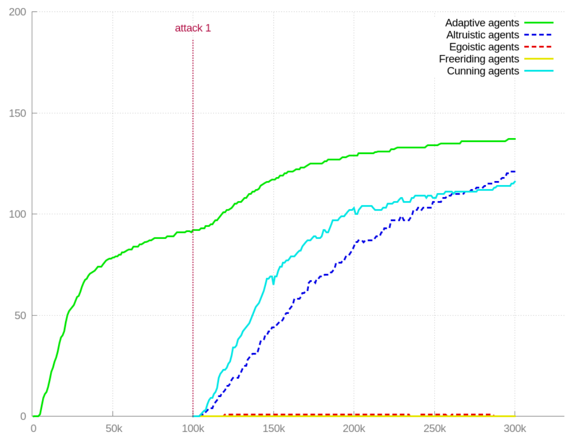
Figure 14: Metrics over time for the TDG scenario [J3].



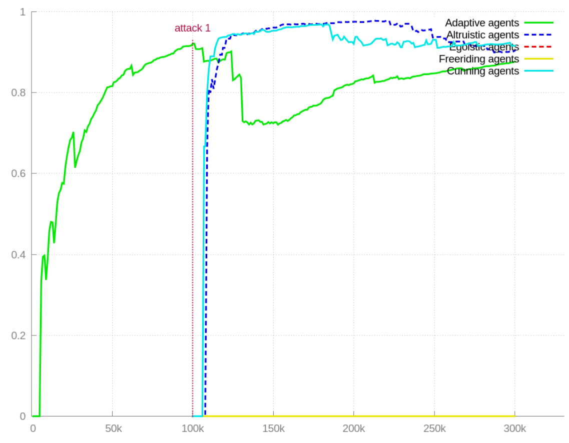
(a) Prestige over time



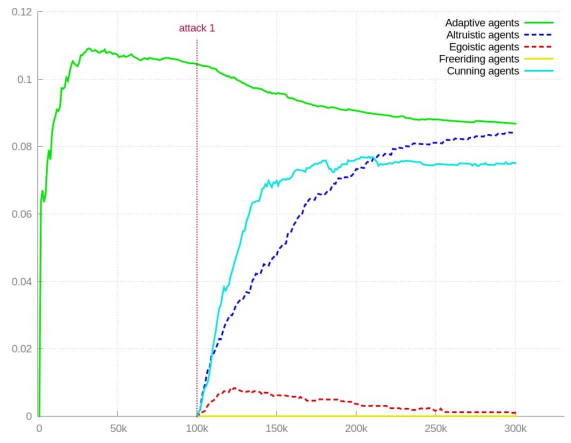
(b) Actor Centrality over time



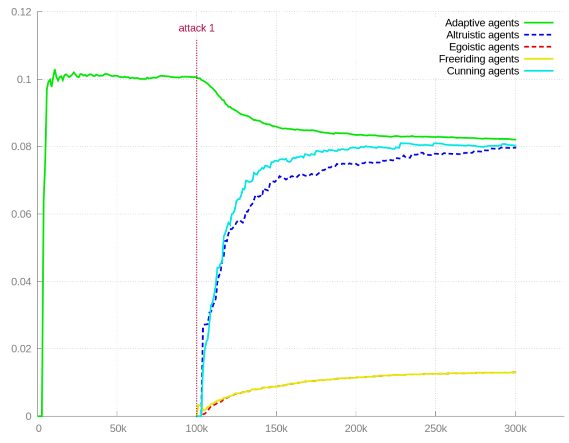
(c) Degree Centrality over time



(d) Clustering Coefficient over time



(e) Authorities over time



(f) Hubs over time

3.9.2. Actor Centrality

This metric counts the outgoing edges for a node in a directed graph (see Equation (20)) and allows us to distinguish cooperative from non-cooperative agents. In general, one of these edges expresses that the agent gave some positive rating to another agent for successful cooperation. Actor Centrality (see Figure 15b) behaves inversely to Prestige because it considers outgoing instead of incoming edges.

$$\text{ActorCentrality}_i := |\{v_j : e_{ij} \in E\}| \tag{20}$$

3.9.3. Degree Centrality

This metrics counts all edges for a node in an undirected graph (see Equation (21)). Since we use the minimum function to unify edges in the undirected graph, it is more than a combination of Prestige and Actor Centrality. In ODS, Degree Centrality will count mutual trust relationships which can be used to identify *Adaptive Agents*. In Figure 15c, we show the Degree Centrality of the agent groups.

$$\deg(i) := N_i := \{v_j : e_{ij} \in E \land e_{ji} \in E\}, D_i := |N_i| \tag{21}$$

3.9.4. Clustering Coefficient

This is a metric to measure the degree to which nodes tend to form a cluster in an undirected graph (see Equation (22) and Figure 15d). This can be used to identify cooperative agents and distinguish them from colluding agents which issue fake ratings because those are less central in the general graph. It helps to find groups of connected nodes with a high density of edges [132].

$$C_i := \frac{|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{D_i(D_i - 1)} \tag{22}$$

3.9.5. Authorities and Hubs

auth and *hub* values allow us to categorise agents into a cooperative and a non-cooperative group. This metrics is calculated by the Hypertext-Induced Topic Selection (HITS) algorithm [133] to rate the importance of a node in the graph. Nodes with a high Authority *auth* value have numerous incoming edges from nodes with a high Hub *hub* value. Similarly, nodes have a high *hub* value when they have numerous outgoing edges to nodes with a high *auth* value. Since this metric is defined recursively, they have to be calculated iteratively. At the beginning, all values get initialised by 1 (see Equation (23)). In the next step, the value of *auth* is updated for every node (see Equation (24) and Figure 15e). Then, the value of *hub* is determined (see Equation (25) and Figure 15f). Originally, after that step, all values get normalised but we skipped that because it would change absolute values when new nodes appear. Afterwards, the previous two steps are repeated until the algorithm converges.

$$\forall p : \mathrm{auth}(p) = 1, \mathrm{hub}(p) = 1 \tag{23}$$

$$\mathrm{auth}(p) := \sum_{i=1}^{n} \mathrm{hub}(i) \tag{24}$$

$$\mathrm{hub}(p) := \sum_{i=1}^{n} \mathrm{auth}(i) \tag{25}$$

3.9.6. PageRank

The PageRank [134] is calculated by a ranking algorithm used by Google (see Equation (26)). It is similar to HITS and ranks nodes by their importance in the graph. However, the accuracy is better than HITS and it does not calculate hubs. Values range from 0 to 1 and can be computed iteratively.

$$\mathrm{PageRank}_i = \frac{1-d}{n} + d \sum_{j \in \{1,\ldots,n\}} \frac{\mathrm{PageRank}_j}{c_j} \tag{26}$$

3.10. Measuring Self-Organisation Activity

In the previous section, we introduced metrics which can be applied to single vertices (representing agents) in our graphs resulting in a set of values. In this step, we introduce statistical methods to unify and evaluate the set of values.

3.10.1. Analysing the Network Structure Graph

This technique measures the amount of communication. We measure the number of possible communication partners in the *Structure Graph* (see Section 3.7.1). We use *Degree Centrality* deg(i) on G_p and apply a 20%-percentile (30) to filter out the edges of the network.

$$D_s(G) := (\deg(i) : v_i \in V) \tag{27}$$

$$D_{s,i}(G) \in D : D_{s,i-1}(G) \leq D_{s,i}(G) \leq D_{s,i+1}(G), i \in N \tag{28}$$

$$n := \left\lfloor \frac{20}{100} \cdot |D_s(G)| \right\rfloor \tag{29}$$

$$\text{Partners}(G) := D_{s,n}(G) \tag{30}$$

$$\text{Partners}(G) \geq \rho \geq 3 \tag{31}$$

In general, we require every agent to have a sufficient number of communication partners ρ. Effectively, this value is at least three because reputation systems do not work well in very sparse networks. However, some nodes may (at least temporarily) have fewer partners (i.e. in wireless systems), therefore, we take the percentile and allow 20% of the nodes to have fewer partners (31).

3.10.2. Communication Graph

We expect agents to communicate less when there are no disturbances and, therefore, no self-organisation is occurring. The amount of communication can be calculated by counting the number of edges $|E_c|$ in G_c. However, when we compare communication for two different periods, communication paths may have changed. Therefore, we normalise the actual communication using the number of paths in G_p in the period (32). This results in a density metric $\mathfrak{C}$ and allows comparing different periods. Since $\mathfrak{C}$ compares the undirected *Structure Graph* with the directed *Communication Graph*, the number of edges in the *Structure Graph* are multiplied by two.

$$\mathfrak{C}(G_c) := \frac{|E_c|}{|E_p| \cdot 2} \tag{32}$$

3.10.3. Degree of Self-Organisation

We define a function $\mathfrak{D}(G)$ to quantify the structure of a graph. Therefore, we present a function which is based on Shannon's entropy with p as the probability of an element x in X (33). We define a set $D_u(G)$ containing all distinct values for Degree Centrality $deg(i)$ (34).

We also need a probability mass function $p_D(x)$ which counts occurrences of a certain D_i value (35). Afterwards, we can easily calculate $\mathfrak{D}(G)$ for a graph $G = (V, E)$ (36).

$$H(X) := -\sum_{x \in X} p(x) \cdot \ln_2(p(x)) \tag{33}$$

$$D_u(G) := \{\deg(i) : v_i \in V\} \tag{34}$$

$$p_D(x) := \frac{|\{v_i : \deg(i) = x \land v_i \in V\}|}{|V|} \tag{35}$$

$$\mathfrak{D}(G) := H(D_u(G)) := -\sum_{x \in D_u(G)} p_D(x) \cdot \mathrm{ld}(p_D(x)) \tag{36}$$

Other variants based on metrics from Section 4.3.7 and stochastic kernels are possible and discussed in [C8].

3.10.4. Changes in Structure

Measuring the change of structure in a graph is a non-trivial problem. Therefore, we define a very basic function $\mathfrak{S}(G_1, G_2)$ for initial evaluations which counts the number of changed edges between two graphs G_1 and G_2 (37). We normalise the metric by the number of agents in the system to allow limited comparisons of absolute values.

$$\mathfrak{S}(G_1, G_2) := \frac{|\{e_{ij} : e_{ij} \in E_1 \oplus e_{ij} \in E_2\}|}{0.5 \cdot (|V_1| + |V_2|)} \tag{37}$$

3.11. Measurement Periods

To compare and quantify self-organisation in a system, the effect of this mechanism has to be measured. Therefore, we use the function $\mathfrak{S}(G)$ (see Equation (37)) which quantifies the structure of a certain graph and use it to compare time intervals. We propose to use the following two situations where this effect is observable in ODS.

3.11.1. Situation 1: Building Structure

In the first situation, we measure the emergence of structure in the system. Therefore, we define three contiguous measurement periods: $p_{s,\text{unstructured}}$ is the time where the system did not build the structure yet (38); $p_{s,\text{emergence}}$ defines the period where structure emerges (39);

$p_{s,\text{structured}}$ is the time frame when structure is steady and settled (40).

$$p_{s,\text{unstructured}} := [t_\text{u}, t_\text{e}) \tag{38}$$
$$p_{s,\text{emergence}} := [t_\text{e}, t_\text{s1}) \tag{39}$$
$$p_{s,\text{structured}} := [t_\text{s1}, t_\text{s2}) \tag{40}$$

We expect to observe an increase in the structure in G_a from $p_{s,\text{unstructured}}$ to $p_{s,\text{structured}}$ (41). Obviously, we expect to recognise structure changes afterwards (42). Additionally, we expect to see more communication during $p_{s,\text{emergence}}$ than before and afterwards (43, 44).

$$\mathfrak{D}(G_a(p_{s,\text{unstructured}})) < \mathfrak{D}(G_a(p_{s,\text{structured}})) \tag{41}$$
$$\mathfrak{S}(G_a(p_{s,\text{unstructured}}), G_a(p_{s,\text{structured}})) \geq \delta, \delta > 0 \tag{42}$$
$$\mathfrak{C}(G_c(p_{s,\text{emergence}})) > \mathfrak{C}(G_c(p_{s,\text{structured}})) \tag{43}$$
$$\mathfrak{C}(G_c(p_{s,\text{emergence}})) \geq \mathfrak{C}(G_c(p_{s,\text{unstructured}})) \tag{44}$$

We have shown in [W32] that this approach works well for all three scenarios mentioned in Section 1.6. Tomforde et al. [C8] extended our approach using stochastic kernels to require less knowledge about the internals of the system. This allows the same approach but does not require minimal communication nor any knowledge about the communication. However, it significantly increases the complexity, probably reduces the accuracy and has to be proven in practical applications.

3.11.2. Situation 2: Disturbance and Recovery

Self-organising systems should be able to recover from failures or disturbances (see Figure 11). Therefore, in situation two, we measure the change of structure during those incidents. Again, we define three periods: $p_{r,\text{reference}}$ is a measurement period before any disturbance occurs. However, we assume that emergence of structure is already completed (45). In the beginning of $p_{r,\text{disturbance}}$ at t_d, the disturbance or failure occurs. Additionally, still within this period, the system also recovers (46); afterwards, at the start of the last frame $p_{r,\text{recovered}}$, the system is recovered and working with undisturbed performance again (47). The length of $p_{r,\text{recovered}}$ should be chosen similar to the length of $p_{r,\text{reference}}$ for comparison reasons.

$$p_{r,\text{reference}} \coloneqq [t_\text{s}, t_\text{d}) \tag{45}$$
$$p_{r,\text{disturbance}} \coloneqq [t_\text{d}, t_\text{r1}) \tag{46}$$
$$p_{r,\text{recovered}} \coloneqq [t_\text{r1}, t_\text{r2}) \tag{47}$$

In this situation, we expect the degree of structure of the system to be similar before the disturbance and after recovery (48). Additionally, we expect some structures to change (49). The exact value required for the degree of change σ depends on the application (see Equation (49)). In our scenarios, a value of 0.1 to 0.2 has proven optimal. Some small changes constantly happen but during self-organisation those changes are huge compared. Moreover, we expect to observe more communication during the recovery period (50, 51).

$$\mathfrak{D}(G_a(p_{r,\text{reference}})) \approx \mathfrak{D}(G_a(p_{r,\text{recovered}})) \tag{48}$$
$$\mathfrak{S}(G_a(p_{r,\text{reference}}), G_a(p_{r,\text{recovered}})) \geq \sigma, \sigma > 0 \tag{49}$$
$$\mathfrak{C}(G_c(p_{r,\text{disturbance}})) > \mathfrak{C}(G_c(p_{r,\text{recovered}})) \tag{50}$$
$$\mathfrak{C}(G_c(p_{r,\text{disturbance}})) > \mathfrak{C}(G_c(p_{r,\text{reference}})) \tag{51}$$

3.11.3. Practical Observation

Understanding the different phases of self-organisation helps to analyse systems utility in retrospect. However, the observer has to decide timely, and, thus, cannot wait until recovery happened and was detected. Therefore, we implement a sliding window approach to detect the current and previous degree of self-organisation (see [C8] for details).

3.12. Clustering of Groups and Aggregation

As a very simple yet effective way to detect groups in the directed *Trust Graph* (see Figure 13), we calculate the global reputation τ^s of every node (see Equation (17)). This is similar but not equal to the weight of all incoming edges of a node. If there are no interactions between two agents, we assume a default weight, but we do not have a rating in that case.

Using the reputation value, we build three buckets:

- **High Reputation** – Reputation τ value higher than 0.5. These agents are very trustworthy.
- **Suspicious** – Reputation τ between 0 and 0.5. All newcomers are in this range. There is no consensus whether these agents are trustworthy.
- **Low Reputation** – Reputation τ below 0. These agents are mostly isolated from the system. Most other agents will refuse cooperation.

This approach uses domain knowledge about our reputation system to sort agents into buckets. By assessing the size of the buckets, we can tell the general system state. If **Suspicious** contains significant numbers of agents, there are likely attacks happening or newcomers just joined the system. The NM will perform actions to increase the robustness of the system if **Suspicious** becomes large. If a high percentage of agents ended in the **High Reputation** bucket, the system is usually very stable and can be optimised for performance. Agents in **Low Reputation** are usually uncooperative. If they already have been isolated they the NM will mostly ignore them but make sure that the system is robust against free-riding. Otherwise, it the NM will take actions to isolate them. If **High Reputation** and **Suspicious** are very small compared to **Low Reputation**, a *trust breakdown* likely occurred. In this case, recovering from this NEB is the first priority.

3.13. Clustering of Colluding Agents

Another way to find groups in a graph is to use clustering algorithms. In previous work [C24], we use the approach presented by [135] and apply it to the directed *Trust Graph*. It finds groups and merges them to maximise the modularity [136]. We call the largest group in the graph *core* and all remaining agents *non-core*. In Figure 13, all well-connected agents in the centre would be the core. However, this very simple approach did not work equally well for all agent groups. For instance, not all *Adaptive Agents* joined the *core*. Additionally, often some *Altruistic Agents* also joined the core.

The advantage of using clustering on the *Trust* or *Work Graph* is that it is not affected by NEB such as a *trust breakdown*. During such a breakdown, agents will receive additional bad ratings from attackers. This will decrease the reputation (Equation (17)) because it considers all ratings. However, agents in one group still mutually trust each other and clustering still works.

To find colluding groups of agents, we need a graph clustering algorithm which fulfils the following requirements:

- **Weighted Cyclic Non-Symmetrical Graph** – The algorithms has to work directly on a weighted graph where distances are not symmetrical $d(x, y) \neq d(y, x)$, cycles may exist and, especially, the triangle inequality $d(x, z) \leq d(x, y) + d(y, z)$ does not hold [137].
- **Semi-Deterministic/Order-Invariant** – The clustering is performed periodically while the underlying graph slightly changes. However, the resulting groups should not change fundamentally. Some algorithms alternate between two different results when the input changes only marginally. This often happened with our previous approach. Therefore, we require a similar output when the input changes only slightly [138].
- **Dynamic Group Count** – Typically, we do not know the total number of different agent groups in advance. Therefore, we require an algorithm which can find a dynamically changing number of groups.
- **Non-Connected Components** – The graph is not necessarily always connected. Some agents or even groups may be permanently disconnected from the other groups. The algorithm has to be able to cope with this requirement.

Based on this analysis, we choose *Markov Cluster Algorithm* (MCL) [139] and Blondel [135] as a suitable algorithm for graph clustering which uses flow simulation (see [C11] for more details; [140] gives an overview over possible algorithms). MCL is deterministic, can handle edge weights and can find a dynamic amount of groups. Therefore, it meets our requirements and can be used with directed or undirected edges.

In the previous section, we only considered groups of colluding agents which can be either well-behaving agents or some kind of attackers. However, not all attackers have to collude. Therefore, in this section, we cluster similar-behaving agents which can be in different groups or in no group at all.

We capture this behaviour using our previously defined metrics and omit all connections in the graph. Therefore, every agent forms a data point in an n-dimensional space where n is the number of metrics considered. This allows us to use algorithms known from traditional clustering.

3.14. Clustering of Similar-Behaving Agents

To identify groups of similar-behaving agents, we apply our metrics to the *Trust Graph* and evaluate them for every node (representing an agent) in the graph. Afterwards, we apply a clustering algorithm to find similar-behaving agents (according to our metric). We capture this behaviour using our previously defined metrics and omit all connections in the graph. Therefore, every agent forms a data point in an n-dimensional space where n is the number of metrics considered. This allows us to use algorithms known from traditional clustering.

To cluster similar-behaving agents, we need an algorithm which fulfils the following requirements:

- **Multi-Dimensional Input** – We characterise every agent by multiple metrics, and the algorithm should be able to take all of them into account. Therefore, it has to cluster an n-dimensional vector for every agent.
- **Semi-Deterministic/Order-Invariant** – The clustering is performed periodically while the underlying graph slightly changes. However, the resulting groups should not change fundamentally.
- **Dynamic Group Count** – Typically, we do not know the number of different agent groups in advance. Therefore, we require an algorithm which can find a dynamic amount of groups.
- **Robust Against Outliers/Noise** – Single agents may behave differently from the rest of a group and the algorithm has to be robust against those outliers. Outliers should form a separate individual group or be ignored altogether.

Based on [141], we selected *Balanced Iterative Reducing and Clustering using Hierarchies* (BIRCH) [142] as a suitable algorithm (see [J2, C14] for more details).

3.15. Controller – Changing Norms

The controller is responsible for guiding the overall system behaviour by applying norms. Such a norm contains a rule and a sanction or an incentive (see Section 3.3). Agents are still autonomous and can violate norms with the risk of being sanctioned.

Based on the information obtained by the observer, the controller decides whether the system norms need to be changed. Norms cannot directly influence agents but modify their actions. To be more specific, norms can impose sanctions or offer incentives to actions. To defend against attacks, we can increase sanctions for certain actions. Under certain conditions we can allow agents to perform security measures, which would lead to sanctions otherwise [87].

To keep the system scalable, the NM has to keep the number of norms limited. Therefore, we try to keep the focus of a norm very narrow which simplifies to filter norms. This is implemented using the context (see Figure 10 for an example norm) which specifies when and to whom the norm applies. Additionally, the NM will always try to merge norms with the same context to make parsing computationally faster [J3].

3.15.1. Incentives vs. Sanctions

Incentives can influence the utility in decision making of autonomous agents. Instead of discouraging agents from making a bad decision, we try to encourage them to make a good one. This also renders the bad decision less attractive but agents may still take it if they gain a greater advantage from it. In our experience, the risk for NEB decreases when using this method. Additionally, if agents do not change their behaviour despite a big incentive, the action is probably highly undesirable from their local perspective and the controller should find another way to guide the system towards the global goal.

One way to discourage a certain action is to impose sanctions in a norm. However, this method often leads to NEB such as a *trust breakdown* when a majority of agents decides to violate the norm. Additionally, it limits the autonomy of agents which may have a good reason to choose the action. Therefore, this should be used carefully and selectively.

It can be a powerful tool to remove sanctions under certain conditions: For instance, the controller may temporarily allow (i.e. not sanction) an agent to stop cooperating with agents which did not prove to be trustworthy yet. This does not limit the agent's autonomy and gives them more options in their decisions. However, in this case, it may ban some agents from participating and should, therefore, not be used permanently (see [C20] for more details).

3.15.2. Robustness vs. Performance

In most cases, the robustness of a system can be increased at the cost of performance. One obvious example in the distributed rendering scenario is to increase the replication factor in a system which creates a high overhead but might prevent certain attacks. More generally, in most scenarios, when increasing incentives new agents will be integrated faster which is desirable. However, at the same time, most attacks get more powerful. Therefore, the controller needs to ensure a good balance based on the known groups in the system. In [C18], we performed a series of experiments to evaluate this trade-off for the TDG.

The system can be tuned for performance when no attacks occur, but as soon as increased levels of self-organisation are observed or the utility drops, the controller needs to adjust the system for robustness. This simple but powerful mechanism allows the NM to employ high-overhead security measures during attacks to get a robust system and to increase performance when no attacks occur.

3.15.3. Decision Making

The controller constantly receives updated situation descriptions from the observer and has to make decisions based on them. The first challenge is *when* the controller should act. Depending on the scenario this either occurs (i) when utility undershoots a known acceptance threshold or (ii) when a lot of self-organisation is detected. However, it is often not feasible to set or know the acceptance threshold and, therefore, one approximation is to detect drop in the utility. Detecting self-organisation is more robust because is also observes changes which do not immediately affect utility. This allows for certain optimisations such as tuning the system for performance when attackers got isolated or improving robustness when malicious agents are present but did not start an attack yet. Therefore, we use a combination of both approaches to decide when to act.

The next challenge for the controller is *how* to act. Since we do not want to design this in advance we use a learning algorithm for this purpose. However, norms certainly offer a lot of flexibility to express conditions and actions which is, thus, hard to explore completely. Additionally, the feedback loop between changing a norm and measuring the result is long (see [C23]), so even advanced learning algorithms such as deep-learning [143] are not feasible. Thus, we require norms with free parameters in either the condition or action part (see Figure 10). In most cases, the values for all sanctions and incentives can become variables and the same applies to thresholds for trust and reputation. Depending on the applications, other variables are possible which are usually somehow tied to the utility function. For instance in the distributed rendering scenario, we turn the reputation factor into a variable in the norm. Using this approach, we reduce the number of parameters and apply simple learning algorithms. Based on our evaluation in [C23], we use reinforcement learning to learn based on the current situation. The output of the learning algorithm will influence whether the norm is active and, if yes, which parameter is used.

3.16. Normative Trust Communities (nTCs)

All agents are autonomous and can decide to adhere to the NM or not. Similar to an iTC, we define the nTC as all agents which adhere to the norms issued by the NM. Multiple nTCs can exist in a system but cooperation between those groups will be difficult when they issue mutually exclusive norms. However, multiple nTCs can exist in the form of eTCs. In that case, the TCM issues norms which are only valid for agents within the eTC. All other norms should be still in place in the eTC (at least when communicating to agents outside of the eTC). This can be also seen as a system of system approach and may be extended to more level (see [W27] for details).

4. Evaluation

To verify our approach, we apply it to the three initially mentioned application scenarios: low-power sensor networks, the TDG and distributed rendering (see Section 1.6). For every scenario, we explain the specifics needed to apply our method and show an evaluation afterwards.

4.1. Sensor Networks

In RPL, nodes have only very limited knowledge about the topology to save power and memory. Since one of the main reasons for energy consumption is the use of radio, there should be as little communication as possible. Because of this constraint to minimise communication, there are no acknowledgements from the *root* about received packets. Unfortunately, this means that nodes cannot know whether their packets were actually forwarded and delivered.

However, to detect malicious or broken nodes, we need to gain knowledge whether our packets actually reached the *root*. Normally, most communication is one-way in such networks, so, there is no feedback. Therefore, we introduce end-to-end trust as an effective low-overhead measurement to fill the information gap about the delivery rate of a route. Our NM runs in the *root* node which is usually not battery powered and has sufficient computing resources.

4.1.1. End-to-End Trust

To achieve end-to-end trust, we first add a sequence number to every packet from a node to *root*. This allows the *root* to determine if it missed packets. However, it still cannot tell whether it got the last packet and, more important, the sending node does not know. Fortunately, the *root* periodically sends a DIO to all nodes (see Section 1.6.1 for details). We add a header to all DIOs which contains the Trust Round (TR) (see Section 4.1.2) and the count of all received packets in this round. Additionally, we also trigger a DIO when the *root* sees a high count of missing packets. Nodes will receive a forwarded DIO from each of their neighbours. However, according to RFC 6550 [56], they shall ignore it from all but their parents. This ensures that every node knows about the last sequence number seen by the *root* and the amount of missing packets. An attacker can only prevent a node from receiving the correct DIO if it controls all neighbours of that node and in that case there would not be a working route anyway. In theory, the attacker would only have to control the parents. Fortunately, implementations (i.e. *Contiki* [144]) also consider DIO from non-parent neighbours which improves the robustness of the implementation because an attacker would have to control all surrounding peers.

4.1.2. Trust Rounds

Previous approaches (such as [116] had problems to determine whether a packet was actually sent before or after the DIO was generated since clocks were either not existent or not synchronised in sensor networks. To solve this dilemma, we introduce so called Trust Round which are used to synchronise packet counters. Nodes attach the TR and a sequence number to every packet. When the *root* generates a DIO, it starts a new TR. After nodes receive the DIO, they reset their sequence counter to zero and use the new round. The parent will only be reselected when a new round starts or when the old parent becomes unavailable. At the beginning of a TR nodes re-evaluate their parent selection and will select a better performing parent if available. This optimisation helps to save memory on all nodes since they only need to remember their sequence and the selected parent.

4.1.3. Trust Metric and Objective Function

With both sequence numbers and TRs, nodes will eventually know which packets reached the *root* and which did not. Based on this knowledge, a node calculates the delivery rate for the parent used in the last TR. This rate should be near to 100% in normal cases, because a packet is retransmitted if no link level acknowledgement was received by the radio.

We use a simple trust metric $T(n, p|r)$ to calculate the trust for all neighbours $n \in \texttt{Nodes}$ (see Equation (57)). In every round r, a node calculates the new direct trust value $T(n, p|r)$ for its selected parent $P(n|r)$ (see Equation (52)) according to the new experiences based on the PDR (see Equation (55)) and the previous trust value $T(n, p|r-1)$. The factor α decides how strong the new experiences are weighted compared to the previous ones (see Equations (56) and (57)). The initial trust value $T(n, p|0)$ of every node is set to 0.5 which represents a neutral reputation (see Equation (53)).

The set $\texttt{Nodes}$ contains all nodes in our WSN. Additionally, we define $N(n) \in \texttt{Nodes}$ to contain all nodes which are located physically next to node n and can communicate with node n using the Radio Frequency (RF) interface (i.e. we consider only neighbouring nodes of node n). For simplicity, changes in physical location are neglected in the formulas.

$$P(n|r) \in N(n) \tag{52}$$

$$T(n,p|r) \in [0,1], T(n,p|0) := 0.5 \tag{53}$$

$$P(n|r) := p, p \in N(n) \land T(n,p|r-1) = \max_{n_2 \in N(n)} T(n,n_2|r-1) \tag{54}$$

$$\tau := \text{PDR} := \frac{\#\text{delivered}}{\#\text{sent}} \tag{55}$$

$$\alpha \in [0,1] \tag{56}$$

$$T(n,p|r) := \begin{cases} \alpha \cdot T(n,p|r-1) + (1-\alpha)\cdot\tau & p = P(n|r-1) \\ T(n,p|r-1) & \text{otherwise} \end{cases} \tag{57}$$

When a node receives a DIO which starts a new TR, it will update the trust value for the current parent. We define an Objective Function (OF) which then uses this metric to select suitable parents. Initially, all nodes are equally trustworthy and the behaviour is equal to standard RPL with Objective Function Zero (OF0) according to RFC 6552 [145]. Each node increases its *rank* to be higher than those of the suitable parents. It then selects the best parent. If the trust value of the parent decreases, it will at some point no longer be considered as a "good" parent. The OF will select a higher ranked parent. Eventually, the node needs to adjust its *rank* to have more suitable parents available.

4.1.4. Overhead

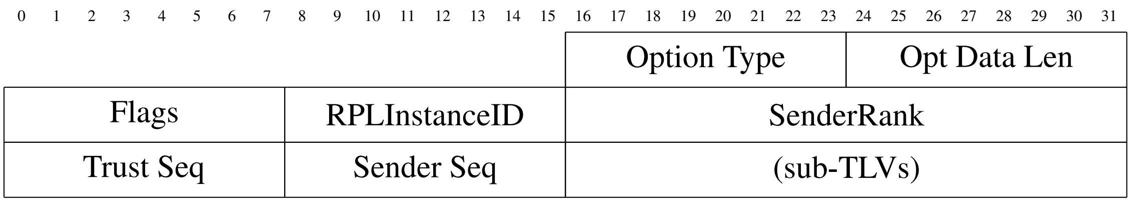
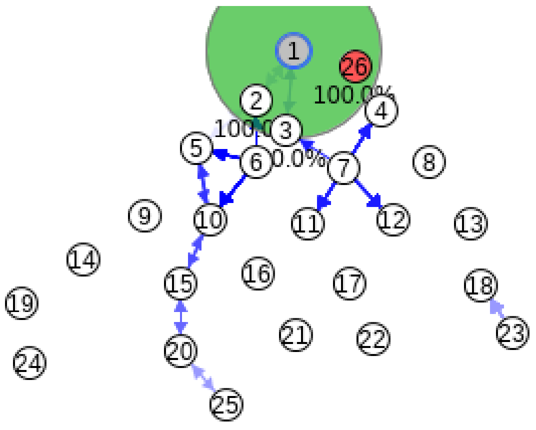


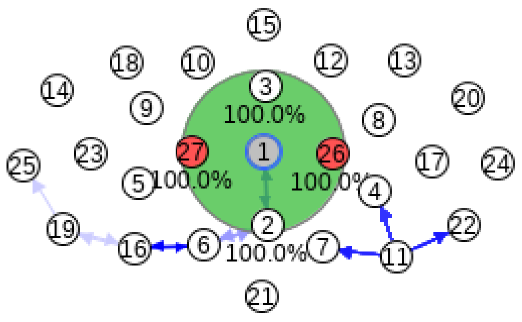
Figure 15: RPL header extension for sender and trust-sequence in Type-Length-Value (TLV) format according to RFC 6553 [146].

Sending additional information always causes overhead which also applies to our RPL header extension (as shown in Figure 15). However, we try to minimise the impact on power consumption. Therefore, we do not send additional packets to keep radio and CPU in power save mode as long as possible. Instead, we use existing messages which are sent anyway and add our data into the message. Additionally, we compress the data as much as possible: Since sequence number and TR are only two bytes they should fit inside the padding of most

Figure 16: Experiment setup with attackers coloured in red, normal nodes in white and *root* in gray. Exemplary paths are shown in blue and the green circle represents the radio of a root. Nodes do not move during the experiment. Attackers perform coordinated periodic attacks.



(a) Experiment A: One attacking node



(b) Experiment B: Two attacking nodes

packets; in contrast, the DIOs which are sent by the *root* need an entry for every node in the network such that the amount of data is not negligible. One main concern is the size of IPv6 addresses which are 16 bytes for every node. E.g. for a network with 400 nodes the DIO would grow to about 7 kB. However, most sensor networks use a common prefix for all nodes and we can exploit this to decrease the address size. We choose a prefix of 14 bytes, so, our addresses contain two bytes. Adding the one byte counter for the received packets, we result in a three byte entry per node. The resulting DIO for 300 nodes with about 1200 bytes still fits inside the standard Message Transfer Unit (MTU)[1].

To measure the effectiveness of our approach, we implemented our RPL extension in *Contiki* [144] and ran simulations in *Cooja* [148]. *Cooja* allows us to use the same code which also runs on real hardware with similar restrictions to memory, CPU and radio. In this experiment, we set up a multi-hop sensor network with 26/27 *Tmote Sky* nodes (see Figure 16). Those sensor nodes run a Texas Instruments MSP430 microcontroller at 8 MHz. They contain 10 kB of RAM and 48 kB of flash. Every node can reach its neighbours without packet loss but radio interferences still occur. We use a single DODAG (see Section 1.6.1) in which nodes periodically send data to the *root*. When simulating broken or malicious nodes, we disable the forwarding of data packets. However, they still send out routing informations (i.e. DIOs), so, they remain inside the network. Indeed, routing information may be stale or wrong since RPL does not validate this information.

[1]According to RFC 2460 [147] every IPv6 link has to support a MTU of at least 1280.

To compare our approach with state-of-the-art RPL, we consider three experiments: First, we simulated an undisturbed case to compare our approach to standard RPL. This will be considered as the reference experiment later on. Second, we construct an experiment with one attacking node next to *root* (Experiment A). Only three nodes can reach *root* directly and one of those is malicious (see Figure 17a). To increase the attack, we provide a third experiment with two attackers next to *root* (Experiment B). They are two out of four nodes with direct link to *root* (see Figure 17b).

4.1.5. Undisturbed Case

In the undisturbed experiment (setup is the same as in Experiment A but without the attacker), we compare RAM, ROM and DIO size between our approach and the reference RPL implementation in *Contiki*. ROM and RAM usage is measured statically for the *Tmote Sky* build (see Figure 18a). Our implementation uses about 1.2 kB of additional ROM, which brings this particular sensor node to its limit. RAM is allocated at compile time and the usage increases by 1.6 kB, which is surprisingly high because we only send very few additional bytes. The reason for this increase it that a *Contiki* node stores the last received DIO for every direct neighbour in RAM – even for neighbours which are not currently parents. Unfortunately, *Contiki* preallocates memory for every possible neighbour (maximal neighbour count is configurable; 20 is default for *Tmote Sky*) and we increased the size of DIOs by 65 bytes. However, a node only needs to remember the latest metric container and the memory usage could be reduced by optimising DIO storage.

We also compared the packet delivery rate between reference and our implementation. As shown in Figure 18b, both behave similarly. Minimal packet loss is happening in both experiments and the delivery rate is about 99%. However, in our implementation, the delivery rate is slightly lower in simulation due to higher CPU usage and more radio interference due to bigger DIOs. In [C10], we presented an approach to reduce the size of the DIOs to prevent this effect.

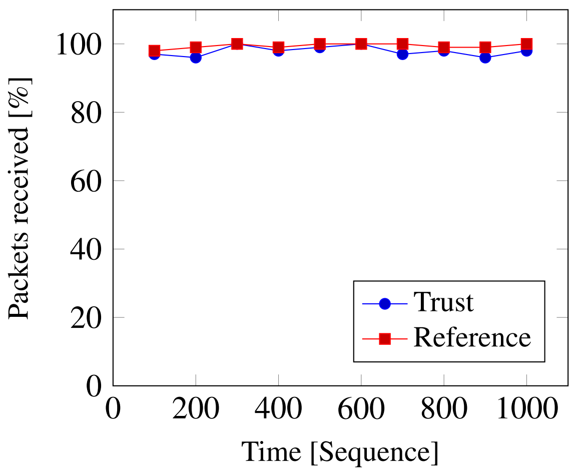
4.1.6. Under Attack

To evaluate our approach under attack, we added one (Experiment A) or two (Experiment B) malicious nodes to the network (see layout in Figure 16). Those malicious nodes do not forward any data packets. However, they still send out RPL routing packets, so, other nodes still may use them as parents. For both attacks, we compare standard RPL with *rank* as metric

Figure 17: System in undisturbed case. The setup is the same as in Experiment A but without the attacker. Our approach behaves similar to the reference experiment.

	Trust	Reference	Δ
ROM	48796 B	47520 B	1276 B
RAM	9516 B	7834 B	1682 B
Metric	65 B	0 B	65 B

(a) Additional Memory Usage



(b) Undisturbed system

and our implementation with trust as metric. We placed attackers next to *root* to maximise their impact on the system.

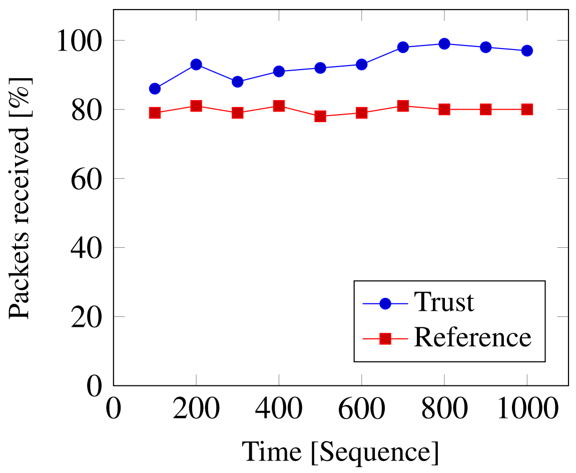
In Experiment A with one attacker, the reference implementation only achieves a delivery rate of constant 79% (see Figures 17a and 19a). This effect is caused by a single attacker which is one out of three nodes which can directly reach *root*. In contrast, the performance of our trust-enhanced RPL improves over time: At the beginning, it reaches 86% PDR, and eventually, it recovers to over 99% PDR.

We add a second attacker in Experiment B and repeat our measurements (see Figures 17b and 19b). In the reference experiment, the PDR dropped to an average of 38.5%. About 60% of the nodes are unable to communicate with *root*. In comparison, our approach reaches a PDR of about 70% initially. After about 400 packets, it improves to nearly 100%. The improvement happens quite abrupt because RPL requires nodes to change their *rank* if they do not have any suitable parents. However, the *rank* also influences the topology of the network and, thereby, the attackers move out of the routes for most nodes.

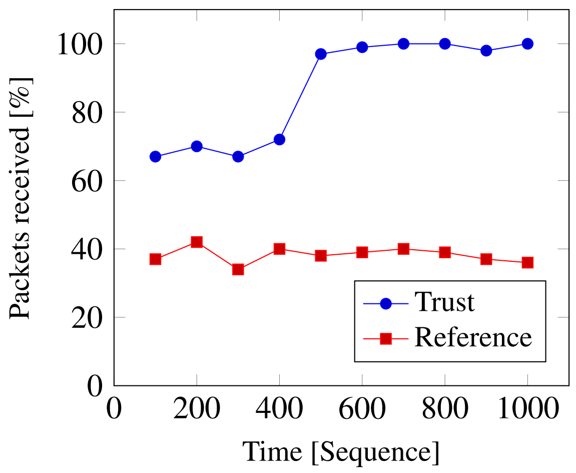
4.1.7. Minimising Transmissions and Energy Consumption

Focusing only on selecting optimal routes and maximise the PDR has one downside when attacks or disturbances occur only temporarily because nodes will keep a longer route after attacks (see [W29]). This causes additional radio transmission and, thereby, more energy usage

Figure 18: System under attack.



(a) Experiment A: One attacking node



(b) Experiment B: Two attacking nodes

which is problematic on battery-powered sensor nodes. We account this overhead indirectly because sending additional packets causes more collisions and therefore influences the PDR. However, to account for minimal energy consumption, we add a second utility function: We want to minimise the number of transmitted packets (#TX). However, maintaining a good PDR is still a higher priority[2].

In Figure 19, we show #TX before, during and after a temporary blackhole attack. In the beginning, our approach (trust + ETX) chooses slightly longer paths and, therefore, uses on average about five more packets than the reference metrics OF0 according to RFC 6552 [145] and Expected Transmission count (ETX) [149]. PDR is nearly 100% for all metrics (see Figure 20). When the attack starts, the number of packets drops because some of them are dropped by the attackers. PDR drops to about 50% for all metrics. ETX and OF0 stay at that level. Our trust metric recovers to nearly 100% within 100 sequences. However, the number of packets sent increases by about 40%. After the attack ends, OF0 and ETX recover PDR and #TX to the old level. Unfortunately, #TX for our metric stays at the same high level as during the attack because the newly chosen routes still work well. However, the old routes are available again and should be used.

[2]Dropping all packets at all nodes would result in a minimal number of packets transmitted but also in a very low PDR.

4.1.8. Second-Chance

The trust + ETX metric considers both utility functions. It selects a parent with a good ETX (resulting in a low number of Transmitted Packets (#TX)) from the list of trustworthy neighbours (to maintain a high PDR). However, when the best route is not in the list of trustworthy neighbours, it will never be considered again as long as the list is not empty. Therefore, we implement the social concept of forgiveness [129] to give nodes a second chance. Otherwise, a node will never explore other paths as long as the current selected parent is good enough. However, in a dynamically changing environment such as a sensor network, this will cause problems when disturbances occur. Therefore, we employ the NM (located on the *root* node) to monitor the network and only enable this measure selectively.

Our approach to implement forgiveness is called trust + ETX + *Second-Chance*. When a node has a parent which delivers a PDR of 100% and another node with a better ETX exists (58), it will route one or two packets through that node (see Equation (59)). If the PDR of that TR is 100% again, the alternative node will receive a good rating and may be considered as parent again. If PDR is less than 100%, no action will be taken and the node may try again later.

$$\begin{aligned} S(n|r) :=& s, s \in N(n) \wedge \mathtt{ETX}(n, s|r-1) \\ =& \max_{n_2 \in N(n)} \mathtt{ETX}(n, n_2|r-1) \end{aligned} \tag{58}$$

$$T(n,p|r) := \begin{cases} \alpha \cdot T(n,p|r-1) + (1-\alpha) \cdot \mathrm{PDR} & p = P(n|r-1) \\ \alpha \cdot T(n,p|r-1) + (1-\alpha) & p = S(n|r) \wedge \mathrm{PDR} = 1 \\ T(n,p|r-1) & \text{otherwise} \end{cases} \tag{59}$$

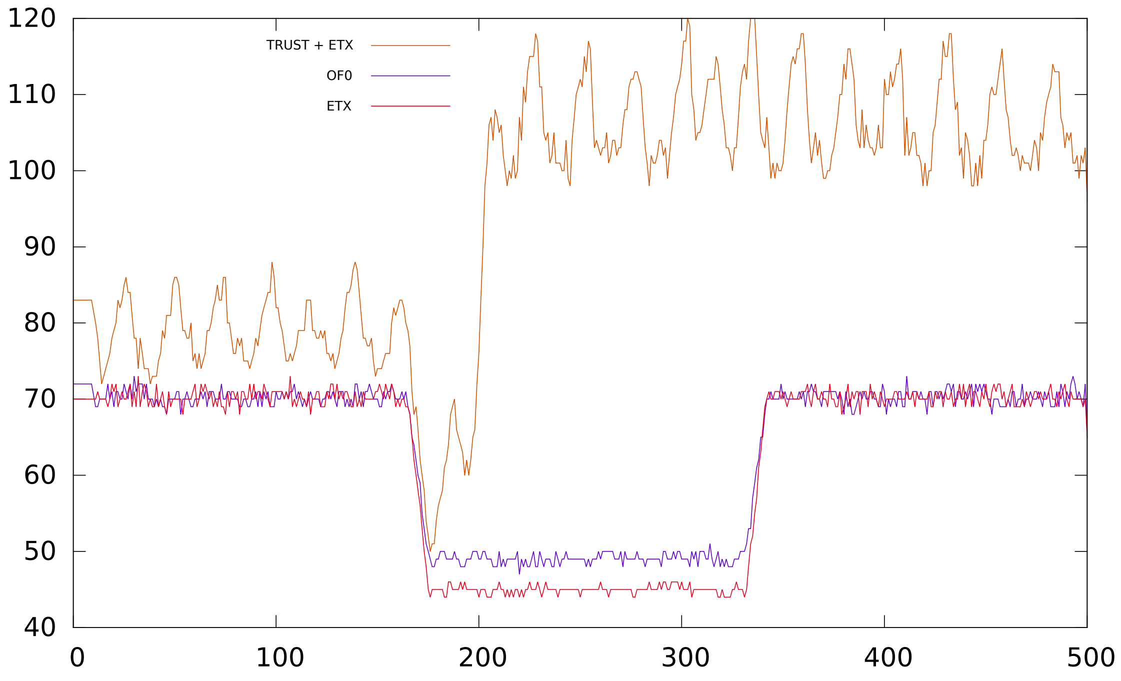
This approach allows optimising the route, when the current parent is already very reliable. Therefore, this is only triggered when the NM observes a stable PDR.

We evaluate *Second-Chance* as metric against our previous approach. Additionally, we compare to OF0 and ETX as baseline.

4.1.9. Norm Manager

Constant retrying/forgiveness causes additional overhead and, thus, is only beneficial during an attack (as discussed in [O40]). However, because of limited knowledge, nodes cannot

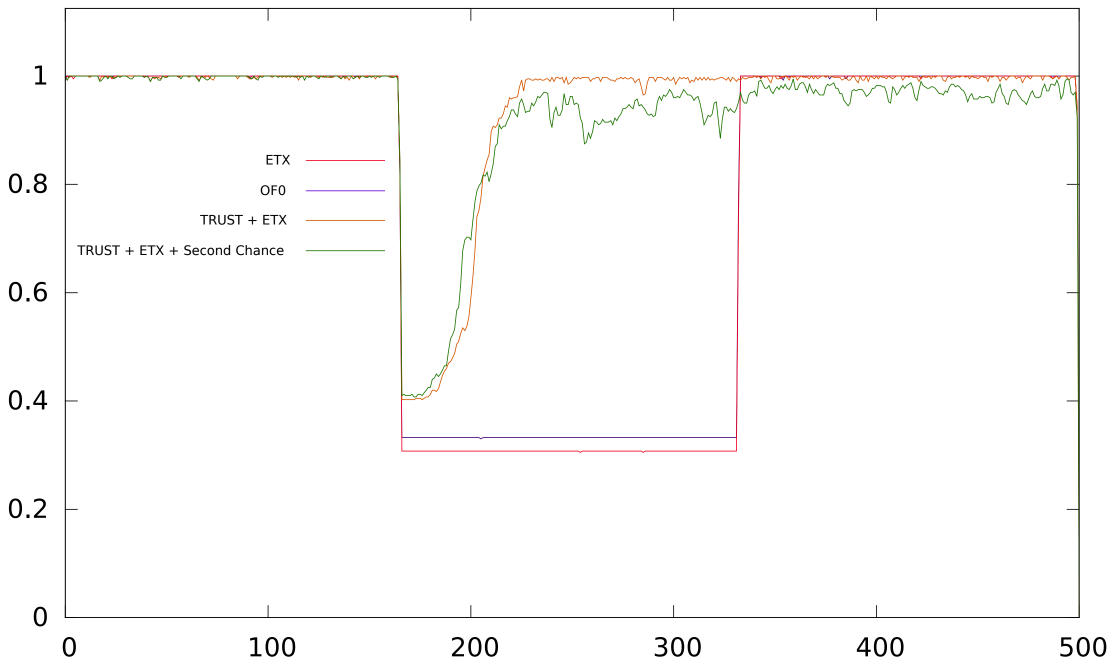
Figure 19: Transmitted Packets (#TX) over sequences (time). An attack starts at sequence 160 and ends at sequence 340. During the attack, #TX increases for the trust + ETX metric and stays at that level after the attack. For OF0 and ETX, #TX drops during the attack and recovers to the same level as before afterwards.



detect this in a timely manner. To solve this challenge, we introduce a higher-level NM at *root* which analyses the situation and sends advice to nodes in the network (see Figure 9). In WSN, the *root* node typically is not battery powered and has significant computing and memory resources which allow additional processing. Based on our work in [C19, W29], it can also obtain an outline about the structure and disturbances occurring. By exploiting this information, the *root* node can detect and assess NEB. Unfortunately, it cannot react directly to malicious behaviour or prevent any attacks. However, it can issue *norms* to influence agents indirectly which are distributed using DIO messages.

Thereby, nodes remain autonomous and still can make decisions locally but get guidance from the NM to allow faster and more informed responses to attacks. In this experiment, we adjust the rate to retry (i.e. forgive) a parent. This should be low during an attack to prevent packet loss but high when the attack ended to recover the paths.

Figure 20: Packet Delivery Rate (PDR) over sequences (time). An attack starts at sequence 160 and ends at sequence 340. PDR drops to about 30% for OF0 and ETX during the attack and recovers afterwards. Trust + ETX and trust + ETX + *Second-Chance* recover during the attack and stay near 100% PDR until the end of the experiment.



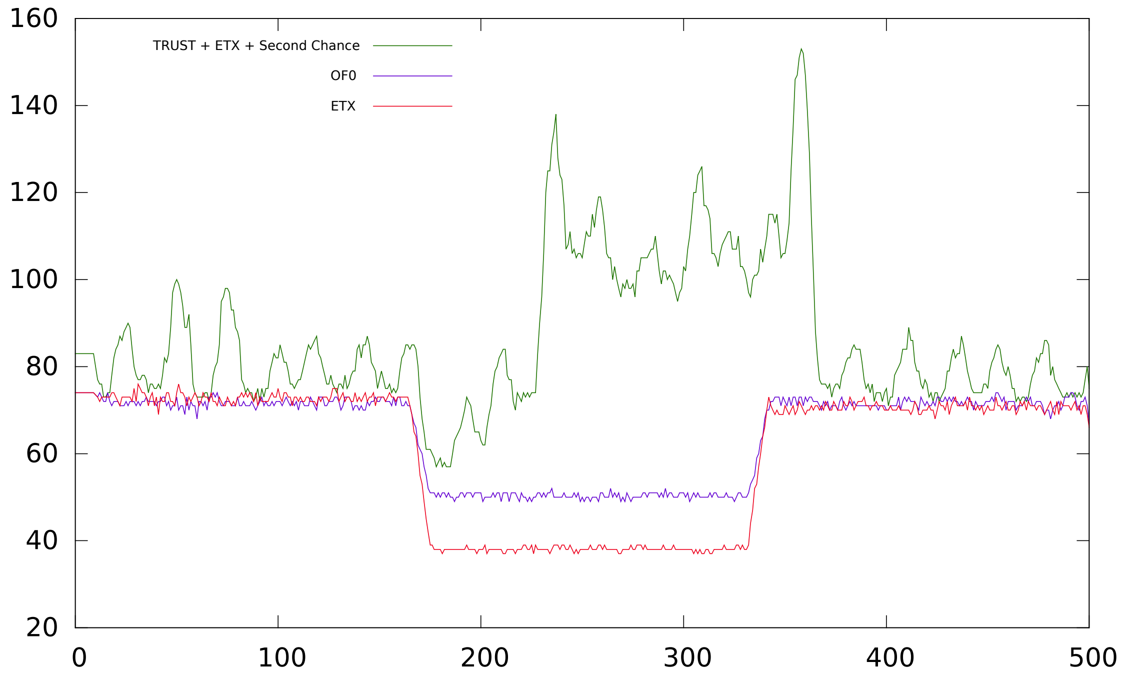
4.1.10. Experiment Settings

To evaluate our approach, we perform a simulation in *Cooja* [148] running *Contiki* nodes [144]. Each run is repeated 20 times and the result values are averaged. Each node sends 5.000 packets to *root* during the experiment. The attack starts at sequence 160 and ends at sequence 330.

4.1.11. Results and Discussion

In Figure 21, we show our results. In an undisturbed system, trust + ETX + *Second-Chance* chooses very similar routes to ETX and #TX is slightly higher. When the attack starts at sequence 160, #TX decreases for all metrics since attackers start to drop packets. As shown in Figure 20, OF0 and ETX do not react to the attack and show a PDR of only 30%. Trust + ETX + *Second-Chance* reacts and recovers to a PDR of > 95% at sequence 220 (see Figure 19). When the attack ends, #TX for trust + ETX + *Second-Chance* decreases back to its initial level of about 80 while trust + ETX (without *Second-Chance*) remains at about 107.

Figure 21: Transmitted Packets (#TX) over sequences (time). An attack starts at sequence 160 and ends at sequence 340. During the attack #TX increases for the trust + ETX + *Second-Chance* metric. For OF0 and ETX #TX drops during the attack and recovers to the same level as before afterwards. Unlike in Figure 19, trust + ETX + *Second-Chance* recovers to a level similar to before the attack.



Using the NM and adding *Second-Chance* to our metric when the disturbance ends allows self-improvement during runtime. Since the approach is only activated by nodes when they observe a very high primary utility (PDR), it does not influence the behaviour during the recovery of attacks. The NM detect the end of the disturbance around sequence 350 and #TX recovery at sequence 360. If we assume that attacks only occur occasionally and that the system is undisturbed most of the time, our approach is clearly beneficial. The overhead during the attack (after recovery) is already small, but it can be decreased further. However, this experiment shows that our approach is working for this scenario.

As a future improvement, the NM could also advise on the frequency of retries because we currently retry in every TR with a constant frequency. To improve this, we could also implement exponential back-off or similar methods. Additionally, nodes could passively listen to the radio to check whether their parent node is actually forwarding messages as suggested in [117]. However, since this is an energy-intense operation, this should only be done selectively when advised be the NM.

4.2. Trusted Desktop Grid

In this section, we apply our approach to the TDG scenario (see Section 1.6). We perform an experiment with a persistent attack by colluding *Cunning Agents* and show how the NM identifies and isolates those. Additionally, we evaluate how the NM can tune the integration of new agents which can be used to allow faster integration of *Adaptive Agents* or faster isolation of *Freeriders* and *Egoists*.

4.2.1. Detecting and Isolating Cunning Agents

Cunning Agents cannot be easily detected and isolated locally in a distributed system. Therefore, they pose a permanent threat to the TDG. However, a higher-level NM can detect them and then can change norms to isolate them.

To detect *Cunning Agents* in the NM we cluster groups using the MCL [139] and the BIRCH [142] algorithms on the *trust graph* and classify them using a decision matrix. For the purpose of this experiment, the matrix can only identify groups of *Cunning Agents*. The metrics used for that purpose are shown in Section 4.3.7. In particular, we use the value of *Authorities* and *DegreeCentrality* to detect groups of *Cunning Agents*.

Once the NM detects groups of *Cunning Agents*, it has to isolate them. Unfortunately, it cannot influence agents directly but it can do this using norms. Therefore, it introduces a new norm which allows agents to refuse work from *Cunning Agents*. We propose to detect them based on their inconsistent behaviour.

4.2.2. Inconsistent Behaviour

In trust-based distributed systems such as the TDG, we condense a series of trust ratings R to a single reputation value τ (see Equation (61)). However, this does not take the consistency of those ratings into account. Normally, agents make their decisions based on the aggregated value τ.

$$r \in [-1, 1] \Rightarrow R \in [-1, 1]^k \tag{60}$$

$$\tau := \mathfrak{T}(R) := \frac{\sum_{r \in R} r}{\sum_{r \in R} |r|} \tag{61}$$

However, *Cunning Agents* behave strategically regarding their reputation value: They behave well until they reach a certain threshold τ_{upper} and then stop to cooperate with other agents until they fall below a threshold τ_{lower}. Therefore, their reputation τ is between those two values most of the time and they adjust their τ_{lower} and τ_{upper} to be considered as well-behaving by other agents based on the reputation (see Equation (62)).

$$\tau_{\text{lower}} \leq \tau \leq \tau_{\text{upper}} \wedge \tau_{\text{lower}} \geq \tau_{\text{wb}} \tag{62}$$

Since those agents intentionally exploit the trust metric, they cannot get detected and isolated using the reputation value. However, they receive very inconsistent ratings r because of their changing behaviour and we can leverage that to detect them. Therefore, we define the consistency κ based on the standard deviation:

$$\kappa := 1 - \frac{\sum_{r \in R, r>0} r}{\sum_{r \in R} |r|} \cdot (\tau - 1)^2 + \frac{\sum_{r \in R, r<0} r}{\sum_{r \in R} |r|} \cdot (\tau + 1)^2 \tag{63}$$

We expect a very low value for *Cunning Agents* and a value close to one for *Adaptive Agents*. Other agents such as *Freeriders* or *Egoists* also should get a value of approximately one since they consistently behave maliciously.

4.2.3. Changing Norms

The NM cannot directly influence agents or force them to perform any actions. However, it can change norms and, thereby, change incentives and sanctions for certain actions. To cope with *Cunning Agents* while maintaining the autonomy of agents we choose to allow them to reject jobs from inconsistently behaving agents. Therefore, agents can decide on their own if they want to work for *Cunning Agents* since they will still receive an incentive for that.

In Figure 22, we show the changed norm. The threshold δ for the reputation τ is smaller than τ_{lower} and the threshold γ for consistency is 0.8 in our experiment. `requester.reputation` is calculated using the trust metric τ and `requester.consistency` is determined by κ. The sanction results in a rating in R for the working agent.

4.2.4. Setup

The setup in the evaluation consists of 100 *Adaptive Agents* and additional 50 *Cunning Agents* for all experiments with attackers. Each experiment ran for 300,000 ticks and was repeated 100 times. We performed three series of experiments: (`E1`) without attackers; (`E2`) with

Figure 22: Changed norm used in the evaluation to isolate *Cunning Agents*. It allows agents to reject jobs from inconsistently behaving agents.

$$
\begin{aligned}
&\textbf{context}\ \overbrace{\text{Worker}}^{\text{Target/Role}}\ \textbf{norm}\ \overbrace{\text{AcceptJob}}^{\text{Name}}:\\
&\qquad\quad \underbrace{\begin{aligned}&\texttt{requester.consisteny} > \gamma \wedge\\ &\texttt{requester.reputation} > \delta\end{aligned}}_{\text{Pertinence Condition}}\\
&\quad\textbf{implies}\ \ \underbrace{\texttt{acceptJob}(\texttt{requester}, \texttt{job}) = \texttt{true}}_{\text{Postcondition}}\\
&\textbf{sanctioned}\\
&\quad\textbf{if}\quad \underbrace{\text{violated}}_{\text{Default Sanction Condition}}\\
&\qquad\textbf{then}\ \ \underbrace{\text{self.reputation += -rejectSanction}}_{\text{Sanction}}
\end{aligned}
$$

attackers; (E3) with attackers and our norm changes.

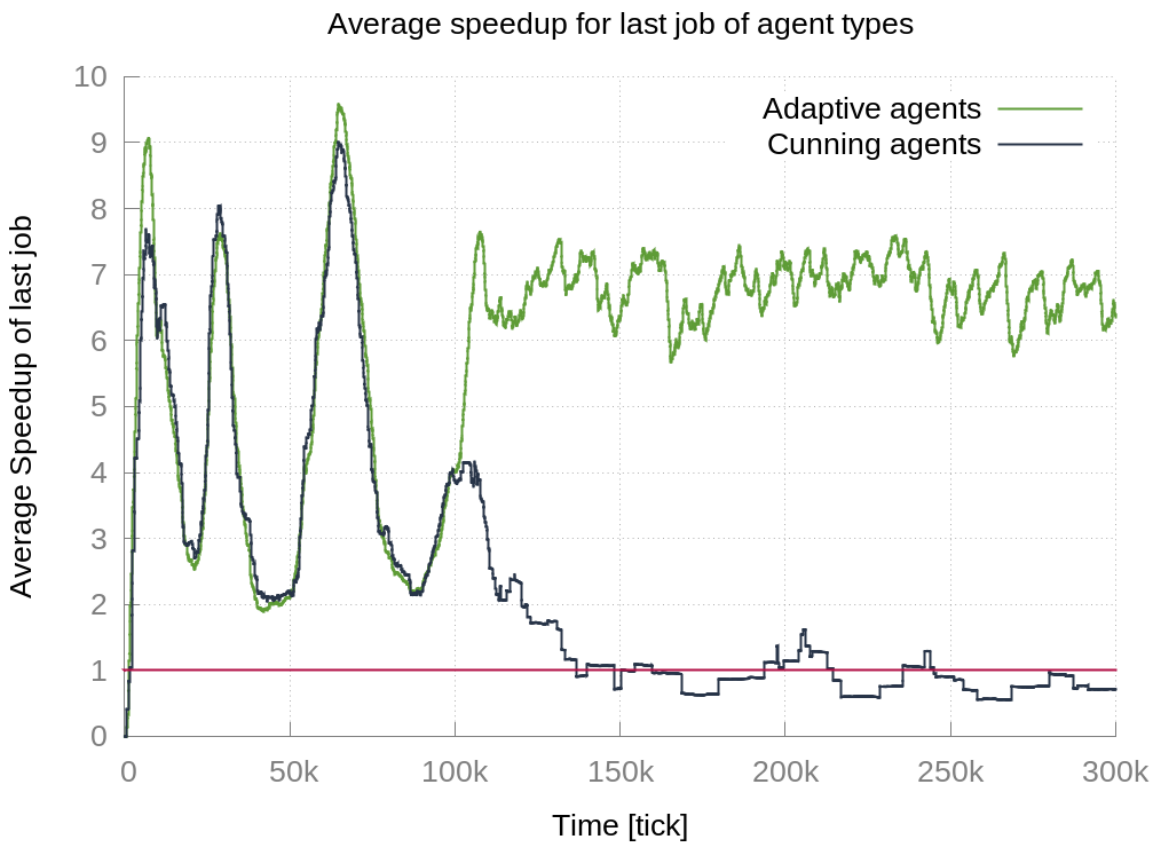
4.2.5. Detection of Cunning Agents

The detection of *Cunning Agents* in (E2) by the NM can be observed between tick 15,000 and 50,000 and was successful in all experiments. On average, the NM detected the attackers at tick 24,920 with a standard deviation of 15,301. The NM introduces the norm at this point. However, to make the impact of the norm change more comparable, we set the time t_{change} to 100,000 for subsequent experiment in (E3).

4.2.6. Consistency Values

We measured the consistency values for *Adaptive Agents* and *Cunning Agents* in (E2) at the end of the experiment. Both values turned out to be as expected: *Adaptive Agents* have a value of 0.9999 ± 0.00053 which is very close to one. In contrast, *Cunning Agents* have a value of 0.0611 ± 0.01337 which is next to zero.

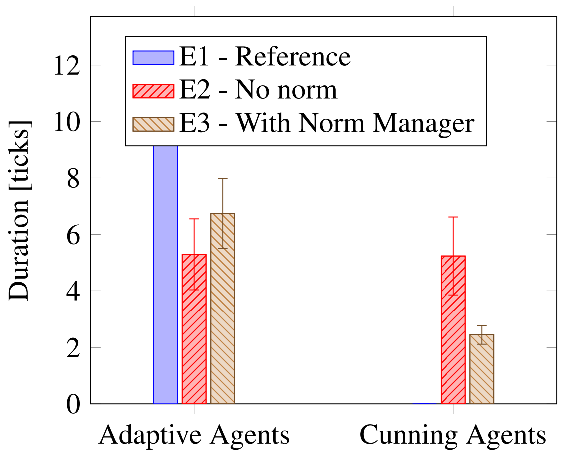
Figure 23: Exemplary simulation run with norm introduced at tick 100k by the Norm Manager. *Cunning Agents* get isolated quickly and *Adaptive Agents* recover to an utility of about 7. A speedup of less than one means that agents no longer have an advantage from participating in the system.



4.2.7. Influence of Norm Changes

We measure the influence of the norm change using the speedup σ for *Adaptive Agents* and *Cunning Agents* in experiments (E1), (E2) and (E3) (see Figure 24). In Figure 23, we show one single exemplary experiment of (E3) with speedup over time. In the beginning, *Adaptive Agents* and *Cunning Agents* achieve a similar but varying speedup. After the NM introduced the norm change at tick 100,000 the speedup for *Cunning Agents* falls below one and those agents no longer gain any advantage from participating in the system. In contrast, *Adaptive Agents* gain a stable high speedup again. In the undisturbed experiment (E1), we measured that *Adaptive Agents* can achieve a speedup of 11.74 ± 0.73 when there is no attack. When a heavy attack of *Cunning Agents* is added in (E2), the speedup decreases to 5.29 ± 1.26. With norm change by the NM, the speedup increases again to 6.75 ± 1.24 in (E3) which is less than

Figure 24: Speedup for *Adaptive Agents* and *Cunning Agents* for every experiment series with 100 Experiments each. In `(E3)` the Norm Manager is active. All experiments lasted 300,000 ticks.



at the end of the experiment (see Figure 23) because the value is averaged over the complete experiment. *Cunning Agents* achieve a speedup of 5.23 ± 1.38 when they exploit the system in (`E1`). At the same time, they work significantly less than *Adaptive Agents*. However, when the NM changes the norm, the speedup of *Cunning Agents* decreases to 2.49 ± 0.33. Again, this is averaged over 300,000 ticks and, as shown in Figure 23, the *Cunning Agents* are isolated with a speedup below one at the end of all experiments.

4.2.8. Integration of New Agents

In Figure 25 and Table 1, we show another experiment where the NM modifies the value of δ (see Figure 22) which influences how well new agents are integrated. If an agent has a reputation below δ, other agents are free to reject a job without sanctions. Since they still receive an incentive for completing jobs, they might still do the work in some cases, but can decide autonomously. Usually, δ is set to the initial reputation of new agents, but the Norm Manager can change δ depending on the situation. When the system is under disturbance δ is increased and may be reduced afterwards.

We consider attacks by *Freeriders* and *Egoists* and evaluate both attacks by adding 100 attacker agents to a system of 200 *Adaptive Agents*. To measure the effect on well-behaving agents we repeat the experiment with 100 *Adaptive Agents* entering the system. The described norm change is performed at the beginning of the attack. Additionally, we run a reference experiment without norm change for all agent types. Since isolation and integration of agents is slower during low load situations, we added this as a scenario. Every experiment is repeated

Figure 25: Duration of isolation/integration per agent group. Generally a lower duration is better. For *Adaptive Agents*, we measure how fast they get integrated and for *Freeriders* and *Egoists* how fast they get isolated.

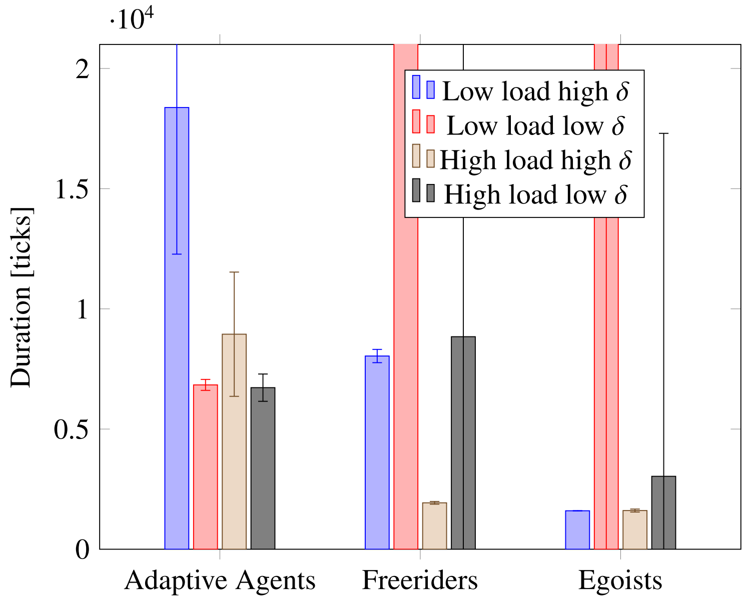


Table 1: Duration of isolation/integration per agent group. Generally less is better. For *Adaptive Agents*, we measure how fast they get integrated and for *Freeriders* and *Egoists* how fast they get isolated.

Agent	Low Load high δ	High Load low δ	Low Load low δ	High Load high δ
Adaptive Agents	6837.1±228.06	6722.1±568.1	18375.3±6099	8945.7±2585
Freerider	145000±0	8841.6±17597	8037.9±275.77	1930.8±57.37
Egoists	41178.4±64102	3034.1±14268	1600±0	1609.3±62.47

one thousand times, resulting in 12,000 experiments.

After the attack starts at t_z, we periodically calculate the speedup σ (defined in Equation (1)) for the attacking agents. $t_{\text{end,isolation}}$ is defined to be the smallest value with $t_{\text{end}} > t_z \wedge \sigma \leq 1$ (64). The duration of isolation $T_{\text{isolation}}$ is then determined as the difference of t_{end} and t_{start} (65; see Section 3.5 for more details).

$$t_{\text{end,isolation}} := \min\{t : t > t_z \wedge \sigma \leq 1\} \tag{64}$$

$$T_{\text{isolation}} := t_{\text{end,isolation}} - t_z \tag{65}$$

For *Adaptive Agents*, we similarly calculate the duration of integration $\delta_{\text{integration}}$ (67). In a reference experiment without norm change, we determine the final speedup after integration σ_{ref}. $T_{\text{end,integration}}$ is then defined to be the first time after attack where $\sigma \geq \sigma_{\text{acc}}$ (66).

$$t_{\text{end,integration}} := \min\{t : t > t_z \land \sigma \geq \sigma_{\text{acc}}\} \tag{66}$$

$$T_{\text{integration}} := t_{\text{end,integration}} - t_z \tag{67}$$

In Figure 25, we present our results for the three agent types. For *Freeriders* and *Egoists*, the graph shows $T_{\text{isolation}}$. In contrast, for *Adaptive Agents*, it illustrates $T_{\text{integration}}$. Full results with standard deviation are listed in Table 1. The results show that isolation of malicious agents greatly improved when norms were changed, especially in low-load situations. For *Freeriders*, the duration decreased by 78% under normal load. Under low load, *Freeriders* were not fully isolated before. However, this changed with our approach: The system did properly isolate the attackers in all experiments. Since isolation did not work in the reference case, we limited the length of that experiment. Therefore, the value for low load without norms in Table 1 has no variance at all and the relative gain cannot be calculated.

Our approach is very effective when dealing with *Egoists*. With changed norms during attack, they get isolated after calculating their first job (duration of a job is 1600 ticks). Without the change, they were not isolated in most cases under low load and it took about twice as long under normal load. However, well-behaving agents are also affected by the norm change: *Adaptive Agents* need 33% longer under normal load and 169% longer under low load. Integration still worked in all experiments and can be considered stable. Our results show that changing norms reduces the impact of attacks by *Freeriders* and *Egoists*. However, this change cannot become the default because it also affects the integration of well-behaving agents. Nevertheless, by using our NM we can change norms when the observer detects an attack by *Freeriders* or *Egoists*. Critical for the success of this method is fast detection of such attacks. However, it can be also useful when the system is in a critical state to prevent further disturbances. After isolation of the attackers, the norm changes can be reversed since isolation of those two groups remains permanently in the TDG. Isolation is performed using trust and reputation mechanism of the TDG. We chose this approach to keep maximal autonomy for the agents (see [C18] for more details).

4.2.9. Discussion and Summary

In this section, we applied our approach to the TDG scenario to demonstrate its effectiveness. First, we showed an experiment with *Cunning Agents* which pose a permanent threat to the system stability. Using the NM which applied a norm to encourage consistent behaviour, those attackers were isolated and the utility returned to a reasonable level. However, in non-disturbed cases this measure causes monitoring overhead and it is, therefore, not well suited to be always active. Nevertheless, with our approach it is only activated on demand.

The second experiment focused on sealing the system from new agents when the system is either already in a critical state (e.g. in *trust breakdown*) or when attackers were detected. Using the δ threshold in the norm, sanctions for rejecting jobs from newcomers are effectively disabled but incentives for performing work stay in place. This gives agents more freedom to make better decisions when under pressure (e.g. during an attack or disturbance). Thus, attackers get isolated faster and cannot disturb the system further. However, this also reduces the incentives to work for other new cooperative agents and it takes longer for those to be integrated in the system. Therefore, this should be also enabled or disabled depending on the situation by the NM.

4.3. Rendering

In this section, we evaluate distributed rendering as application on top of the TDG and compare it to BURP (see Section 1.6.3). BURP is running on top of BOINC (see Section 2.1.1) which distributes work using a central tracker. To make the TDG comparable, we limit it to a single submitter[3]. We compare the success of the approach used in the TDG and BURP by experimenting with four different scenarios and analysing the achieved results with a focus on correctness and throughput as utility functions.

4.3.1. Distribution Strategies

First, we explore different distribution strategies which select (i) the replication factor, and (ii) the worker for a given work unit. Work units are accepted by the submitter when the majority of returned results matches (quorum). The NM can select the strategy and the corresponding parameters.

[3]We also evaluated the TDG with multiple submitters and rendering as application in [C16].

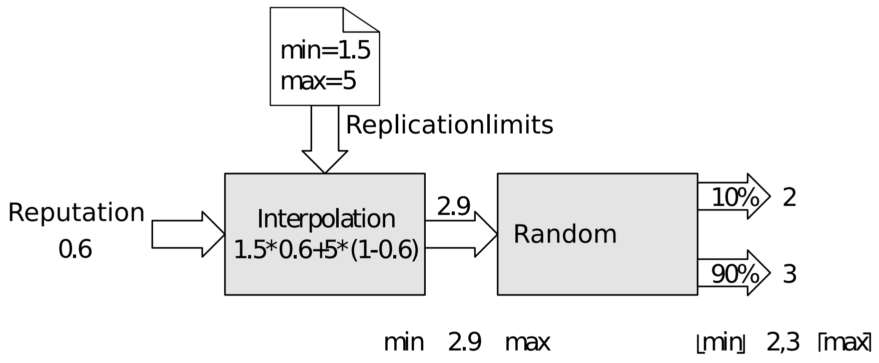
4.3.2. Static Random Distribution Strategy (BURP)

The strategy used by BURP is a static random distribution strategy with a static factor f of 2. Workers are chosen randomly without any consideration of reputation or trust. When the returned results do not match, BURP discards them and chooses two new workers randomly. Therefore, the practical replication factor can be larger than 2.

4.3.3. Dynamic Random Distribution Strategy (DRDS)

As a baseline we developed Dynamic Random Distribution Strategy (DRDS) which uses the reputation to calculate the minimum reputation factor f_{min} for each worker. It defines limits (in our case between 1.5 and 5.0) and interpolates using the reputation $\tau(R)$ of the worker. Afterwards, f_{min} is rounded to the next integer using a roulette wheel random generator (see Figure 26).

Figure 26: We calculate the minimum replication factor f_{min} for each worker. First, we interpolate f_{min} based on the reputation τ between defined limits (here from the interval $[1.5, 5]$). Then, we round f_{min} to the next integer using a roulette wheel random generator.



DRDS starts with a random worker and continues to add workers until the replication requirement for all of them is satisfied. Effectively, the largest f_{min} is used for all workers even if most of them have much lower requirements. Therefore, we expect this pessimistic strategy to show a very high correctness but slightly reduced throughput.

4.3.4. Dynamic Ordered Distribution Strategy (DODS)

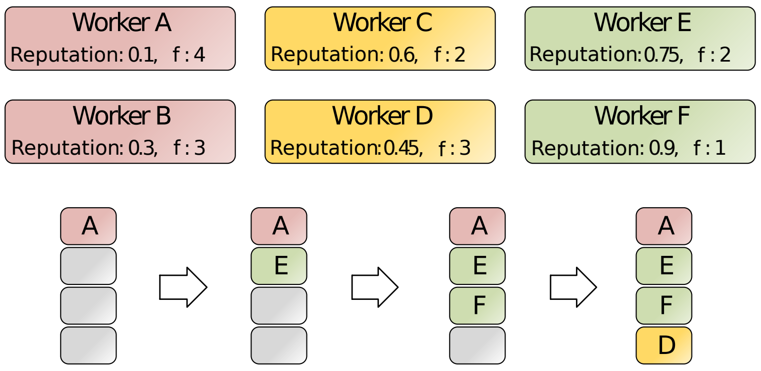
To optimise the replication factor f, in Dynamic Ordered Distribution Strategy (DODS) we order workers by their replication requirements f_{min} (which roughly also orders them by their reputation τ). Afterwards, the top $f_{\min}$ workers for the first work unit are selected. The next $f_{\min}$ is applied for the second group etc. Since $f_{\min}$ depends on the worker, the group size increases along the list.

DODS finds a solution for the minimal number of replicas. However, when selecting a group of workers with very low reputation, it is very likely that the majority does not return the correct results and no quorum can be reached. This will trigger additional replication or may result in incorrect results.

4.3.5. Dynamic Grouping Distribution Strategy (DGDS)

To prevent potential problems with quorum, we developed Dynamic Grouping Distribution Strategy (DGDS) which selects groups using heuristics. The goal of the strategy is to choose at least as much "trusted" ($\tau > 0.7$) as "untrusted" ($\tau \leq 0.4$) or "undecided" ($0.4 < \tau \leq 0.7$) workers. At the same time, the replication requirement $f_{\min}$ for the group is minimised.

Figure 27: DGDS selects one "untrusted" ($\tau \leq 0.4$) worker in the first step. Afterwards, it adds two "trusted" ($\tau > 0.7$; green) workers and one "undecided" ($0.4 < \tau \leq 0.7$; yellow) worker to make sure the "untrusted" workers ($\tau \leq 0.4$; red) do not form the majority.



DGDS starts by selecting "untrusted" workers with similar replication requirements and adds up to $\lfloor \frac{f_{\min}-1}{2} \rfloor$ entities of this category to the group. Then, it adds the same amount of "trusted" workers. Afterwards, it fills the group with "undecided" and "trusted" to reach $f_{\min}$ (Figure 27 shows an example). If such combinations cannot be found, it also selects less desirable groups as long as "untrusted" workers cannot form the majority. Depending on the current situation, the NM sets the replication limits (as shown in Figure 26).

4.3.6. Experimental Setup

We run our experiments on a cluster with 17 nodes and 204 CPU cores. 100 workers render frames for one submitter (to make it comparable to BURP; TDG supports multiple submitters). In each experiment, we render 30 films with 1,000 frames each. To render the experiments fast enough, every frame was reduced to require only between 1 and 1.5s of computing time. Every worker only works at one frame at a time. Every experiment is repeated ten times and the results are averaged (with standard deviation). For DGDS, we set the replication limits to 1.1 and 3 (chosen to be around 2 which BURP uses; other limits are investigated in Experiment 3).

4.3.7. Metrics

During the evaluation, we measure the following metrics:

- **Throughput**: Completed and accepted work units. Measured at the submitter in work units per second.
- **Correctness**: Measures the correctness of completed and accepted work units in percent. Validation is performed using prior knowledge.
- **Replication Factor**: The average replication factor f among all workers. Replication factor one means no replication. Only one worker computes the work unit and no overhead is generated. With factor two, every frame is rendered twice which results in 100% additional overhead.
- **Reputation**: Average reputation of an agent group based on the reputation system as introduced in Section 3.2.

4.3.8. Worker and Attacker Model

In our experiments, we use four types of worker implementations.

- **Faithful Worker**: The faithful worker behaves well and usually returns all work units correctly. This type is the desired benevolent interaction partner.
- **Incorrect Worker Independent**: This worker is faulty or malicious and returns incorrect results all the time. The incorrect results of multiple workers look different.
- **Incorrect Worker Colluding**: This worker is malicious and returns incorrect results all the time. However, the incorrect results of multiple workers look the same. Therefore,

Table 2: Throughput in work units per second for all distribution strategies with (i) no attackers, (ii) 20% independent incorrect workers and (iii) 20% colluding incorrect workers.

Strategy	no attackers	20% independent	20% colluding
BURP	39.88 ± 0.03	26.65 ± 0.08	27.92 ± 0.10
DRDS	68.89 ± 0.12	44.23 ± 0.11	46.05 ± 0.11
DODS	68.73 ± 0.07	45.30 ± 0.08	57.27 ± 0.29
DGDS	72.23 ± 0.05	56.80 ± 0.10	56.43 ± 0.25

Table 3: Distributed work units per second and worker for (ii) 20% colluding attackers. DODS distributes equally to incorrect and faithful workers (observed using prior knowledge). DGDS distributes significantly more work units to faithful workers and isolates attackers.

Strategy	all	incorrect workers	faithful workers
DODS	0.79 ± 0	0.79 ± 0.002	0.79 ± 0.001
DGDS	0.70 ± 0	0.33 ± 0.024	0.80 ± 0.001

incorrect results may reach a quorum at the submitter and get accepted. This concept resembles intentionally malicious behaviour of a group of cooperating agents.

- **Periodic Worker**: The periodic worker iterates between returning correct and incorrect results. It can be both colluding or independent.

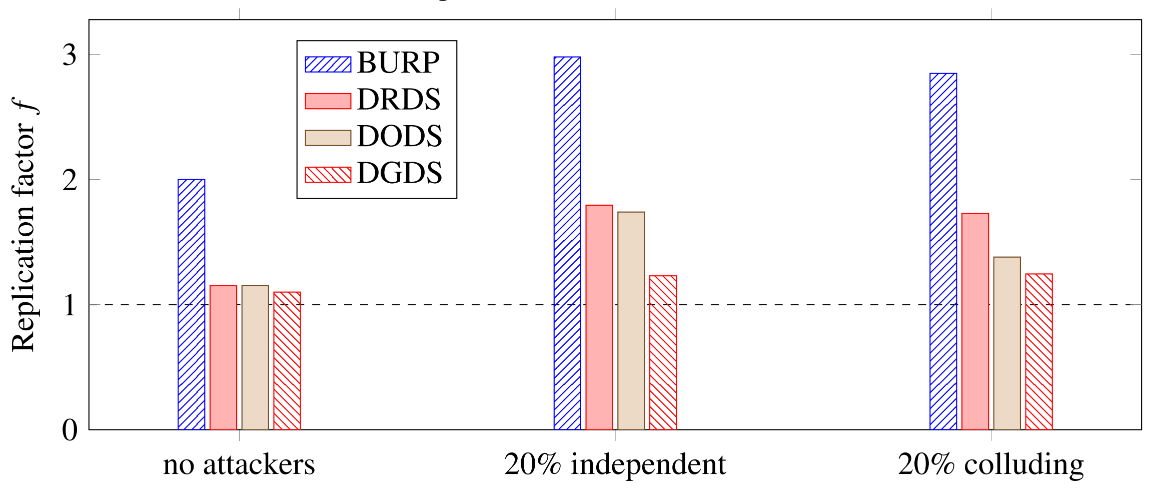
We refer to an attack when incorrect workers or periodic workers are in the system or enter the system. The system always consists of 100 workers and the number of attackers is included (i.e., 20% attackers means 20 incorrect/periodic workers and 80 faithful workers).

4.3.9. Experiment 1: Comparing Distribution Strategies

In the first experiment, we compare our distribution strategies according to the first metric, see Section 4.3.7: the throughput. We consider three scenarios: (i) no attack, based on 100% faithful workers, (ii) 20% independent incorrect workers, and (iii) 20% colluding incorrect workers. We perform ten runs for every strategy and scenario combination.

In Table 2, we show the throughput. Additionally, we show the average replication factor in Figure 28. Without an attack, all our dynamic distribution strategies perform similarly and outperform BURP by about 80%. DGDS achieves a slightly larger throughput and a slightly smaller replication factor. When adding 20% independent attackers, DRDS and DODS outperform BURP by about 90% and DGDS outperforms BURP by over 100%. In both scenarios, all strategies reach 100% correctness. It is notable that BURP shows a replication factor f of 3 which is probably caused by frequent mismatches (about 50% of the time)

Figure 28: Average replication factor f of strategies BURP, DRDS, DODS and DGDS for three scenarios: (i) no attackers (100 faithful workers), (ii) 20% colluding incorrect workers (80 faithful workers, 20 colluding incorrect workers), and (iii) 20% independent incorrect workers (80 faithful workers, 20 independent incorrect workers). Standard deviation is too small to be visible in this experiment.



between the two results. This is further investigated in Experiment 2 (see Section 4.3.10).

However, when adding 20% colluding attackers, DODS and DGDS outperform BURP by about 100% and only DGDS achieves 100% correctness (correctness is not shown in the graph). DODS achieves a slightly higher throughput with a higher replication factor which looks odd at first. However, DODS also does accept incorrect results and does not achieve 100% correctness. In Table 3, we show the average number of distributed work units per second for DODS and DGDS. It can be seen that DGDS distributes less work units to attackers (observed using prior knowledge), which causes the lower replication factor, but also a slightly lower throughput. However, accepting incorrect results to increase the throughput is not desirable and will, thus, be investigated further in Experiment 3 (see Section 4.3.11).

Overall, we conclude that DGDS is the most promising algorithm, since it achieved 100% correctness and the lowest replication factor in all scenarios. Thus, we further investigate the performance of the DGDS algorithm with the NM in the next experiments.

4.3.10. Experiment 2: Independent Attackers

In the second experiment, we investigate the influence of different percentages of independent attackers. An attacker will not return any correct results and the throughput will decrease with increasing percentages of attackers. Therefore, we measure the replication factor which

Figure 29: Average replication factor for BURP and DGDS when attacked by independent incorrect workers. On the x-axis, increasing the percentage of attackers is shown. Standard deviation is too small to be visible in this graph. A lower replication factor is better (i.e. less overhead).

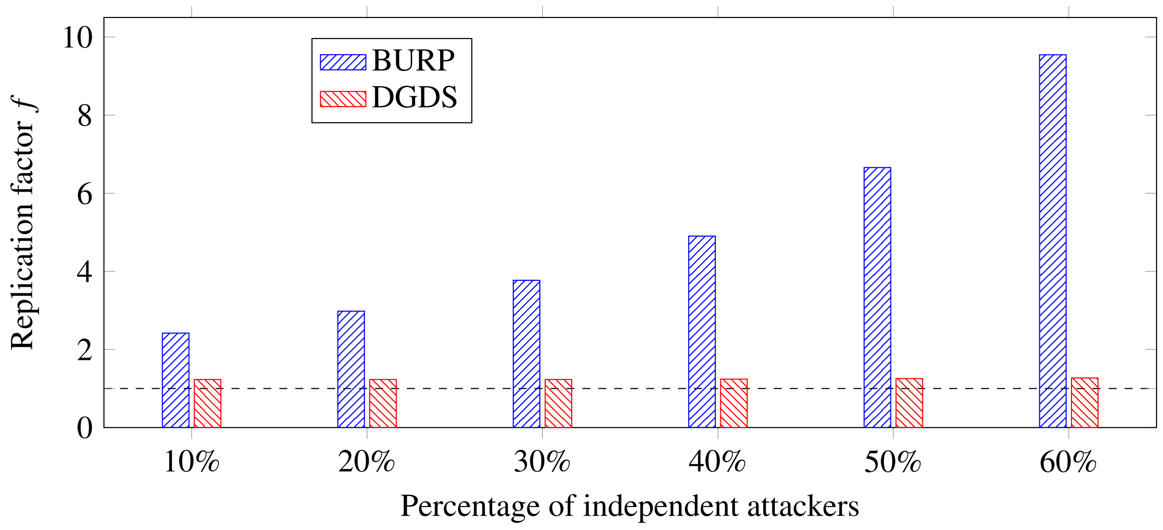


Table 4: Average number of distributed work units per second and per attacking worker for increasing percentages of attackers for BURP and DGDS. BURP distributes a constant number of work units to attackers. However, when using DGDS the number decreases as more attackers enter the system because less “trusted” workers are available to form groups that are not dominated by attackers.

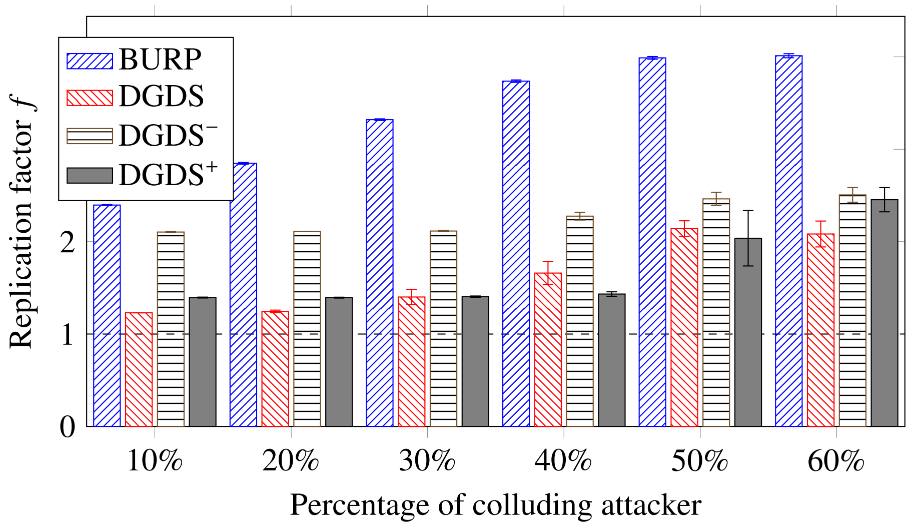
Percentage of attackers	BURP	DGDS
10%	0.796 ± 0.001	0.653 ± 0.001
20%	0.794 ± 0.003	0.302 ± 0.001
30%	0.795 ± 0.001	0.183 ± 0.001
40%	0.795 ± 0.001	0.123 ± 0.001
50%	0.793 ± 0.002	0.083 ± 0.004
60%	0.794 ± 0.001	0.058 ± 0.000

represents the effort to get a correct work unit processed.

In Figure 29, we show the replication factor for BURP and DGDS under attacks ranging from 10% to 60% of independent attackers. For BURP the replication factor f strongly increases from 2 (no attackers) to nearly 10 for 60% of attackers. Therefore, with 60% attackers, BURP retries every distribution nearly five times in average (always with two random workers). Since results of independent attackers can be distinguished, the correctness is always 100% for BURP.

DGDS stays at an average replication factor of 1.1 for all runs. This means that only every

Figure 30: Average replication factor for BURP and DGDS, DGDS$^-$ and DGDS$^+$ when being attacked by colluding incorrect workers. The x-axis shows the percentage of attackers. For all strategies, the replication factor f constantly increases. BURP starts at factor two and saturates at factor 4 with 60% colluding attackers which is less than for independent attackers because it accepts incorrect results. DGDS has a lower factor in all cases (i.e. less overhead). DGDS$^-$ is the worst variant (however still less replication than BURP). DGDS$^+$ is better with more attackers and the base DGDS with less attackers.



tenth work unit got replicated and nine out of ten were not replicated at all. However, because of the attack model, the correctness is still 100%. It is also possible to increase the minimum replication factor limit for DGDS to two, which makes it at least as safe as BURP in this case. This is further investigated in Experiment 4 (see Section 4.3.12). Similar to Experiment 1, DGDS distributes less work to attackers compared to BURP, which happens when it is not possible to form groups where "untrusted" workers do not form the majority (see Table 4).

Overall, DGDS with NM performs well in all our scenarios with independent attackers containing great improvements for large percentages of attackers.

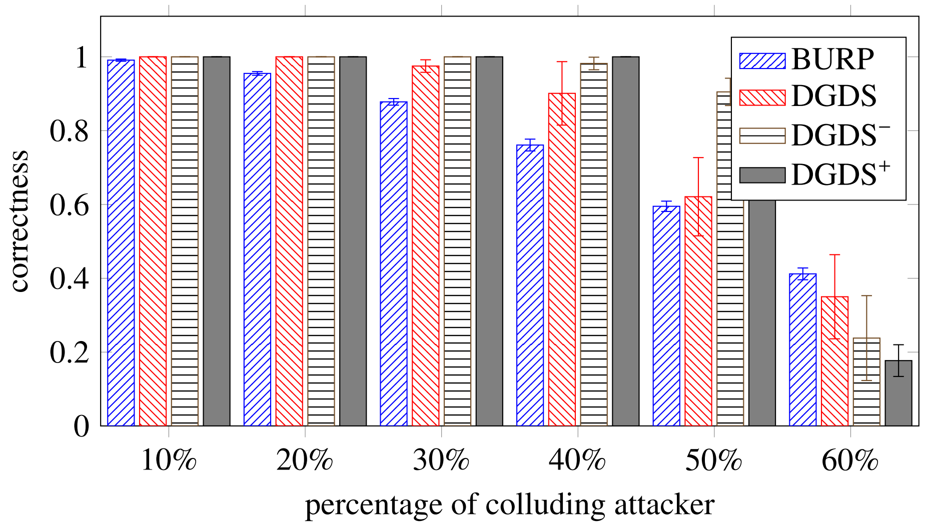
4.3.11. Experiment 3: Colluding Attackers

In this experiment, we investigate attacks by colluding attackers. Those attackers return incorrect or fake results which look the same for a certain frame[4]. If a submitter receives a quorum of incorrect results, it will accept them and award positive ratings to the attackers.

Initially, we set the limits for the NM with DGDS to $[1.1, 3]$, which showed very high throughputs in Experiments 1 and 2. However, it led to incorrect results, especially when

[4]This can be caused by broken software or malicious behaviour. The reason is not important for our approach.

Figure 31: Average correctness for BURP, DGDS, DGDS$^-$ and DGDS$^+$ when being attacked by colluding incorrect workers. The x-axis shows the percentage of attackers. For BURP, the correctness decreases monotonically. DGDS performs better (i.e. more correct results) as long as 50% or less attackers are in the system. When more than 50% of attackers are in the system, all DGDS variants collapse. DGDS$^+$ with replication factor limits [1.1, 5] performs best of all variants with nearly 100% correctness for up to 50% attackers.



dealing with colluding attackers. Therefore, we investigate two additional variants of DGDS with modified limits to give the NM more autonomy. In DGDS$^+$, we increase the upper replication limit from 3 to 5 (see Figure 26 for the calculation) which helps to identify attackers in the beginning. Similarly, DGDS$^-$ increases the lower replication limit from 1.1 to 2 and, hence, will never trust a single worker and always replicate at least once.

In Figure 30, we show the replication factor f for BURP and our three variants of DGDS for up to 60% of attackers. Additionally, in Figure 31, we show the correctness for the same experiments. BURP has the highest replication factor in all cases. However, it is notable that it replicates far less than when being attacked by independent incorrect workers (see Figure 29) because it accepts work units when it selected two colluding attackers. For BURP, the correctness decreases linearly with increasing percentages of attackers.

For DGDS, the replication factor slightly increases when more attackers join the system, but even for 60% of attackers, the factor is still better than for 10% of attackers when using BURP. For up to 50% attackers, DGDS shows better correctness than BURP. DGDS$^+$ even shows nearly 100% correctness. However, when attackers become the majority (more than 50% of attackers), our approach fails and the correctness is worse compared to BURP. This happens because the chance that the submitter accepts incorrect results in the quorum increases. At

Figure 32: Reputation over time for faithful workers and incorrect workers when being attacked by 40% colluding attackers when using DGDS with replication limits [1.1, 3]. Faithful workers use a small hit in the beginning and then slowly build up their positive reputation. Incorrect workers are slowly isolated and receive a negative reputation.

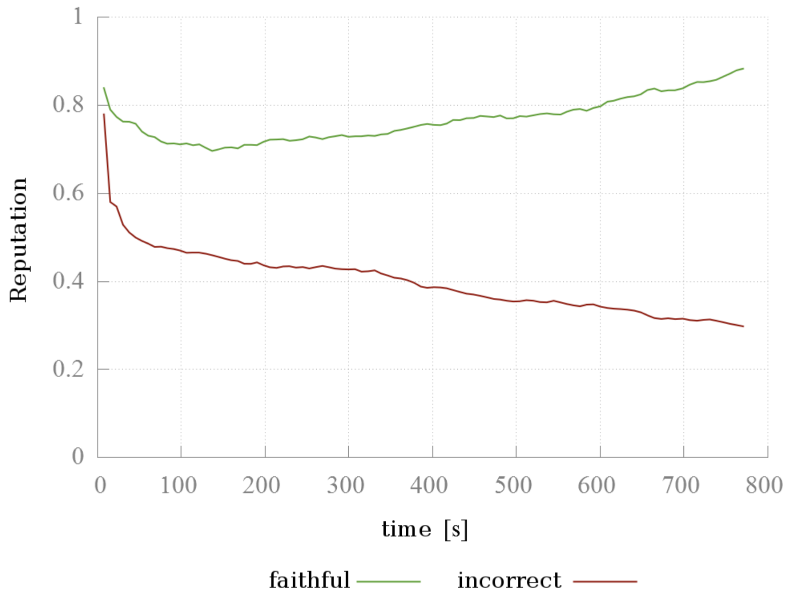


Figure 33: Reputation over time for faithful workers and incorrect workers when being attacked by 40% colluding attackers and using DGDS$^+$ with replication limits [1.1, 5]. Faithful workers quickly build up their positive reputation. Incorrect workers are isolated nearly instantly and receive a very negative reputation.

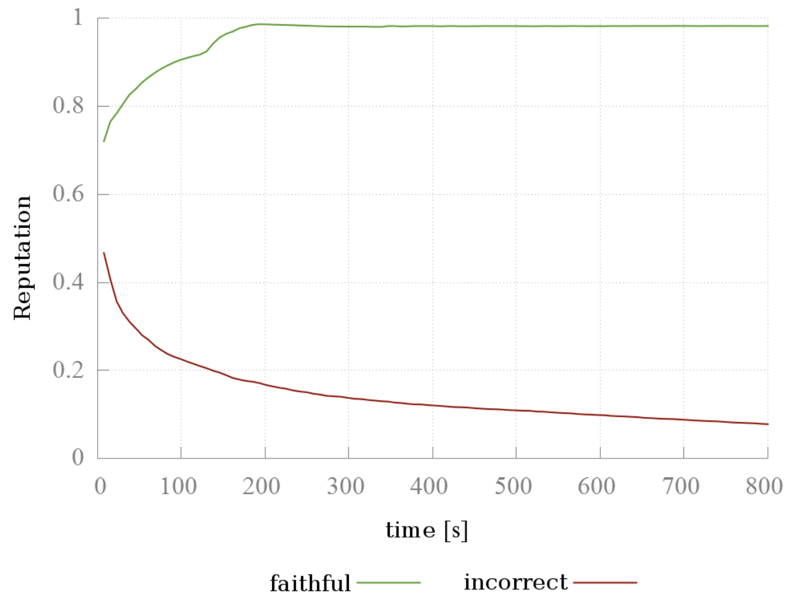


Table 5: Average reputation of attackers for DGDS, DGDS$^-$ and DGDS$^+$ with different percentages of colluding attackers. Measured when steady state is reached. Attackers gain a positive reputation when they form the majority (more than 50%). For 40% or fewer attackers, incorrect workers receive negative reputation (less than 0.4).

Strategy	30%	40%	50%	60%
DGDS	0.16 ± 0.06	0,30 ± 0.12	0.56 ± 0.10	0.68 ± 0.06
DGDS$^-$	0.05 ± 0.00	0.12 ± 0.02	0.22 ± 0.05	0.74 ± 0.07
DGDS$^+$	0.05 ± 0.00	0.08 ± 0.01	0.19 ± 0.04	0.76 ± 0.02

Table 6: Average reputation of faithful workers for DGDS, DGDS$^-$ and DGDS$^+$ with different percentages of colluding attackers. Measured when steady state is reached. Faithful workers maintain a good reputation as long as they form the majority (more than 50%). With more than 50% of attackers, faithful workers can no longer maintain positive reputation.

Strategy	30%	40%	50%	60%
DGDS	0.93 ± 0.02	0.85 ± 0.07	0.66 ± 0.06	0.49 ± 0.14
DGDS$^-$	1.00 ± 0.00	0.97 ± 0.02	0.90 ± 0.04	0.28 ± 0.16
DGDS$^+$	0.98 ± 0.00	0.98 ± 0.00	0.96 ± 0.04	0.24 ± 0.06

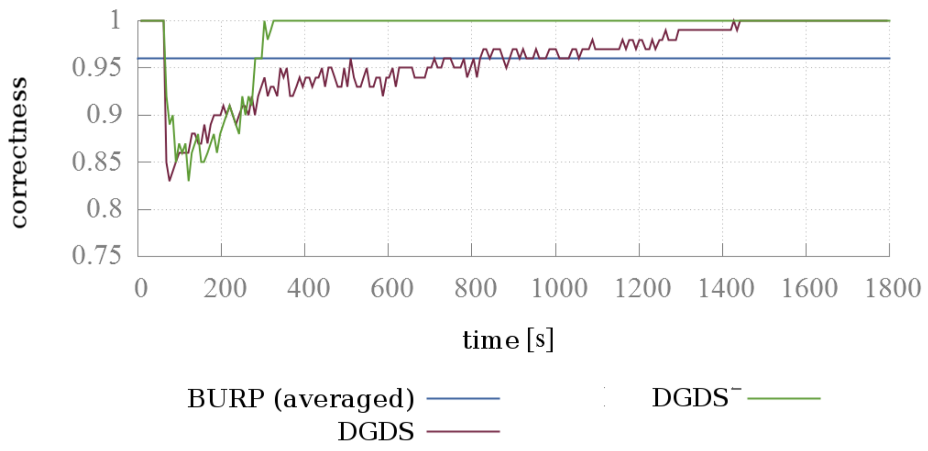
that point, attackers receive positive ratings and faithful workers receive negative ratings. A so-called *trust breakdown* [53] occurs.

In Table 5 and Table 6, we show the average reputation for incorrect workers (attackers) and faithful workers for different percentages of attackers. For up to 50% of attackers (for initial DGDS only up to 40%), the incorrect workers get isolated (reputation smaller than 0.4) and faithful workers maintain a positive reputation (larger than 0.7). However, with 60% or more attackers, the aforementioned *trust breakdown* occurs and incorrect workers actually gain a positive reputation. Similarly, faithful workers no longer maintain a positive reputation or even drop to bad reputation.

In all cases, initial DGDS performs a bit worse than DGDS$^+$ and DGDS$^-$. Therefore, we investigated the development of the reputation for DGDS and DGDS$^+$ and plotted it in Figures 32 and 33 for one exemplary run with 40% of attackers. DGDS shows a much worse settling behaviour and it takes more time for faithful workers to gain reputation. At the beginning, they even receive some negative ratings probably because of bad group selection. Similarly, it takes a long time to isolate incorrect workers. In contrast, reputation settles much faster when using DGDS$^+$. Faithful workers quickly gain good reputation and incorrect workers get isolated.

The initial choice of $[1.1, 3]$ for the replication limits did not turn out to be optimal.

Figure 34: Recovery behaviour of correctness when using BURP, DGDS and DGDS^{-} over time when being attacked by 20% incorrect workers. The attackers behave like faithful workers in the beginning to build a positive reputation and start the attack after 60s. DGDS recovers after 1400s. DGDS^{-} only needs 300s.



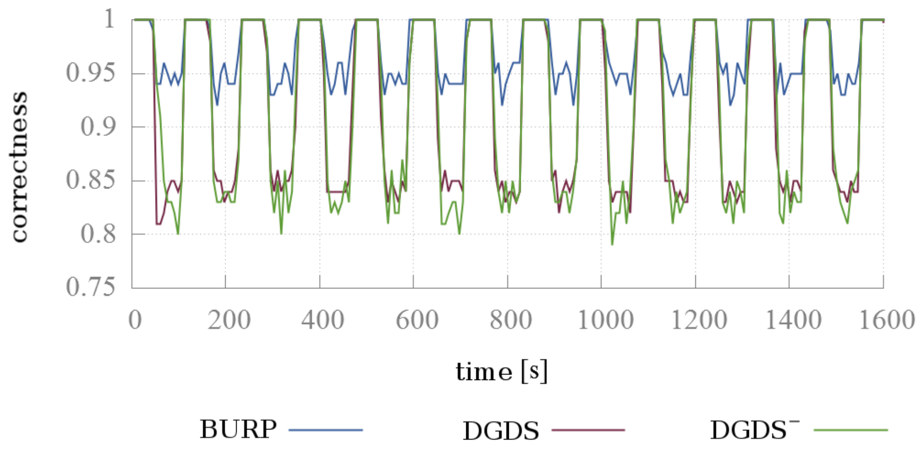
However, the NM handles the added autonomy very well and chooses an optimal balance between throughput and correctness. It turned out to be fine to increase the upper replication limit for the NM without any negative effects. The lower limit can be set to tune the system between robustness and performance as discussed in Section 3.15.2. For a performance-focused system, we would choose $[1.1, 5]$ or $[2, 5]$ for concentrating on robustness against attacks.

4.3.12. Experiment 4: Dynamic Attackers

In the previous experiments, we only considered attackers which constantly attacked the system. Therefore, in this experiment, we investigate attackers which behave like faithful workers in the beginning and change their behaviour during the experiment. This allows attackers to build a positive reputation and leverage it to maximise the damage. To simulate a maximal disturbance, we consider colluding attackers. We consider two types of attacks: i) a delayed attack which persists, and ii) a periodic attack which starts and stops in predefined periods. To capture the complete effect of both attacks, we increased the number of jobs from 30 to 100 (each with 1000 frames). We compare BURP, DGDS and DGDS^{-}. We decided to omit DGDS^{+} because it has little effect once attackers reached a high reputation.

One exemplary run of the delayed attack (which starts after 60s) is shown in Figure 34. When the attack starts, the correctness drops for DGDS and DGDS^{-} to about 85%. However,

Figure 35: Correctness of BURP, DGDS and DGDS$^-$ over time when attacked by 20% incorrect workers. The attackers behave like faithful workers in the beginning to build a positive reputation and start the attack after 60s. After another 60s, the attack stops and the attackers regain reputation for 60s. This is repeated periodically. In this worst-case scenario, DGDS and DGDS$^-$ cannot react to the attack fast enough and show a worse performance compared to BURP.



DGDS$^-$ recovers much faster than DGDS which needs about 1,400s to reach 100% again. This behaviour is consistent with Figures 32 and 33 from Experiment 3 (see Section 4.3.11). BURP maintains an average correctness of 96%. Therefore, it depends on the frequency of such attacks by colluding attackers, whether DGDS$^-$ or BURP show better correctness in the long run. However, using DGDS with the NM, it may be possible to detect and rerun all work units of periodic workers after an attack when all attackers are identified.

In Figure 35, we show a periodic attack of 20% colluding attackers which is tuned to exploit the reputation system (as a simulation of a worst-case scenario). Attackers build a positive reputation in the beginning and start an attack for 60s which is less time than the DGDS needs to recognise the attack. After the attack, they behave correctly again, rebuild their reputation and attack again. This way, they maximise the damage to the system. During the attack, DGDS only reaches 80% to 85% correctness while BURP achieves 95% correctness. Again, this may be mitigated by rerunning work units from later identified attackers (but not trivially). This is a specially crafted attack which is intentionally designed to clearly show the limits of our approach. While not shown here, this behaviour can be mitigated in the NM using the approach presented in Section 4.2 by encouraging consistent behaviour.

4.3.13. Discussion and Summary

In our evaluation, we showed that BURP in practise uses quite large replication factors even when dealing with small percentages of independent attackers (e.g., factor 4 for 40% attackers). Also, BURP has weaknesses when dealing with colluding attackers which led to incorrect results in all experiments. On the other hand, the TDG using DGDS with NM handles independent attackers with about half of the replication factor. Furthermore, it maintains 100% correctness for up to 50% of colluding attackers.

5. Discussion and Conclusion

In this section, we discuss our results regarding the success criteria formulated in Section 1.8.2. Then, we discuss open challenges and potential improvements. Afterwards, we summarise and conclude this thesis.

5.1. Maximise Utility

In all three scenarios, our approach maximises the utility when no disturbance occurs. Especially in the rendering scenario, no measurable overhead can be observed and the system performs better than a state-of-the-art system (according to the utility). This happens because the NM tunes the built-in replication mechanism and reduces security measures outside of attacks. Furthermore, even when disturbances happen, the NM selects a lower replication factor (on average) and achieves a higher correctness and less overhead (i.e. replication) compared to state-of-the-art. However, our approach shows increased performance here because BURP is not well optimised. If state-of-the-art was carefully statically tuned, we would expect that our approach exhibits a small measurable overhead when no disturbances occur. Our approach would still perform better during disturbance and, thereby, achieve a higher utility on average.

In the sensor network scenario, which represents an already well-tuned scenario, we observe a small overhead (i.e. slightly lower PDR) when no disturbance occurs. We can measure this overhead when running experiments in overload situations because our approach adds a few extra bytes to the routing messages and, therefore, reaches the overload threshold earlier. However, under normal load, the overhead of sending a few additional bytes is usually too small to be noticeable. Under attack, our approach improves the utility in almost all cases. Furthermore, after disturbances, the NM is able to optimise the secondary utility to consume less energy per node. Again, this is achieved by activating security measures only when under attack and enabling optimisation when the system is undisturbed.

Overall, a small monitoring overhead caused by the observer can be measured (which is expected because our approach sends more bytes over the RF interface) but in all scenarios it is reasonably small. In some cases where disturbances hardly occur (i.e. in closed systems which are not the primary focus of this thesis), this approach might not be beneficial. However, in all open systems and even in most very large closed systems, this overhead should be neglectable because disturbances happen frequently (mostly non-intentionally caused by broken components) and the advantages of a more robust system prevail. Furthermore, in some cases the benefit from runtime optimisations can outweigh the monitoring overhead (i.e. in the rendering scenario).

5.2. Robustness Under Attack

Using the norms in ODS allows the NM to adjust expected behaviour during runtime. Since agents are autonomous by definition (see Section 1.3), they can choose not to adhere to the valid norms. However, other agents will assign them bad ratings as a consequence and they become isolated if their reputation decreases too much. This mechanism is leveraged by the NM to punish certain behaviours or to encourage a different behaviour depending on the type of attackers. For example in the TDG, the NM punishes *Egoists* by increasing sanctions for cancelling WUs which renders this behaviour unprofitable for the attacker. This allows an ODS to deal with attackers as long as they do not become a majority (see Section 5.5 for a discussion of this case).

In the TDG example, this works well and the system recovers even under very large attacks for various attacker types and scenarios. Additionally, in some cases (see Section 4.2.8), the TDG is able to become immune against attackers (i.e. achieving a perfect passive robustness). Similarly, the rendering scenario becomes very robust against attackers using our approach (again as long as they stay a minority).

While the state-of-the-art sensor network scenario is well-tuned for undisturbed operation, it shows flaws under disturbance. This is mostly because additional safety measures cause overhead which would reduce the utility in general. Therefore, we achieve huge improvements with our approach because it only activates those measures when attacks occur. As an effect, nodes route their traffic around disturbances which might be non-intentional and temporary. However, when the disturbance ends, they stick with those longer routes. This causes more radio transmissions which should be minimised as secondary utility. Therefore, the NM activates another norm to have nodes retry shorter paths when it detects that the disturbance no longer exists. This optimisation allows nodes to perform exploration for a better solution to maximise the long-term utility.

In all scenarios, additional measures and optimisations exist but only selected measures were available to the NM (based on best practises). This is a limitation of this approach because the NM can only change norms which are known at design time. The NM will learn to find the best possible norm for a certain situation but it cannot invent new measures if none of the existing parameters or actions affect the type of attacker. For example, if an attacker physically destroys nodes in a sensor network, there is currently no way to recover from that for the NM.

5.3. Permanent Threats and Attacks to the Reputation System

Another challenge in ODS are permanent threats, for example represented by agents which cannot be easily isolated because they behave strategically. For example, in the TDG scenario, *Cunning Agents* are such a threat (see Section 4.2). They behave well for a certain time and start an attack once they have gained high reputation. The NM isolates them by issuing a norm to enforce consistent behaviour which proved to work well (see Figure 23). Again, this norm is not always beneficial since agents sometimes need to change their behaviour to adapt to new situations. Therefore, the NM will only selectively activate this norm until such attackers are isolated.

While *Cunning Agents* behave strategically regarding their reputation, more malicious behaviour is possible. For example, agents can collude to exploit the system as *Freeriders* and *Egoists* do in Experiment 2 (see Section 4.2.8). Furthermore, groups of agents can also issue fake ratings among each other which gives them an incorrect high reputation. However, the NM can detect such groups (as we have shown in [C11]) and mitigate their attacks by issuing norms.

5.4. Recovery from NEB

NEB can occur as result of attacks or during disturbances such as overload. Ideally, the NM can counter attacks early to prevent NEB such as in Section 4.2.8. Unfortunately, this is not possible in most cases. Therefore, the system needs to be able to recover from NEB when detected. Usually, this requires the NM to somehow identify and possibly mitigate the attack. Additionally, it needs to encourage cooperation because NEB often leads to a *trust breakdown* or overload which essentially stop agents from cooperating. Since the overload is caused by the attack, one strategy in the NM is to loosen obligations in norms. This allows agents to selectively accept WUs from trusted partners but spares them from sanctions when not working for other agents. Furthermore, when a full *trust breakdown* occurred, reputation limits are usually adapted as well.

This strategy proved successful in the TDG which recovers from attacks of large groups of *Freeriders* that caused a *trust breakdown*. Furthermore, this also works for our distributed rendering scenario on top of the TDG which constantly operates in saturation. The sensor networks scenario also recovers from groups of attackers but there is only a limited amount of NEB occurring.

5.5. Attacker Majority

One challenge arises in the rendering and TDG scenario when attackers become the majority. At this point, only a minority of agents adhere to the norms of the NM which renders the NM and the reputation system useless. Instead of being purely malicious, the majority could as well install another reputation system with another NM and, thereby, deprecate the old NM. More generally, our approach requires a majority of agents to adhere to our NM, or it will eventually fail. In some cases, the system might be still operational when more than 50% of attackers join (if not all of them collude), but the system will no longer be robust.

A majority of (colluding) attackers also poses problems to most of our measures such as replication. As shown in the rendering scenario (see Section 4.3), attackers will win a majority quorum if they are in majority. The same applies to statistics which are used to assess good behaviour in the systems. Eventually, the attackers will gain good reputation and well-behaving agents will receive a bad reputation. Therefore, to use our approach, the percentage of attackers has to stay below 50%.

5.6. Scaling Clustering

When observing the system, the observer builds the *trust graph* and performs clustering. Unfortunately, most clustering algorithms have an exponential complexity which does not scale for larger systems. For our system sizes in the evaluation with 50,000 or fewer agents, the runtime is still reasonably short. However, there exists an upper limit where the runtime becomes so large that we cannot add processing power nor increase the monitoring interval. Fortunately, the underlying *trust graph* changes only slowly. Therefore, it should be certainly possible to start with the previous graph and only perform a few clustering iterations every time. This would reduce the average complexity and the clustering should improve over time which should allow to scale our approach to very large systems.

5.7. Feedback Duration

To learn which norm performs well in a given situation, the controller has to get feedback after selecting a given action. As known from control theory [150], the feedback is usually delayed for a certain time. However, using norms, delay time strongly depends on the parameter or action, and that value is unknown to the NM. Therefore, we currently simply wait for a system response with a very long timeout because in some cases there may be no reaction. To improve this, we keep historic data to calculate the average response and adjust the timeout. Nevertheless, this can probably still be improved to search the norm space more efficiently.

5.8. Conclusion

In this thesis, we presented an approach to guide self-organisation in ODS. We started by defining ODS and describing our system model. Afterwards, we discussed challenges which arise from the open nature of such systems and listed our assumptions about such systems. In particular, agents in ODS have to be considered as black boxes and there is no assumption of benevolence, leading to uncertainty about their behaviour and intention. To enforce a desirable behaviour in a distributed manner, we introduced the concept of trust and reputation. Furthermore, we defined technical norms to achieve desirable behaviour with corresponding incentives and sanctions. Using these norms, agents can reason about actions during runtime and derive consequences. This mechanism enables systems to self-organise and isolate single attackers. However, if larger groups of colluding attackers enter the system, negative emergent behaviour can occur, resulting in a *trust breakdown*. As a consequence, agents do not trust any other agents and, therefore, are no longer willing to cooperate. This can also occur during disturbances (i.e. broken components) or when the system experiences overload.

To further illustrate challenges in ODS, we introduced three application scenarios. First, we described the distributed low-power sensor network scenario with geographically distributed nodes which only communicate locally with their neighbours. To send data to the *root* node, nodes have to forward packets for their peers. However, attackers can drop packets silently and the sender will not notice. The second scenario are OGS where agents can join the system voluntarily. Agents promise to work for other agents in the OGS and, therefore, may also submit jobs into the system to gain a high speedup. However, we cannot control the agent behaviour and agents may try to exploit the system. In the third scenario, as exemplary application on top of the OGS, we introduce a distributed rendering scenario. In contrast to the plain OGS, this application is constantly in overload, because there much more jobs than workers exist. Therefore, agents try to optimise their throughput. At the same time, they try to achieve a high correctness and, therefore, employ replication to ensure correct results.

In a next step, we introduced agent organisations in which agents organise in a decentralised way. We defined an iTC, representing the group of agents with a high reputation in the system.

This is a direct result of the trust and reputation system and leads to isolation of attackers. However, when NEB happens, those iTCs quickly dissolve and isolation of attackers fails. Therefore, we introduced eTC which implement an explicit membership functions. Inside an eTC, agents have to comply to stricter rules and will get excluded from the community when they break those rules.

Afterwards, we formulated the objectives and success criteria for this thesis. We defined criteria for developing a low-overhead mechanism to improve the performance of ODS. It should increase the robustness under attack and allow recovery from permanent threats. Furthermore, it should impede attacks to the reputation system and NEB or, at least, allow recovery after such disturbances occurred. Thereafter, we discussed related work and compared it to our criteria.

Next, we presented our approach to implement a mechanism fulfilling the previously defined criteria. Agents vote a higher-level NM which observes the system and reacts depending on the situation. The NM consists of an observer and a controller using the well-known observer-controller pattern. Depending on the situation, the NM influences the system by changing norms. If the NM detects disturbances, it will activate safety measures and deactivate them afterwards.

To monitor the system, the NM builds a *trust graph* based on the reputation system which can be used to determine the system state. To achieve this, we defined requirements for a suitable trust metric and developed such a metric. Furthermore, we formalised norms which are used in ODS and described how they interact with the reputation system using incentives and sanctions. In a next step, we discussed how the observer can quantify recovery after an attack. This is leveraged by the NM to learn how the selected measure performed to counter the observed type of disturbance. Additionally, we presented an approach to measure the amount of self-organisation occurring in the system. This information is used by the NM to assess whether the system is currently stable or if disturbances are taking place. Furthermore, we introduced graph metrics to identify agent behaviour within the *trust graph*. Based on this metrics, we defined requirements for suitable clustering algorithms to find groups in the *trust graph*. We were able to identify groups of similar-behaving agents and groups of colluding agents. On the one hand, groups of similar agents are used to classify the different types of agents in the system (i.e. this allows optimisation if certain attackers are not present). On the other hand, groups of colluding agents are used to detect colluding attackers and attacks to the reputation system. Using this information, the controller changes norms when deemed beneficial. However, it does not know the effect of the parameters in norms initially. It starts

Figure 36: Comparison of related work according to our criteria. Our approach is included in the last row and fulfils all criteria. ✓= criteria fulfilled, (✓)= partially fulfilled, ✗= not fulfilled or does not apply (i.e. no reputation system exists)

	Utility Maximisation	Fast Recovery from Attacks	Detection and Mitigation of Permanent Threats	Detection and Recovery from Attacks to the Reputation System	Recovery from NEB
RPL [56]	✓	✗	✗	✗	✗
Ganeriwal et al. [116]	✓	(✓)	(✓)	✗	✗
Leligou et al. [117]	✗	(✓)	(✓)	✗	✗
TARF [118]	✓	✓	(✓)	✗	✗
TORQUE [119]	✓	✗	✗	✗	✗
TDG [51]	✓	✓	(✓)	(✓)	(✓)
HTCondor [120]	✓	✗	✗	✗	✗
JoSchKa [121]	✓	✗	✗	✗	✗
BOINC [66, 122]	✗	(✓)	(✓)	✗	✗
Our approach	✓	✓	✓	✓	✓

by randomly trying changes and measuring the robustness afterwards. When no recovery occurred, it will try another action until recovery succeeded. After a few iterations, this mechanism learns to select the correct action for certain situations.

To prove our approach, we performed experiments for each of the three application scenarios. In the sensor network scenario, we presented an experiment with end-to-end trust which increases the PDR when disturbances occur. Using this approach, nodes in a WSN detect attackers and find an alternative route. However, after the disturbance ends, those routes still prevail. Unfortunately, those alternative routes are typically longer and, thereby, require more radio transmissions consuming more energy. Therefore, the NM triggers a second-chance mechanism to retry previously broken (or malicious) nodes to recover from this (otherwise permanent) NEB. With our approach, we achieved a higher PDR when attacks occurred and the system recovered from NEB caused by those disturbances.

In the OGS scenario, we demonstrated that the NM can detect and isolate *Cunning Agents* which behave inconsistently. Those are identified as a colluding group which is then classified using our metrics. Afterwards, the NM issues a norm to encourage consistent behaviour in the system. This allows full isolation of the attackers and the system recovers to full utility. Furthermore, we showed an experiment where different agent types joined the system. On the one hand, for *Adaptive Agents*, a fast integration is desired. On the other hand, for attackers (such as *Freeriders* or *Egoists*), we aim for fast isolation. The NM can tune isolation and integration time using the initial reputation parameter δ. This mechanism lead to quicker isolation of attackers but also to slower integration of well-behaving agents. Thereby, the NM can fence the system from new agents to become more robust when disturbances occur.

In the third scenario, we compared our approach to BURP, which is a state-of-the-art rendering application. We first described the distribution strategies used in our approach and then evaluated them against BURP. Using the NM, the overhead is reduced (i.e. less replication) when no disturbance occurs. Furthermore, when attacks take place, the NM ensures a higher correctness and less overhead.

Additionally, we investigated the limits of our approach. First, when attackers become the majority, they win the quorum and our approach fails. Furthermore, when tuning the system for performance (i.e. low replication factor), strategic attacks become feasible and attackers can exploit the system. However, the NM can detect those cases and adjust the limits accordingly. Overall, we have shown that our approach significantly outperforms state-of-the-art approaches and fulfils the success criteria (see Figure 36).

Appendices

A. Trust Metrics

To use trust in OC systems, the ratings represented should have a well-defined semantic. In general, a rating r is either good or bad. Neutral ratings have no impact and, therefore, can be omitted. In the following, we consider a rating with a value above/below 0 as good/bad rating and define r_+/r_- as a set of all good/bad ratings (69, 70). In all cases, $\mathfrak{T}(R) = 0$ is considered as a neutral reputation.

An agent receives multiple ratings over time which are stored in a FIFO queue called R_n, containing up to k elements (R_0 will be empty in the beginning). We also define a trust metric $\mathfrak{T}(R)$ to calculate the *reputation* τ based on R_n (68). In the FIFO queue R_n, recent ratings are in the back and move to the front when more ratings are added. We define a function $\mathfrak{A}(R_n)$ to add an entry r to R_n (71).

$$r \in \mathfrak{R}, R_n \in \mathfrak{R}^k, n, k \in \mathbb{N} \tag{68}$$

$$\mathfrak{R}_+ := \{r : r > 0 \land r \in \mathfrak{R}\} \tag{69}$$

$$\mathfrak{R}_- := \{r : r < 0 \land r \in \mathfrak{R}\} \tag{70}$$

$$R_{n+1} := \mathfrak{A}(R_n, r) \tag{71}$$

$$\mathfrak{T} : \mathfrak{R}^k \to [-1, 1] \tag{72}$$

$$\tau := \mathfrak{T}(R_n) \tag{73}$$

$$\tag{74}$$

To consider R over time, we add an iterator n which gets increased when new ratings are added. For multiple reasons, a trust metric has to implement the concept of *ageing values* or forgiveness in a social sense [127]. In most cases, recent behaviour is more relevant when calculating a reputation value. Otherwise, agents will not be able to integrate into the system after they changed their behaviour. Additionally, there is usually some kind of memory limit k such that not all previous ratings can be stored forever (typically $500 \leq k \leq 1000$ ratings). If R_n already contains k elements, the first element of the list will be removed (FIFO behaviour;

Equation (75)).

$$R_{n+1} := \mathfrak{A}(R_n, r) = \begin{cases} (z_1, \dots, z_k, r) & |R_n| \geq k, z \in R_n \\ (z_2, \dots, z_i, r) & |R_n| = i < k, z \in R_n \end{cases} \tag{75}$$

We will use this notation through the paper and add exponents for variants: R^b for a binary metric; R^c for a continuous metric; R^w for a weighted metric; R^s for a weighted SES metric. The exponent k is an exception and is used as Cartesian product.

A.1. Binary Metric

Depending on the semantics of a rating, different representations are possible. In the simplest case, ratings are binary: Either good or bad. They can be implemented as a boolean type. If the rating is bad, we clear the bit and set it otherwise. To aggregate binary ratings, we assign a value of -1 to a bad rating, 1 to a good rating, and use a normalised sum function calculate the reputation value τ_b (76, 77). R^b behaves similarly to a FIFO queue as above (78).

$$\mathfrak{R}^b := \{-1, 1\}, r^b \in \mathfrak{R}^b, R^b_n \in \mathfrak{R}^{b^k} \tag{76}$$

$$\tau^b := \mathfrak{T}^b(R^b_n) := \frac{\sum R^b_n}{|R^b_n|} \tag{77}$$

$$R^b_{n+1} := \mathfrak{A}^b(R^b_n, r^b) \tag{78}$$

A.2. Continuous Metric

The second case covers continuous ratings which are used, e.g., in TDG. In such systems, ratings have a real value, where positive values represent good ratings and negative numbers represent bad ones. This is useful when certain actions are more desirable than others. Therefore, ratings are represented by numeric values between -1 for a very negative and 1 for a very positive rating (79). To aggregate those ratings, systems like the TDG use a normalised sum (80). R^c behaves similar to R^b (81).

$$\mathfrak{R}^c := [-1, 1], r^c \in \mathfrak{R}^c, R^c_n \in \mathfrak{R}^{c^k} \tag{79}$$

$$\tau^c := \mathfrak{T}^c(R^c_n) := \frac{\sum R^c_n}{|R^c_n|} \tag{80}$$

$$R^c_{n+1} := \mathfrak{A}^c(R^c_n, r^c) \tag{81}$$

A.3. Problems with Previous Metrics

If one agent performs an action A which is preferred, it receives a more positive rating. E.g., A may have a value of 0.9 while a less desirable action B may get a value of 0.45 assigned. If an agent performed some unwanted action C with a value of -0.9, it would either need to perform A once or B twice to gain a neutral reputation of 0.

Unfortunately, this only works for combinations of negative and positive ratings. Intuitively, we would assume that an agent may gain the maximal reputation by either performing a few very desirable actions or, numerous positive, but less desirable actions. However, this is not the case for $\mathfrak{T}^c(R_n^c)$:

$$R_1^c := (1, 1, 1) \tag{82}$$

$$\mathfrak{T}^c(R_1^c) := \frac{1 + 1 + 1}{3} = 1 \tag{83}$$

$$R_2^c := (1, 1, 1, 0.5, 0.5, 0.5) \tag{84}$$

$$\mathfrak{T}^c(R_2^c) := \frac{3 \cdot 1 + 3 \cdot 0.5}{6} = 0.75 \tag{85}$$

In R_1^c (82), we consider an agent which received three perfect ratings. In R_2^c (84) the same agent received three additional medium positive ratings which were added using Equation (81) (considering $k \geq 6$). As expected, R_1^c results in a perfect reputation $\mathfrak{T}^c$ of 1 (83). We would assume that this is also valid for R_2^c. Additional positive ratings should not decrease the reputation. Unfortunately, this is happening with $\mathfrak{T}^c$ and the reputation decreases to 0.75 (85).

Consider a rational agent which gained a perfect reputation of 1: Since it behaves rationally, it will not select any action which will decrease its reputation. Unfortunately, with our previous metric $\mathfrak{T}^c$, it can no longer select any action with a rating other than 1. However, all ratings with a value higher than 0 are considered as good and, thus, desirable. Therefore, we require a metric which will not decrease the reputation an agent receives a good rating.

A.4. Requirements for Trust Metric

In this section, we formalise our requirements that have to be fulfilled by the trust metric $\mathfrak{T}$ to calculate a reputation τ. Afterwards, we test previous work against the requirements and introduce new improved metrics. To meet the challenges highlighted in the previous section, a representation of trust should be able to handle ratings of different intensities. I.e., if an agent has to choose from two possible desirable actions where one is favoured, we want to represent this fact in the resulting trust ratings r_1 and r_2. If r_1 is more favourable than r_2 we expect the resulting reputation to increase more after receiving r_1 than after r_2 unless the reputation is already maximised (R1). The same should symmetrically apply to negative ratings. Additionally, we expect the reputation to increase for every positive rating until it reaches its maximum (R2). Again, the exact opposite should apply for negative ratings.

Our main motivation for R2 is a scenario where an agent can mandate to perform a desirable action which has a low priority (e.g., it is not required to do something). Based on previous metrics, if the agent already has a high reputation (i.e., received many good ratings), its reputation would suffer. Therefore, the agent will choose not to take the job. This could also be circumvented using other external incentives (such as virtual currencies), but in this paper we try to solve it without adding more unnecessary complexity.

$$\begin{aligned}\forall r_1, r_2 \in \mathfrak{R}_+ : \mathfrak{T}(\mathfrak{A}(R_n, r_1)) > \mathfrak{T}(\mathfrak{A}(R_n, r_2)) \vee \\ \mathfrak{T}(\mathfrak{A}(R_n, r_2)) = 1, \\ r_1 > r_2 \Rightarrow |\mathfrak{R}_+| > 1\end{aligned} \tag{R1}$$

$$\forall r \in r_+ : \mathfrak{T}(\mathfrak{A}(R_n, r)) > \mathfrak{T}(R_n) \vee \mathfrak{T}(R_n) = 1 \tag{R2}$$

A.5. Previous Metrics

Unfortunately, both metrics mentioned in previous work are not fulfilling those requirements. In case of $\mathfrak{T}^b(R_n^b)$, ratings are binary and $\mathfrak{R}_+^b$ only contains one element (86). Therefore, Equation (R1) cannot be fulfilled since we are unable to select two distinct elements r_1 and r_2 and $\mathfrak{T}^b$ is not always 1.

$$\mathfrak{R}_+^b := \{1\} \Rightarrow |\mathfrak{R}_+^b| = 1 \not> 1 \lightning \tag{86}$$

For $\mathfrak{T}^c(R_n^c)$, we have more than two different positive ratings in $\mathfrak{R}_+^c$ and Equation (R1) holds. Unfortunately, we can easily find an R^c and select an r out of $\mathfrak{R}_+^c$, such that Equation (R2) breaks (90, 91):

$$\mathfrak{R}_+^c := (0, 1] \Rightarrow |\mathfrak{R}_+^c| > 1 \tag{87}$$

$$R_1^c := (0.5, 0.5) \tag{88}$$

$$r := 0.1 \in \mathfrak{R}_+^c \tag{89}$$

$$\mathfrak{T}^c(\mathfrak{A}^c(R_1^c, 0.1)) \overset{?}{>} \mathfrak{T}^c(R_1^c) \vee \mathfrak{T}^c(R_1^c) \overset{?}{=} 1 \tag{90}$$

$$\frac{0.5 + 0.5 + 0.1}{3} \not> \frac{0.5 + 0.5}{2} \vee \frac{0.5 + 0.5}{2} \neq 1 \lightning \tag{91}$$

A.6. Weighted Trust Metric

Our first proposal is a weighted trust metric: Ratings are still represented by a float in the range between -1 and 1 and stored similarly (92). However, we change the trust function $\mathfrak{T}^w(R_n^w)$ to aggregate positive and negative weighted ratings. We normalise trust value by the sum of all ratings to account for the strength of ratings (93).

$$\mathfrak{R}^w := [-1, 1], r^w \in \mathfrak{R}^w, R_n^w \in \mathfrak{R}^{w^k} \tag{92}$$

$$\begin{aligned} \tau^w := \mathfrak{T}^w(R_n^w) &:= \frac{\sum_{r \in R_n^w, r>0} r - \sum_{r \in R_n^w, r<0} -r}{\sum_{r \in R_n^w, r>0} r + \sum_{r \in R_n^w, r<0} -r} \\ &= \frac{\sum_{r \in R_n^w} r}{\sum_{r \in R_n^w} |r|} \end{aligned} \tag{93}$$

A weighted trust metric generally fulfils our requirements from Equation (R1) when $|R_n^w| < k$. We substitute $\mathfrak{A}^w$ in Equation (96) and $\mathfrak{T}^w$ in Equation (97). If R_n^w also contains negative ratings, the first case of the equation holds. If it only contains positive ones, the second case holds (97).

$$R_1^w := (p_1, \cdots, p_{k-2}), p_i \in \mathfrak{R}^w \tag{94}$$

$$\begin{gathered} \forall r_1, r_2 \in \mathfrak{R}_+^w : \mathfrak{T}^w(\mathfrak{A}^w(R_1^w, r_1)) \overset{?}{>} \mathfrak{T}^w(\mathfrak{A}^w(R_1^w, r_2)) \vee \\ \mathfrak{T}^w(\mathfrak{A}^w(R_1^w, r_2)) \overset{?}{=} 1, \\ r_1 > r_2 \Rightarrow |\mathfrak{R}_+^w| > 1 \end{gathered} \tag{95}$$

$$\begin{aligned} \mathfrak{T}^w((p_1, \cdots, p_{k-2}, r_1)) \overset{?}{>} \mathfrak{T}^w((p_1, \cdots, p_{k-2}, r_2)) \\ \vee \mathfrak{T}((p_1, \cdots, p_{k-2}, r_2)) \overset{?}{=} 1 \end{aligned} \tag{96}$$

$$\begin{aligned}
\sum_{p \in R_1^w} |p| > \sum_{p \in R_1^w} p &\Rightarrow \frac{r_1 + \sum_{p \in R_1^w} p}{r_1 + \sum_{p \in R_1^w} |p|} > \frac{r_2 + \sum_{p \in R_1^w} p}{r_2 + \sum_{p \in R_1^w} |p|}, \\
\sum_{p \in R_1^w} |p| = \sum_{p \in R_1^w} p &\Rightarrow \frac{r_2 + \sum_{p \in R_1^w} p}{r_2 + \sum_{p \in R_1^w} |p|} = 1 \ \blacksquare
\end{aligned} \tag{97}$$

Similarly, it can be shown that the requirements from Equation (R2) are always fulfilled:

$$\forall r \in \mathfrak{R}_+^w : \mathfrak{T}^w(\mathfrak{A}^w(R_n^w, r)) \overset{?}{>} \mathfrak{T}^w(R_n^w) \vee \mathfrak{T}^w(R_n^w) \overset{?}{=} 1 \tag{98}$$

$$\begin{aligned}
\sum_{p \in R_1^w} |p| > \sum_{p \in R_1^w} p &\Rightarrow \frac{r + \sum_{p \in R_1^w} p}{r + \sum_{p \in R_1^w} |p|} > \frac{\sum_{p \in R_1^w} p}{\sum_{p \in R_1^w} |p|}, \\
\sum_{p \in R_1^w} |p| = \sum_{p \in R_1^w} p &\Rightarrow \frac{\sum_{p \in R_1^w} p}{\sum_{p \in R_1^w} |p|} = 1 \ \blacksquare
\end{aligned} \tag{99}$$

Unfortunately, this does not always hold, when $|R_n^w| \geq k$. When adding a new rating to the FIFO queue R_n^w (using $\mathfrak{A}^w(R_n^w, r)$) and the first rating p_1 in R_n^w being smaller than r, the overall reputation will drop even if $r > 0$. Thereby, Equation (R2) does not hold universally when R_n^w has limited size. However, it would hold if we dropped the FIFO properties of R_n^w (which would be require more memory).

$$R_2^w := (p_1, \cdots, p_{k-1}),\ p_i \in \mathfrak{R}^w \tag{100}$$

$$\forall r \in \mathfrak{R}_+^w : \mathfrak{T}^w(\mathfrak{A}^w(R_2^w, r)) \overset{?}{>} \mathfrak{T}^w(R_2^w) \vee \mathfrak{T}^w(R_2^w) \overset{?}{=} 1 \tag{101}$$

$$\begin{aligned}
&\sum_{p \in R_2^w} |p| > \sum_{p \in R_2^w} p, r \in \mathfrak{R}_+^w, p_1 > r \Rightarrow \\
&\qquad \frac{r - p_1 + \sum_{p \in R_2^w} p}{r - |p_1| + \sum_{p \in R_2^w} |p|} \not> \frac{\sum_{p \in R_2^w} p}{\sum_{p \in R_2^w} |p|} \\
&\qquad \vee \frac{\sum_{p \in R_2^w} p}{\sum_{p \in R_2^w} |p|} \neq 1, \ \lightning
\end{aligned} \tag{102}$$

In reality, this is a border case with little impact and the metric is nevertheless very well suited, but strictly speaking, it does not fulfil the requirements.

A.7. Weighted Simple Exponential Smoothing Trust Metric

To further improve the trust metric, we leverage *Simple Exponential Smoothing* (SES) [126]. This method can be seen as an advanced version of a rolling average, but it does not have to remember historic values. Therefore, R^s is no longer considered as a FIFO queue (104). As a side effect, newer ratings have more impact to the resulting value, which is desirable for many applications (i.e., this implements some kind of forgiveness [127]).

Those properties allow us to create an advanced metric based on R^w_n, which (I) fulfils our requirements and (II) requires less memory. Obviously, SES can be used to reduce the memory usage of R^c_n down to just one float value. However, to use it on R^w_n, we need to store the weight of positive and negative ratings separately. Based on the principle of SES, we also define $\mathfrak{A}^s(R^s_n, r)$ to modify the weight of either positive or negative ratings. However, both sums are multiplied by α to keep the values proportional (105). The value of α controls how much influence a new rating has. A typical value for α would be between 0.95 and 0.999, depending on how many old values should influence the trust value (similar to k in the previous metrics). Calculating the trust values works by subtracting the negative weight from the positive weight and normalize is by the total weight (see Equation (106); similar to $\mathfrak{T}^w(R^w_n)$ with the difference that the sums for positive and negative ratings are already precomputed).

$$\mathfrak{R}^s := [-1, 1],\, r^s \in \mathfrak{R}^w \tag{103}$$

$$R^s := [0, 1]^2 \tag{104}$$

$$\begin{aligned} R^s_{n+1} &:= \mathfrak{A}^s(R^s_n, r) \\ &:= \begin{cases} (p_1 \cdot \alpha + (1-\alpha) \cdot r, p_2 \cdot \alpha) & r > 0, (p_1, p_2) \in R^s_n \\ (p_1 \cdot \alpha, p_2 \cdot \alpha - (1-\alpha) \cdot r) & r < 0, (p_1, p_2) \in R^s_n \\ R^s_n & \text{otherwise} \end{cases} \end{aligned} \tag{105}$$

$$\tau^s := \mathfrak{T}^s(R^s_n) := \frac{p_1 - p_2}{p_1 + p_2}, (p_1, p_2) \in R^s_n \tag{106}$$

For this metric, we can show that our requirements from Equation (R1) are always fulfilled:

$$R_1^s := (p_1, p_2) \tag{107}$$

$$\begin{aligned}\forall r_1, r_2 \in \Re_+^s : \mathfrak{T}^s(\mathfrak{A}^s(R_n^s, r_1)) \overset{?}{>} \mathfrak{T}(\mathfrak{A}^s(R_n^s, r_2)) \vee \\ \mathfrak{T}^s(\mathfrak{A}^s(R_n^s, r_2)) \overset{?}{=} 1, \\ r_1 > r_2 \Rightarrow |\Re_+^s| > 1\end{aligned} \tag{108}$$

$$\begin{aligned}\mathfrak{T}^s(\begin{pmatrix} p_1 \cdot \alpha + (1-\alpha) \cdot r_1 \\ p_2 \cdot \alpha \end{pmatrix}) \overset{?}{>} \mathfrak{T}^s(\begin{pmatrix} p_1 \cdot \alpha + (1-\alpha) \cdot r_2 \\ p_2 \cdot \alpha \end{pmatrix}) \\ \vee \mathfrak{T}^s(\begin{pmatrix} p_1 \cdot \alpha + (1-\alpha) \cdot r_2 \\ p_2 \cdot \alpha \end{pmatrix}) \overset{?}{=} 1\end{aligned} \tag{109}$$

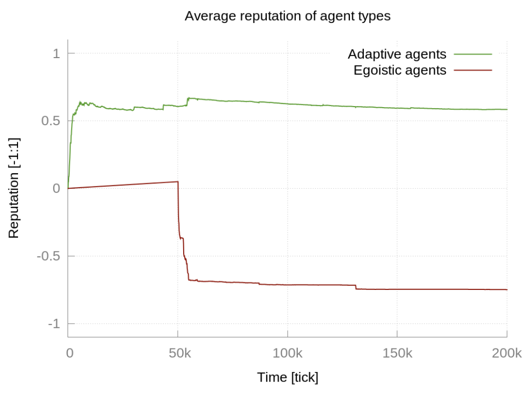
$$\begin{aligned}0 < \alpha < 1, r_1 > r_2, 0 \leq p_1 \leq 1, 0 \leq p_2 \leq 1 \Rightarrow \\ \frac{p_1 \cdot \alpha + (1-\alpha) \cdot r_1 - p_2 \cdot \alpha}{p_1 \cdot \alpha + (1-\alpha) \cdot r_1 + p_2 \cdot \alpha} > \frac{p_1 \cdot \alpha + (1-\alpha) \cdot r_2 - p_2 \cdot \alpha}{p_1 \cdot \alpha + (1-\alpha) \cdot r_2 + p_2 \cdot \alpha}, \\ \vee \frac{p_1 \cdot \alpha + (1-\alpha) \cdot r_2 - p_2 \cdot \alpha}{p_1 \cdot \alpha + (1-\alpha) \cdot r_2 + p_2 \cdot \alpha} = 1 \quad \blacksquare\end{aligned} \tag{110}$$

Also, the requirements from Equation (R2) are always fulfilled:

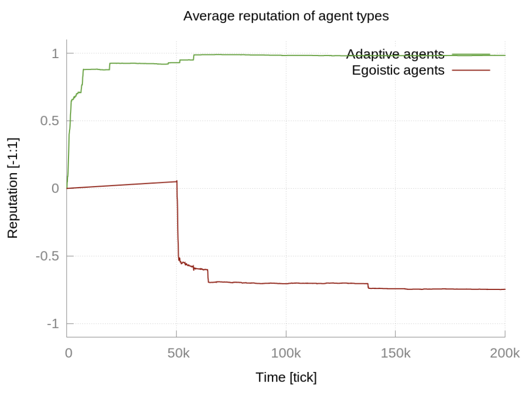
$$\forall r \in \Re_+^s : \mathfrak{T}^s(\mathfrak{A}^s(R_n^s, r)) \overset{?}{>} \mathfrak{T}^s(R_n^s) \vee \mathfrak{T}^s(R_n^s) \overset{?}{=} 1 \tag{111}$$

$$\begin{aligned}0 < \alpha < 1 \Rightarrow \frac{1}{\alpha} > 1 \Rightarrow (\frac{1}{\alpha} - 1) \cdot r_1 > 0, \\ 0 < p_1 \leq 1, 0 < p_2 \leq 1 \Rightarrow \\ \frac{p_1 \cdot \alpha + (1-\alpha) \cdot r_1 - p_2 \cdot \alpha}{p_1 \cdot \alpha + (1-\alpha) \cdot r_1 + p_2 \cdot \alpha} \overset{?}{>} \frac{p_1 - p_2}{p_1 + p_2} \vee \frac{p_1 - p_2}{p_1 + p_2} \overset{?}{=} 1 \\ \frac{p_1 + (\frac{1}{\alpha} - 1) \cdot r_1 - p_2}{p_1 + (\frac{1}{\alpha} - 1) \cdot r_1 + p_2} > \frac{p_1 - p_2}{p_1 + p_2} \vee \frac{p_1 - p_2}{p_1 + p_2} = 1 \quad \blacksquare\end{aligned} \tag{112}$$

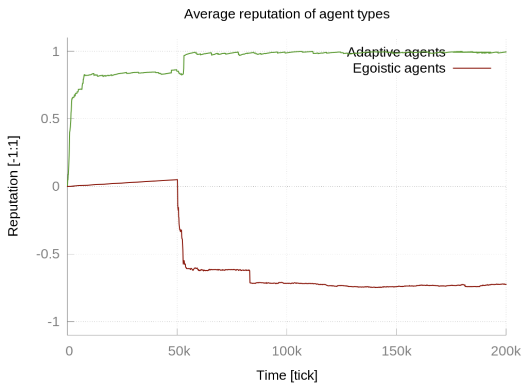
Figure 37: Reputation per metric. The system consists of 100 agents in the beginning and an attack starts at tick 50.000 with 100 attackers.



(a) Reputation for metric $\mathfrak{T}^c$. 1 is the best and -1 the worst reputation.



(b) Reputation for metric $\mathfrak{T}^w$. 1 is the best and -1 the worst reputation.



(c) Reputation for metric $\mathfrak{T}^s$. 1 is the best and -1 the worst reputation.

Metric	well-behaving agents	attackers
$\mathfrak{T}^c$	0.59±0.017	-0.77±0.034
$\mathfrak{T}^w$	0.98±0.0083	-0.74±0.049
$\mathfrak{T}^s$	0.99±0.0044	-0.75±0.047

(d) Reputation averaged per metric.

A.8. Evaluation and Discussion

In this section, we first compare the metrics according to their memory usage. Afterwards, we perform an experimental evaluation to compare their behaviour when used in a Trusted Desktop Grid.

A.8.1. Memory Requirements

To measure the memory usage of R, we define a function $\mathfrak{M}(R)$. Also, we aim at minimising $\mathfrak{M}$ because an agent typically needs to store a potentially large number of trust values (i.e. for every other agent it knows). $k \in \mathbb{N}$ represents the number of remembered ratings (not used by R^s). We assume single-precision 32-bit IEEE 754 floats which need 4 bytes of memory.

The binary metric R^b_n can be represented as boolean type and needs only one bit of memory per rating (113):

$$\mathfrak{M}(R^b_n) := \left\lceil \frac{k}{8} \right\rceil \text{ [byte]} \tag{113}$$

For the continuous and weighted metric, each rating is represented by a float value and, therefore, R^c and R^w need four bytes of memory multiplied by the amount of ratings k (114):

$$\mathfrak{M}(R^c_n) := \mathfrak{M}(R^w_n) := 4 \cdot k \text{ [byte]} \tag{114}$$

SES can be used to reduce the memory usage of a continuous metric R^c_n down to just one float and four bytes of memory. However, to keep properties of the weighted metric R^w_n, we need to store the weight of positive and negative ratings separately. Therefore, we need two times four bytes of memory (115).

$$\mathfrak{M}(R^s_n) := 8 \text{ [byte]} \tag{115}$$

A.8.2. Experimental Evaluation

In our TDG we use norms to communicate incentives and sanctions, which typically results in positive or negative ratings to the agents. Some actions are more desirable than others and, therefore, should result in a more positive rating than others.

For our evaluation, we introduce a norm that encourages agents to work for other agents with a lower reputation which is helpful to integrate newcomers into the system. A formalised

Figure 38: Norm used in the evaluation. It sanctions cancelling work and rewards completing work. In addition, working for submitters with lower reputation generates a stronger rating.

$$\textbf{context}\ \overbrace{\text{Worker}}^{\text{Target/Role}}\ \textbf{norm}\ \overbrace{\text{ReturnJob}}^{\text{Name}}:$$

$$\underbrace{\text{always}}_{\text{Pertinence Condition}}$$

$$\textbf{implies}\ \underbrace{\texttt{self.returnJob}(\texttt{job}, \texttt{requester}) = \texttt{true}}_{\text{Postcondition}}$$

$$\textbf{sanctioned}$$

$$\textbf{if}\ \underbrace{\text{violated}}_{\text{Default Sanction Condition}}$$

$$\textbf{then}\ \underbrace{\text{self.reputation += -cancelSanction}}_{\text{Sanction}}$$

$$\textbf{incentivised}$$

$$\textbf{if}\ \underbrace{\texttt{requester.reputation} < \text{highReputationThreshold}}_{\text{Sanction Condition}}$$

$$\textbf{then}\ \underbrace{\text{self.reputation += highWorkDoneIncentive}}_{\text{Incentive}}$$

$$\textbf{if}\ \underbrace{\texttt{requester.reputation} \geq \text{highReputationThreshold}}_{\text{Sanction Condition}}$$

$$\textbf{then}\ \underbrace{\text{self.reputation += lowWorkDoneIncentive}}_{\text{Incentive}}$$

version of the norm is shown in Figure 38. A worker will receive a rating for every completed work unit. However, if the requester has a reputation below a certain threshold, the worker receives a stronger rating. We then test how well the trust metrics $\mathfrak{T}^c$, $\mathfrak{T}^w$ and $\mathfrak{T}^s$ are suited. Additionally, workers will receive a bad rating if they reject work from agents with high reputation.

Each experiment is repeated 50 times and averaged. The initial system consists of 100 agents (Adaptive Agents) which are attacked by 100 malicious agents (Egoistic Agents) at tick 50.000. The attackers accept work but will return bogus results or cancel the jobs close to the deadline for 80% of the jobs. All results are shown in Figure 38d.

The reference experiment in Figure 38a shows reputation over the time for one exemplary experiment of the reference metric $\mathfrak{T}^n$. Well-behaving agents reach a very good speedup. When attackers join the system, the speedup for well-behaving agents decreases and malicious agents reach a speedup of about two. Unfortunately, in this scenario attackers do not get isolated (speedup stays significantly above a value of one). Additionally, well-behaving agents (i.e., Adaptive Agents) do not reach a reputation of about one.

Figure 38b and Figure 38c show reputation of the same experiment using the metrics from our approach. When applying $\mathfrak{T}^w$ the reputation of well-behaving agents reaches a value of nearly one as expected (see green line in Figure 38b). Additionally, the isolation of malicious agents works much better and the speedup decreases to a value of about one (i.e., the attackers no longer have any advantage from participating in the system). Agents get isolated when they lose their reputation and other agents are no longer obliged to work for them.

Metric $\mathfrak{T}^s$ behaves similarly but the values are significantly smoother (see Figure 38c). Here, the reputation reaches one and the desired isolation of attackers works as expected. Compared to $\mathfrak{T}^w$ the reputation of $\mathfrak{T}^s$ even grows closer to one (see Figure 38d).

In contrast to $\mathfrak{T}^c$, both $\mathfrak{T}^w$ and $\mathfrak{T}^s$ perform well in our scenario. The reputation of well-behaving agents reaches a value of nearly one (see Figure 38d) and attackers get isolated with a very low reputation value. $\mathfrak{T}^s$ should be preferred because it has lower memory requirements and fulfils our requirements unconditionally [C15].

List of Publications

Journals

[J1] Sven Tomforde, **Jan Kantert**, Christian Müller-Schloer, Sebastian Bödelt, and Bernhard Sick. “Comparing the Effects of Disturbances in Self-Adaptive Systems – A Generalised Approach for the Quantification of Robustness”. In: *Transactions on Computational Collective Intelligence (TCCI)* (2017), accepted for publication.

[J2] **Jan Kantert**, Sven Tomforde, Richard Scharrer, Susanne Weber, Sarah Edenhofer, and Christian Müller-Schloer. “Identification and Classification of Agent Behaviour at Runtime in Open, Trust-based Organic Computing Systems”. In: *Journal of Systems Architecture (JSA)* 75 (Apr. 2017), pp. 68–78. ISSN: 1383-7621.

[J3] **Jan Kantert**, Sven Tomforde, Melanie Kauder, Richard Scharrer, Sarah Edenhofer, Jörg Hähner, and Christian Müller-Schloer. “Controlling Negative Emergent Behavior by Graph Analysis at Runtime”. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 11.2 (June 2016), 7:1–7:34. ISSN: 1556-4665.

Book Chapters

[B4] **Jan Kantert**. “Trust Communities”. In: *Organic Computing – Technical Systems for Survival in the Real World*. Ed. by Christian Müller-Schloer and Sven Tomforde. Springer International Publishing, 2017, to appear. ISBN: 978-3-319-68476-5.

[B5] **Jan Kantert**, Sarah Edenhofer, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. “Normative Control – Controlling Open Distributed Systems with Autonomous Entities”. In: *Trustworthy Open Self-Organising Systems*. Ed. by Wolfgang Reif, Gerrit Anders, Hella Seebach, Jan-Philipp Steghöfer, Elisabeth André, Jörg Hähner, Christian Müller-Schloer, and Theo Ungerer. Vol. 7. Autonomic Systems. Springer International Publishing, 2016, pp. 89–126. ISBN: 978-3-319-29199-4.

[B6] Sarah Edenhofer, Sven Tomforde, **Jan Kantert**, Lukas Klejnowski, Yvonne Bernard, Jörg Hähner, and Christian Müller-Schloer. “Trust Communities: An Open, Self-Organised Social Infrastructure of Autonomous Agents”. In: *Trustworthy Open Self-Organising Systems*. Ed. by Wolfgang Reif, Gerrit Anders, Hella Seebach, Jan-Philipp Steghöfer, Elisabeth André, Jörg Hähner, Christian Müller-Schloer, and

Theo Ungerer. Vol. 7. Autonomic Systems. Springer International Publishing, 2016, pp. 127–152. ISBN: 978-3-319-29199-4.

Conferences

[C7] **Jan Kantert**, Sven Tomforde, Christian Müller-Schloer, Sarah Edenhofer, and Bernhard Sick. "Quantitative Robustness – A Generalised Approach to Compare the Impact of Disturbances in Self-Organising Systems". In: *Proceedings of the 9th International Conference on Agents and Artificial Intelligence (ICAART 2017).* Vol. 1. **(Best Student Paper Award)**. Porto, PT: SciTePress, Feb. 2017, pp. 39–50.

[C8] Sven Tomforde, **Jan Kantert**, and Bernhard Sick. "Measuring Self-Organisation at Runtime – A Quantification Method based on Divergence Measures". In: *Proceedings of the 9th International Conference on Agents and Artificial Intelligence (ICAART 2017).* Vol. 1. Porto, PT: SciTePress, Feb. 2017, pp. 96–106.

[C9] **Jan Kantert**, Christian Reinbold, Sven Tomforde, and Christian Müller-Schloer. "An Evaluation of Two Trust-Based Autonomic/Organic Grid Computing Systems for Volunteer-Based Distributed Rendering". In: *"Proceeding of the 13th IEEE International Conference on Autonomic Computing (ICAC 2016)".* Würzburg, Germany: IEEE, July 2016, pp. 137–146.

[C10] **Jan Kantert**, Sven Tomforde, Georg von Zengen, Susanne Weber, Lars Wolf, and Christian Müller-Schloer. "Improving Reliability and Reducing Overhead in Low-Power Sensor Networks using Trust and Forgiveness". In: *Proceeding of the 13th IEEE International Conference on Autonomic Computing (ICAC 2016).* Würzburg, Germany: IEEE, July 2016, pp. 325–333.

[C11] **Jan Kantert**, Melanie Kauder, Sarah Edenhofer, Sven Tomforde, and Christian Müller-Schloer. "Detecting Colluding Attackers in Distributed Grid Systems". In: *Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016).* Vol. 1. Rome, Italy: SciTePress, Feb. 2016, pp. 198–206. ISBN: 978-989-758-172-4.

[C12] Sarah Edenhofer, Christopher Stifter, Sven Tomforde, **Jan Kantert**, Christian Müller-Schloer, and Jörg Hähner. "Comparison of Surveillance Strategies to Identify Undesirable Behaviour in Multi-Agent Systems". In: *Proceedings of the 8th International*

Conference on Agents and Artificial Intelligence (ICAART 2016). Vol. 1. Rome, Italy: SciTePress, Feb. 2016, pp. 132–140. ISBN: 978-989-758-172-4.

[C13] **Jan Kantert**, Lukas Klejnowski, Sarah Edenhofer, Sven Tomforde, and Christian Müller-Schloer. "A Threatmodel for Trust-based Systems Consisting of Open, Heterogeneous, and Distributed Agents". In: *Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016).* Vol. 1. Rome, Italy: SciTePress, Feb. 2016, pp. 173–180. ISBN: 978-989-758-172-4.

[C14] **Jan Kantert**, Richard Scharrer, Sven Tomforde, Sarah Edenhofer, and Christian Müller-Schloer. "Runtime Clustering of Similarly Behaving Agents in Open Organic Computing Systems". In: *Proceedings of the 29th International Conference on Architecture of Computing Systems (ARCS 2016).* Ed. by F. Hannig, J.M.P. Cardoso, T. Pionteck, D. Fey, W. Schröder-Preikschat, and J Teich. Vol. 9637. LNCS. Nuremberg, Germany: Springer, Apr. 2016, pp. 321–333. ISBN: 978-3-319-30694-0.

[C15] **Jan Kantert**, Sarah Edenhofer, Sven Tomforde, and Christian Müller-Schloer. "Representation of Trust and Reputation in Self-Managed Computing Systems". In: *"Proceedings of the 13th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC 2015)".* Liverpool, UK: IEEE, Oct. 2015, pp. 1827–1834.

[C16] **Jan Kantert**, Henning Spiegelberg, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. "Distributed Rendering in an Open Self-Organised Trusted Desktop Grid". In: *"Proceeding of the 12th IEEE International Conference on Autonomic Computing (ICAC 2015)".* Grenobles, FR: IEEE, July 2015, pp. 267–272.

[C17] Sarah Edenhofer, Christopher Stifter, Uwe Jänen, **Jan Kantert**, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. "An Accusation-Based Strategy to Handle Undesirable Behaviour in Multi-Agent Systems". In: *"Proceeding of the 12th IEEE International Conference on Autonomic Computing (ICAC 2015)".* Grenobles, FR: IEEE, July 2015, pp. 243–248.

[C18] **Jan Kantert**, Sarah Edenhofer, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. "Defending Autonomous Agents Against Attacks in Multi-Agent Systems Using Norms". In: *Proceedings of the 7th International Conference on Agents and Artificial Intelligence (ICAART 2015).* Lisbon, Portugal: SciTePress, Jan. 2015, pp. 149–156.

[C19] **Jan Kantert**, Sergej Wildemann, Georg von Zengen, Sarah Edenhofer, Sven Tomforde, Lars Wolf, Jörg Hähner, and Christian Müller-Schloer. “Improving Reliability and Endurance using End-to-End Trust in Distributed Low-Power Sensor Networks”. In: *Proceedings of the 28th International Conference on Architecture of Computing Systems (ARCS 2015)*. Ed. by L.M. Pinho, W. Karl, A. Cohen, and U. Brinkschulte. Vol. 9017. LNCS. Springer, Mar. 2015, pp. 135–145. ISBN: 978-3-319-16085-6.

[C20] **Jan Kantert**, Sarah Edenhofer, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. “Detecting and Isolating Inconsistently Behaving Agents Using an Intelligent Control Loop”. In: *Proceedings of the 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2015)*. Colmar, FR: SciTePress, July 2015, pp. 246–253.

[C21] Sven Tomforde, **Jan Kantert**, Sebastian von Mammen, and Jörg Hähner. “Cooperative Self-Optimisation of Network Protocol Parameters at Runtime”. In: *Proceedings of the 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2015)*. Colmar, FR: SciTePress, July 2015, pp. 123–130.

[C22] **Jan Kantert**, Sven Tomforde, and Christian Müller-Schloer. “Addressing Challenges Beyond Classic Control with Organic Computing”. In: *Proceedings of the 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2015)*. Colmar, FR: SciTePress, July 2015, pp. 264–269.

[C23] **Jan Kantert**, Lukas Klejnowski, Yvonne Bernard, and Christian Müller-Schloer. “Influence of Norms on Decision Making in Trusted Desktop Grid Systems: Making Norms Explicit”. In: *Proceedings of the 6th International Conference on Agents and Artificial Intelligence (ICAART 2014)*. Vol. 2. Angers, France: SciTePress, Mar. 2014, pp. 278–283.

[C24] **Jan Kantert**, Hannes Scharf, Sarah Edenhofer, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. “A Graph Analysis Approach to Detect Attacks in Multi-Agent-Systems at Runtime”. In: *Proceedings of the Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2014)*. London, UK: IEEE, Sept. 2014, pp. 80–89.

[C25] **Jan Kantert**, Yvonne Bernard, Lukas Klejnowski, and Christian Müller-Schloer. “Estimation of Reward and Decision Making for Trust-Adaptive Agents in Normative Environments”. In: *Proceedings of the 27th International Conference on Architecture of Computing Systems (ARCS 2014)*. Ed. by Erik Maehle, Kay Römer, Wolfgang

Karl, and Eduardo Tovar. Vol. 8350. LNCS. Lübeck, Germany: Springer, Feb. 2014, pp. 49–59. ISBN: 978-3-319-04890-1.

[C26] **Jan Kantert**, Hannes Scharf, Sarah Edenhofer, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. “Implementing an Adaptive Higher Level Observer in Trusted Desktop Grid to Control Norms”. In: *Proceedings of the 11th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2014)*. Lisbon, Portugal: SciTePress, Sept. 2014, pp. 288–295.

Workshops

[W27] **Jan Kantert**, Sven Tomforde, Ada Diaconescu, and Christian Müller-Schloer. “Incentive-Oriented Task Assignment in Holonic Organic Systems”. In: *Proceedings of the 30th International Conference on Architecture of Computing Systems Workshops (ARCSW 2017)*. Berlin, Offenbach, DE: VDE Verlag GmbH, Apr. 2017, pp. 47–54. ISBN: 978-3-8007-4395-7.

[W28] Sarah Edenhofer, Christopher Stifter, Youssef Madkour, Sven Tomforde, **Jan Kantert**, Christian Müller-Schloer, and Jörg Hähner. “Bottom-up Norm Adjustment in Open, Heterogeneous Agent Societies”. In: *Proceedings of the 1st IEEE International Workshops on Foundations and Applications of Self-* Systems (FAS-W 2016)*. Augsburg, DE: IEEE, Sept. 2016, pp. 36–41.

[W29] **Jan Kantert**, Florian Reinhard, Georg von Zengen, Sven Tomforde, Lars Wolf, and Christian Müller-Schloer. “Combining Trust and ETX to Provide Robust Wireless Sensor Networks”. In: *Proceedings of the 29th International Conference on Architecture of Computing Systems Workshops (ARCSW 2016)*. Ed. by Ana Lucia Varbanescu. Nuremberg, Germany: VDE Verlag GmbH, Berlin, Offenbach, DE, Apr. 2016. Chap. 16, pp. 1–7. ISBN: 978-3-8007-4157-1.

[W30] Stefan Rudolph, **Jan Kantert**, Uwe Jänen, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. “Measuring Self-Organisation Processes in Smart Camera Networks”. In: *Proceedings of the 29th International Conference on Architecture of Computing Systems (ARCSW 2016)*. Ed. by Ana Lucia Varbanescu. Nuremberg, Germany: VDE Verlag GmbH, Berlin, Offenbach, DE, Apr. 2016. Chap. 14, pp. 1–6. ISBN: 978-3-8007-4157-1.

[W31] **Jan Kantert**. “Developing Robust Autonomous Open Distributed Systems using Guided Self-Organisation”. In: *”Organic Computing - Doctoral Dissertation Colloquium 2016 (Intelligent Embedded Systems)”*. Ed. by Bernhard Sick and Sven Tomforde. Kassel University Press, June 2016, pp. 57–68. ISBN: 978-3737603003.

[W32] **Jan Kantert**, Sven Tomforde, and Christian Müller-Schloer. “Measuring Self-Organisation in Distributed Systems by External Observation”. In: *Proceedings of the 28th International Conference on Architecture of Computing Systems Workshops (ARCSW 2015)*. Berlin, Offenbach, DE: VDE Verlag GmbH, Mar. 2015, pp. 1–8. ISBN: 978-3-8007-3657-7.

[W33] **Jan Kantert**, Christian Ringwald, Georg von Zengen, Sven Tomforde, Lars Wolf, and Christian Müller-Schloer. “Enhancing RPL for Robust and Efficient Routing in Challenging Environments”. In: *”Proceedings of the Ninth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2015)”*. Cambridge, MA, US: IEEE, Sept. 2015, pp. 7–12.

[W34] Adrian Calma, Martin Jänicke, **Jan Kantert**, Nils Kopal, Florian Siefert, and Sven Tomforde. “Horizontal Integration of Organic Computing and Control Theory Concepts”. In: *”Organic Computing - Doctoral Dissertation Colloquium 2015 (Intelligent Embedded Systems)”*. Ed. by Bernhard Sick and Sven Tomforde. Kassel University Press, June 2015, pp. 157–164. ISBN: 978-3-7376-0028-6.

[W35] **Jan Kantert**. “Guided Self-Organisation in Autonomous Open Distributed Systems”. In: *”Organic Computing - Doctoral Dissertation Colloquium 2015 (Intelligent Embedded Systems)”*. Ed. by Bernhard Sick and Sven Tomforde. Kassel University Press, June 2015, pp. 13–24. ISBN: 978-3-7376-0028-6.

[W36] **Jan Kantert**, Sarah Edenhofer, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. “Robust Self-Monitoring in Trusted Desktop Grids for Self-Configuration at Runtime”. In: *Proceedings of the Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2014)*. London, UK: IEEE, Sept. 2014, pp. 178–185.

[W37] Sarah Edenhofer, **Jan Kantert**, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. “Advanced Attacks to Trusted Communities in Multi-Agent Systems”. In: *Proceedings of the Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2014)*. London, UK: IEEE, Sept. 2014, pp. 186–191.

[W38] Jan-Philipp Steghöfer, Gerrit Anders, Wolfgang Reif, **Jan Kantert**, and Christian Müller-Schloer. “An Effective Implementation of Norms in Trust-Aware Open Self-Organising Systems”. In: *Proceedings of the Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2014)*. London, UK: IEEE, Sept. 2014, pp. 76–77.

[W39] **Jan Kantert**. “Norm-Based System Control in Multi-Agent Systems”. In: *”Organic Computing - Doctoral Dissertation Colloquium 2014 (Intelligent Embedded Systems)”*. Ed. by Bernhard Sick and Sven Tomforde. Kassel University Press, Oct. 2014, pp. 43–56. ISBN: 978-3-86219-832-0.

Demos and Posters

[O40] **Jan Kantert**, Sarah Edenhofer, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. “Norm-Based System Control in Distributed Low-Power Sensor Networks”. In: *Proceedings of the 28th International Conference on Architecture of Computing Systems Poster Session (ARCSP 2015)*. Mar. 2015, pp. 13–14.

[O41] **Jan Kantert**, Sebastian Bödelt, Sarah Edenhofer, Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. “Interactive Simulation of an Open Trusted Desktop Grid System with Visualisation in 3D”. In: *Proceedings of the Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2014)*. **(Best Demo Award)**. London, UK: IEEE, Sept. 2014, pp. 191–192.

[O42] **Jan Kantert**, Yvonne Bernard, Lukas Klejnowski, and Christian Müller-Schloer. “Interactive Graph View of Explicit Trusted Communities in an Open Trusted Desktop Grid System”. In: *Proceedings of the 7th IEEE International Conference on Self-Adaptation and Self-Organizing Systems Workshops (SASOW 2013)*. Philadelphia, USA: IEEE, Sept. 2013, pp. 13–14.

References

[43] Andrew S Tanenbaum. *Moderne Betriebssysteme*. Pearson Deutschland GmbH, 2009.

[44] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN: 0-8053-0177-1.

[45] Sven Tomforde, Holger Prothmann, Jürgen Branke, Jörg Hähner, Moez Mnif, Christian Müller-Schloer, Urban Richter, and Hartmut Schmeck. "Observation and Control of Organic Systems". In: *Organic Computing - A Paradigm Shift for Complex Systems*. Ed. by Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer. Basel, CH: Birkhäuser, 2011, pp. 325–338.

[46] Yao Wang and J. Vassileva. "Trust-Based Community Formation in Peer-to-Peer File Sharing Networks". In: *Proc. on Web Intelligence*. Beijing, China: IEEE, Sept. 2004, pp. 341–348.

[47] Micah Adler, Ying Gong, and Arnold L Rosenberg. "Optimal sharing of bags of tasks in heterogeneous clusters". In: *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*. ACM. 2003, pp. 1–10.

[48] Soledad Escolar Díaz, Jesús Carretero Pérez, Alejandro Calderón Mateos, Maria-Cristina Marinescu, and Borja Bergua Guerra. "A Novel Methodology for the Monitoring of the Agricultural Production Process Based on Wireless Sensor Networks". In: *Computers and Electronics in Agriculture* 76.2 (2011), pp. 252–265.

[49] Fiona Edwards Murphy, Michele Magno, Padraig Whelan, and Emanuel Popo Vici. "b+WSN: Smart Beehive for Agriculture, Environmental, and Honey Bee Health Monitoring – Preliminary Results and Analysis". In: *Sensors Applications Symposium (SAS), 2015 IEEE*. IEEE. 2015, pp. 1–6.

[50] F. Busching, W. Pottner, D. Brokelmann, G. Von Zengen, R. Hartung, K. Hinz, and L. Wolf. "A demonstrator of the GINSENG-approach to performance and closed loop control in WSNs". In: *2012 Ninth International Conference on Networked Sensing Systems (INSS)*. June 2012, pp. 1–2.

[51] Lukas Klejnowski. "Trusted Community: A Novel Multiagent Organisation for Open Distributed Systems". PhD thesis. Leibniz Universität Hannover, 2014.

[52] Ernst Fehr. "Human behaviour: don't lose your reputation". In: *Nature* 432.7016 (2004), pp. 449–450.

[53] Christiano Castelfranchi and Rino Falcone. *Trust Theory: A Socio-Cognitive and Computational Model*. Vol. 18. Chichester, UK: John Wiley & Sons, 2010. ISBN: 0470028750, 9780470028759.

[54] Jan-Philipp Steghöfer, Rolf Kiefhaber, Karin Leichtenstern, Yvonne Bernard, Lukas Klejnowski, Wolfgang Reif, Theo Ungerer, Elisabeth André, Jörg Hähner, and Christian Müller-Schloer. “Trustworthy Organic Computing Systems: Challenges and Perspectives”. In: *Proc. of ATC 2010*. Boston, MA: Springer, 2010, pp. 62–76.

[55] IEEE. *IEEE Standard for Local and metropolitan area networks – Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*. 2011.

[56] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, and R. Alexander. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550 (Proposed Standard). Internet Engineering Task Force, Mar. 2012.

[57] C.E. Perkins and E.M. Royer. “Ad-hoc On-Demand Distance Vector Routing”. In: *Second IEEE Workshop on Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA ’99*. Feb. 1999, pp. 90–100.

[58] T. Clausen and P. Jacquet. *RFC3626 Optimized Link State Routing Protocol (OLSR)*. 2003.

[59] Yvonne Bernard, Lukas Klejnowski, Emre Çakar, Jörg Hähner, and Christian Müller-Schloer. “Efficiency and Robustness Using Trusted Communities in a Trusted Desktop Grid”. In: *Proc. of SASO Workshops*. Michigan, US: IEEE, 2011, pp. 21–26.

[60] Janus B. Kristensen. *Big Buck Bunny 3D Rendering Exploration*. Blender Foundation. 2014. URL: `http://bbb3d.renderfarming.net/explore.html` (visited on 03/31/2017).

[61] Daniel Terdiman. *New technology revs up Pixar’s ’Cars 2’*. CNet. 2014. URL: `http://www.cnet.com/news/new-technology-revs-up-pixars-cars-2/` (visited on 03/31/2017).

[62] Andreas Noack. “Unified quality measures for clusterings, layouts, and orderings of graphs, and their application as software design criteria.” PhD thesis. Brandenburg University of Technology, 2007, pp. 1–289.

[63] Yvonne Bernard. “Trust-aware Agents for Self-Organising Computing Systems”. PhD thesis. Leibniz Universität Hannover, 2014.

[64] Cosimo Anglano, Massimo Canonico, Marco Guazzone, Marco Botta, Sergio Rabellino, Simone Arena, and Guglielmo Girardi. "Peer-to-Peer Desktop Grids in the Real World: The ShareGrid Project". In: *Proc. of CCGrid 2008* 8 (4 2008), pp. 609–614.

[65] Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. "Application-Specific Scheduling for the Organic Grid". In: *Proc. of GRID 2004 Workshops*. Washington, DC, USA: IEEE, 2004, pp. 146–155. ISBN: 0-7695-2256-4.

[66] David P. Anderson and Gilles Fedak. "The Computational and Storage Potential of Volunteer Computing". In: *Proc. of CCGRID 2006*. Singapore: IEEE, 2006, pp. 73–80. ISBN: 0-7695-2585-7.

[67] Raj Jain, Gojko Babic, Bhavana Nagendra, and Chi-Chung Lam. "Fairness, Call Establishment Latency and Other Performance Metrics". In: *ATM-Forum* 96.1173 (Aug. 1996), pp. 1–6. ISSN: 0163-6804.

[68] Alan Demers, Srinivasan Keshav, and Scott Shenker. "Analysis and Simulation of a Fair Queueing Algorithm". In: *Symposium Proceedings on Communications Architectures & Protocols*. SIGCOMM '89. New York, NY, USA: ACM, 1989, pp. 1–12. ISBN: 0-89791-332-9.

[69] Jon C.R. Bennett and Hui Zhang. "WF2Q: Worst-case Fair Weighted Fair Queueing". In: *INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*. Vol. 1. San Francisco, CA, USA: IEEE, Mar. 1996, pp. 120–128.

[70] Jeffrey S. Rosenschein and Gilad Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers*. Cambridge: MIT Press, 1994. ISBN: 0-262-18159-2.

[71] Carl Hewitt. "Open Information Systems Semantics for Distributed Artificial Intelligence". In: *Artificial intelligence* 47.1 (1991), pp. 79–106.

[72] Cosimo Anglano, John Brevik, Massimo Canonico, Dan Nurmi, and Rich Wolski. "Fault-aware Scheduling for Bag-of-Tasks Applications on Desktop Grids". In: *Proc. of GRID 2006*. Singapore: IEEE, 2006, pp. 56–63. ISBN: 1-4244-0343-X.

[73] SungJin Choi, HongSoo Kim, EunJoung Byun, MaengSoon Baik, SungSuk Kim, ChanYeol Park, and ChongSun Hwang. "Characterizing and Classifying Desktop Grid". In: *Proc. of CCGRID 2007*. Rio de Janeiro, Brazil: IEEE, 2007, pp. 743–748.

[74] Sungjin Choi, Rajkumar Buyya, Hongsoo Kim, and Eunjoung Byun. *A Taxonomy of Desktop Grids and its Mapping to State of the Art Systems*. Tech. rep. Grid Computing and Dist. Sys. Laboratory, The University of Melbourne, 2008, pp. 1–61.

[75] Marvin Karlins and Herbert I Abelson. "Persuasion: How opinions and attitudes are changed.." In: (1970).

[76] David Hume. *A Treatise of Human Nature, 2nd edn, ed. LA Selby-Bigge and PH Nidditch*. 1978.

[77] Vincent Buskens. "The social structure of trust". In: *Social networks* 20.3 (1998), pp. 265–289.

[78] Jordi Sabater and Carles Sierra. "Review on computational trust and reputation models". In: *Artificial intelligence review* 24.1 (2005), pp. 33–60.

[79] Sarvapali D. Ramchurn, Dong Huynh, and Nicholas R. Jennings. "Trust in multi-agent systems". In: *Knowl. Eng. Rev.* 19.1 (2004), pp. 1–25. ISSN: 0269-8889.

[80] Patricio Domingues, Bruno Sousa, and Luis Moura Silva. "Sabotage-tolerance and Trustmanagement in Desktop Grid Computing". In: *Future Generation Computer Systems* 23.7 (2007), pp. 904–912.

[81] Giovanni Sartor. *Legal Reasoning: A Cognitive Approach to Law*. Berlin Heidelberg, DE: Springer, 2005. ISBN: 1-4020-3387-7.

[82] Donald Nute. "Defeasible Logic". In: *Handbook of Logic in Artificial Intelligence and Logic Programming* 3 (1994), pp. 353–395.

[83] Donald Nute. "Defeasible Reasoning: A Philosophical Analysis in Prolog". In: *Aspects of Artificial Intelligence*. Berlin Heidelberg, DE: Springer, 1988, pp. 251–288.

[84] Donald Nute. "Defeasible Logic". In: *Proceedings of the Applications of Prolog 14th International Conference on Web Knowledge Management and Decision Support*. INAP'01. Berlin, Heidelberg: Springer, 2003, pp. 151–169. ISBN: 3-540-00680-X.

[85] David Billington. "Defeasible Logic is Stable". In: *Journal of Logic and Computation* 3.4 (1993), pp. 379–400.

[86] Andreea Urzică and Cristian Gratie. "Policy-Based Instantiation of Norms in MAS". In: *Intelligent Distributed Computing VI*. Ed. by Giancarlo Fortino, Costin Badica, Michele Malgeri, and Rainer Unland. Vol. 446. Studies in Computational Intelligence. Calabria, Italy: Springer, 2013, pp. 287–296.

[87] Tina Balke, Célia da Costa Pereira, Frank Dignum, Emiliano Lorini, Antonino Rotolo, Wamberto Vasconcelos, and Serena Villata. "Norms in MAS: Definitions and Related Concepts". In: *Normative Multi-Agent Systems*. Ed. by Giulia Andrighetto, Guido Governatori, Pablo Noriega, and Leendert W. N. van der Torre. Vol. 4. Dagstuhl Follow-Ups. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 1–31. ISBN: 978-3-939897-51-4.

[88] Georg Henrik von Wright. *Norms and Action: A Logical Enquiry*. London, UK: Routledge & Kegan Paul, 1963.

[89] Raimo Tuomela and Maj Bonnevier-Tuomela. "Norms and Agreements". In: *E. J. of Law, Philosophy and Computer Science* 5 (1995), pp. 41–46.

[90] Christopher D. Hollander and Annie S. Wu. "The Current State of Normative Agent-Based Systems". In: *Journal of Artificial Societies and Social Simulation* 14.2 (2011), p. 6. ISSN: 1460-7425.

[91] Munindar P Singh. "An Ontology for Commitments in Multiagent Systems". In: *Artificial Intelligence and Law* 7.1 (1999), pp. 97–113.

[92] Guido Governatori and Antonino Rotolo. "BIO Logical Agents: Norms, Beliefs, Intentions in Defeasible Logic". In: *Autonomous Agents and Multi-Agent Systems* 17.1 (2008), pp. 36–69. ISSN: 1387-2532.

[93] Alexander Artikis and Jeremy Pitt. "Specifying Open Agent Systems: A Survey". In: *Engineering Societies in the Agents World IX*. Ed. by Alexander Artikis, Gauthier Picard, and Laurent Vercouter. Vol. 5485. LNCS. Saint-Etienne, FR: Springer, 2009, pp. 29–45. ISBN: 978-3-642-02561-7.

[94] Guido Boella, Gabriella Pigozzi, and Leendert van der Torre. "Normative Systems in Computer Science - Ten Guidelines for Normative Multiagent Systems". In: *Normative Multi-Agent Systems*. Ed. by Guido Boella, Pablo Noriega, Gabriella Pigozzi, and Harko Verhagen. Dagstuhl Seminar Proceedings 09121. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009, pp. 1–21.

[95] Rosaria Conte, Cristiano Castelfranchi, and Frank Dignum. "Autonomous Norm Acceptance". In: *Intelligent Agents V: Agents Theories, Architectures, and Languages*. Ed. by JörgP. Müller, AnandS. Rao, and MunindarP. Singh. Vol. 1555. LNCS. Paris, France: Springer, 1999, pp. 99–112. ISBN: 978-3-540-65713-2.

[96] Bastin Tony Roy Savarimuthu and Stephen Cranefield. “Norm Creation, Spreading and Emergence: A Survey of Simulation Models of Norms in Multi-Agent Systems”. In: *Multiagent and Grid Systems* 7.1 (2011), pp. 21–54.

[97] Munindar P. Singh, Matthew Arrott, Tina Balke, Amit K. Chopra, Rob Christiaanse, Stephen Cranefield, Frank Dignum, Davide Eynard, Emilia Farcas, Nicoletta Fornara, Fabien Gandon, Guido Governatori, Hoa Khanh Dam, Joris Hulstijn, Ingolf Krueger, Ho-Pun Lam, Michael Meisinger, Pablo Noriega, Bastin Tony Roy Savarimuthu, Kartik Tadanki, Harko Verhagen, and Serena Villata. “The Uses of Norms”. In: *Normative Multi-Agent Systems*. Vol. 4. Dagstuhl Follow-Ups. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 191–229. ISBN: 978-3-939897-51-4.

[98] Garrett Hardin. “The Tragedy of the Commons”. In: *Science* 162.3859 (1968), pp. 1243–1248.

[99] Elinor Ostrom. *Governing the Commons: The Evolution of Institutions for Collective Action*. Cambridge, US: Cambridge university press, 1990.

[100] Jeremy Pitt, Julia Schaumeier, and Alexander Artikis. “The Axiomatisation of Socio-Economic Principles for Self-Organising Systems”. In: *Self-Adaptive and Self-Organizing Systems (SASO), 2011 Fifth IEEE International Conference on*. Michigan, US: IEEE, Oct. 2011, pp. 138–147.

[101] Duncan S Callaway, Mark EJ Newman, Steven H Strogatz, and Duncan J Watts. “Network robustness and fragility: Percolation on random graphs”. In: *Physical review letters* 85.25 (2000), p. 5468.

[102] Jean-Jacques E Slotine, Weiping Li, et al. *Applied nonlinear control*. Vol. 199. 1. Prentice-Hall Englewood Cliffs, NJ, 1991.

[103] Genichi Taguchi. “Robust technology development”. In: *Mechanical Engineering-CIME* 115.3 (1993), pp. 60–63.

[104] Armin Scholl et al. *Robuste Planung und Optimierung: Grundlagen, Konzepte und Methoden; Experimentelle Untersuchungen*. Tech. rep. Darmstadt Technical University, Department of Business Administration, Economics and Law, Institute for Business Studies (BWL), 2000.

[105] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice-Hall, Inc., 1994.

[106] Daniel A. Menascé, Mohamed N. Bennani, and Honglei Ruan. “Self-star Properties in Complex Information Systems: Conceptual and Practical Foundations”. In: ed. by Ozalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. Chap. On the Use of Online Analytic Performance Models, in Self-Managing and Self-Organizing Computer Systems, pp. 128–142. ISBN: 978-3-540-32013-5.

[107] *2015 IEEE International Conference on Autonomic Computing, Grenoble, France, July 7-10, 2015*. IEEE Computer Society, 2015. ISBN: 978-1-4673-6971-8.

[108] *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems, Cambridge, MA, USA, September 21-25, 2015*. IEEE Computer Society, 2015. ISBN: 978-1-4673-7535-1.

[109] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter. “Adaptivity and Self-organisation in Organic Computing Systems”. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 5.3 (2010), pp. 1–32. ISSN: 1556-4665.

[110] F. Nafz, H. Seebach, J-P. Steghöfer, G.t Anders, and W. Reif. “Constraining Self-organisation Through Corridors of Correct Behaviour: The Restore Invariant Approach”. In: *Organic Computing – A Paradigm Shift for Complex Systems*. Ed. by C. Müller-Schloer, H. Schmeck, and T. Ungerer. Autonomic Systems. Basel, Switzerland: Birkhäuser Verlag, 2011, pp. 79–93.

[111] Benedikt Eberhardinger, Gerrit Anders, Hella Seebach, Florian Siefert, and Wolfgang Reif. “A Research Overview and Evaluation of Performance Metrics for Self-Organization Algorithms”. In: *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASO Workshops 2015, Cambridge, MA, USA, September 21-25, 2015*. 2015, pp. 122–127.

[112] Richard Holzer and Hermann de Meer. “Methods for Approximations of Quantitative Measures in Self-Organizing Systems”. In: *Self-Organizing Systems - 5th International Workshop, IWSOS 2011, Karlsruhe, Germany, February 23-24. 2011. Proceedings*. 2011, pp. 1–15.

[113] Richard Holzer and Hermann de Meer. “Quantitative Modeling of Self-organizing Properties”. In: *Self-Organizing Systems, 4th IFIP TC 6 International Workshop, IWSOS 2009, Zurich, Switzerland, December 9-11, 2009. Proceedings*. 2009, pp. 149–161.

[114] Giovanna Di Marzo Serugendo. “Stabilization, Safety, and Security of Distributed Systems: 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009. Proceedings”. In: ed. by Rachid Guerraoui and Franck Petit. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. Chap. Robustness and Dependability of Self-Organizing Systems - A Safety Engineering Perspective, pp. 254–268. ISBN: 978-3-642-05118-0.

[115] Jens Nimis and Peter C Lockemann. “Robust multi-agent systems: The transactional conversation approach”. In: *First International Workshop on Safety and Security in Multiagent Systems (SASEMAS’04), S.* 2004, pp. 73–84.

[116] Saurabh Ganeriwal, Laura K. Balzano, and Mani B. Srivastava. “Reputation-based Framework for High Integrity Sensor Networks”. In: *ACM Trans. Sen. Netw.* 4.3 (June 2008), 15:1–15:37. ISSN: 1550-4859.

[117] N. Leligou, L. Sarakis, P. Trakadas, V. Gay, and K. Georouleas. *Design Principles of Trust-aware Routing Protocol supporting Virtualization*. Deliverable D4.1. 2011.

[118] Guoxing Zhan, Weisong Shi, and J. Deng. “Design and Implementation of TARF: A Trust-Aware Routing Framework for WSNs”. In: *Dependable and Secure Computing, IEEE Transactions on* 9.2 (Mar. 2012), pp. 184–197. ISSN: 1545-5971.

[119] *TORQUE Open-Source Resource Manager*. Adaptive Computing. URL: `http://www.adaptivecomputing.com/products/open-source/torque/` (visited on 03/31/2017).

[120] Douglas Thain, Todd Tannenbaum, and Miron Livny. “Distributed computing in practice: the Condor experience.” In: *Concurrency - Practice and Experience* 17.2-4 (2005), pp. 323–356.

[121] Matthias Bonn and Hartmut Schmeck. “The joschka system: organic job distribution in heterogeneous and unreliable environments”. In: *International Conference on Architecture of Computing Systems*. Springer. 2010, pp. 73–86.

[122] David P. Anderson. “BOINC: A System for Public-Resource Computing and Storage”. In: *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. GRID ’04. Washington, DC, USA: IEEE, 2004, pp. 4–10. ISBN: 0-7695-2256-4.

[123] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. “SETI@home: An Experiment in Public-Resource Computing”. In: *Communications of the ACM* 45.11 (2002), pp. 56–61.

[124] J Aasi, J Abadie, BP Abbott, R Abbott, TD Abbott, M Abernathy, T Accadia, F Acernese, C Adams, T Adams, et al. "Einstein@ Home all-sky search for periodic gravitational waves in LIGO S5 data". In: *Physical Review D* 87.4 (2013), p. 042001.

[125] S.D. Corcoran, M.T. Kalmbach, E.R. Larese, J.W. Patterson, and K. Wendzel. *Establishing a bi-directional grid computing network*. US Patent App. 12/146,038. Dec. 2009.

[126] Robert G Brown. "Exponential Smoothing for predicting demand". In: *Operations Research*. Vol. 5. 1. Inst Operations Research Management Sciences 901 Elkridge Landing RD, STE 400, Linthicum HTS, MD 21090-2909. 1957, pp. 145–145.

[127] Asimina Vasalou, Jeremy Pitt, and Guillaume Piolle. "From Theory to Practice: Forgiveness as a Mechanism to Repair Conflicts in CMC". In: *Trust Management*. Ed. by Ketil Stølen, WilliamH. Winsborough, Fabio Martinelli, and Fabio Massacci. Vol. 3986. LNCS. Springer, 2006, pp. 397–411. ISBN: 978-3-540-34295-3.

[128] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-37940-6.

[129] Asimina Vasalou, Astrid Hopfensitz, and Jeremy V Pitt. "In praise of forgiveness: Ways for repairing trust breakdowns in one-off online interactions". In: *International Journal of Human-Computer Studies* 66.6 (2008), pp. 466–480.

[130] Hartmut Schmeck, Christian Müller-Schloer, Emre Çakar, Moez Mnif, and Urban Richter. "Adaptivity and self-organization in organic computing systems". In: *ACM Trans. Auton. Adapt. Syst.* 5 (3 Sept. 2010), 10:1–10:32. ISSN: 1556-4665.

[131] Stanley Wasserman. *Social network analysis: Methods and applications*. Vol. 8. Cambridge, US: Cambridge university press, 1994.

[132] Duncan J. Watts and Steven H. Strogatz. "Collective dynamics of 'small-world' networks". In: *Nature* 393 (June 1998), pp. 440–442.

[133] Jon M Kleinberg. "Authoritative sources in a hyperlinked environment". In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 604–632.

[134] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual Web search engine". In: *Computer networks and ISDN systems* 30.1 (1998), pp. 107–117.

[135] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. "Fast unfolding of communities in large networks". In: *J. of Statistical Mechanics: Theory and Experiment* 2008.10 (2008), P10008.

[136] Mark E. J. Newman. "Analysis of weighted networks". In: *Phys. Rev. E* 70 (5 Nov. 2004), p. 056131.

[137] Mohamed A Khamsi and William A Kirk. *An introduction to metric spaces and fixed point theory*. Vol. 53. Chichester, UK: John Wiley & Sons, 2011.

[138] Nicholas Jardine and Robin Sibson. *Mathematical taxonomy*. Chichester, UK: John Wiley & Sons, 1971. ISBN: 978-0471440505.

[139] Stijn Marinus Van Dongen. "Graph clustering by flow simulation". PhD thesis. Utrecht University, 2001.

[140] Satu Elisa Schaeffer. "Graph clustering". In: *Computer Science Review* 1.1 (2007), pp. 27–64.

[141] Rui Xu and Donald Wunsch. "Survey of clustering algorithms". In: *Neural Networks, IEEE Transactions on* 16.3 (2005), pp. 645–678.

[142] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. "BIRCH: an efficient data clustering method for very large databases". In: *ACM SIGMOD Record*. Vol. 25. Montreal, Canada: ACM, 1996, pp. 103–114.

[143] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[144] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors". In: *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462. ISBN: 0-7695-2260-2.

[145] P. Thubert. *Objective Function Zero for the Routing Protocol for Low-Power and Lossy Networks (RPL)*. RFC 6552 (Proposed Standard). Internet Engineering Task Force, Mar. 2012.

[146] J. Hui and JP. Vasseur. *The Routing Protocol for Low-Power and Lossy Networks (RPL) Option for Carrying RPL Information in Data-Plane Datagrams*. RFC 6553 (Proposed Standard). Internet Engineering Task Force, Mar. 2012.

[147] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard). Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112. Internet Engineering Task Force, Dec. 1998.

[148] F. Österlind. *A Sensor Network Simulator for the Contiki OS*. Tech. rep. Swedish Institute of Computer Science, May 2006.

[149] Douglas S. J. De Couto. “High-Throughput Routing for Multi-Hop Wireless Networks”. PhD thesis. Massachusetts Institute of Technology, 2004.

[150] William S Levine. *The Control Handbook*. CRC press, 1996. ISBN: 0-8493-8570-9.

Wissenschaftlicher Werdegang

Persönliche Daten

Name:	Jan Kantert
Anschrift:	Dieterichsstr. 13 30159 Hannover
Telefon:	0511 762-19726
E-Mail:	kantert@sra.uni-hannover.de
Geburtsdatum/-ort:	21.04.1987 in Hannover

Schulische Ausbildung

1994-1998	kath. Grundschule Celle
1998-2000	Orientierungsstufe an der Heese Celle
2000-2004	Herman Billung Gymnasium Celle, Abschluss: erweiterter Realschulabschluss
2004-2007	Axel-Bruns-Schule Celle, Abschluss: Abitur

Studium

2007-2010	Leibniz Universität Hannover, Informatik (B.Sc.)
2010-2011	Leibniz Universität Hannover, Informatik (M.Sc.)
2013-2017	Wissenschaftlicher Mitarbeiter am Institut für System und Rechnerarchitektur (SRA), Hannover