Byzantine Fault Tolerance in Distributed Constraint Optimization Problems

Koji Noshiro
University of Tsukuba
Tsukuba, Japan
noshiro@mas.cs.tsukuba.ac.jp

Koji Hasebe
University of Tsukuba
Tsukuba, Japan
hasebe@cs.tsukuba.ac.jp

ABSTRACT

Fault tolerance is crucial for the reliability of multi-agent system applications. This paper addresses fault-tolerant algorithms for solving distributed constraint optimization problems (DCOPs) in the presence of Byzantine faults, the most critical type of fault. First, we define a novel class of DCOPs to handle faulty agents, namely a Fault-Tolerant DCOP (FT-DCOP). This class extends the standard DCOP formulation to explicitly introduce the concept of faulty agents. Additionally, we prove an impossibility result for FT-DCOPs regarding the allowable proportion of faulty agents, revealing fundamental limitations. Furthermore, we propose a fault-tolerant algorithm for solving FT-DCOPs that combines the well-known Max-Sum algorithm with replication. Unlike general replication approaches that rely on complex Byzantine consensus protocols and strong assumptions, the proposed algorithm employs a simple verification step based on a synchronous message-passing mechanism and the deterministic nature of Max-Sum, thereby avoiding Byzantine consensus and relaxing the assumptions. Finally, we experimentally evaluate the performance of the standard Max-Sum and the proposed algorithm in the presence of Byzantine faulty agents. The results demonstrate that the standard Max-Sum algorithm is vulnerable to faulty agents and generates low-quality solutions, while the proposed algorithm is resilient and matches the fault-free performance of Max-Sum with manageable communication overhead.

KEYWORDS

Distributed Constraint Optimization; Byzantine Fault Tolerance; Replication

ACM Reference Format:
Koji Noshiro and Koji Hasebe. 2026. Byzantine Fault Tolerance in Distributed Constraint Optimization Problems. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 9 pages. https://doi.org/10.65109/JZVR6522

1 INTRODUCTION

A Distributed Constraint Optimization Problem (DCOP) [7, 17] is a framework for formulating multi-agent coordination problems. In this framework, multiple agents cooperate to find an optimal solution that maximizes (or minimizes) the sum of utility (or cost) functions defined over their variable domains. DCOPs have been applied in various areas, such as event scheduling [14], sensor networks [14, 23], and smart environment configuration [21].

Over the decades, a variety of algorithms have been proposed for solving DCOPs, and they are typically classified into two categories: complete and incomplete. Complete algorithms, such as ADOPT [17] and DPOP [19], guarantee optimal solutions but incur significant computational costs due to the NP-hardness of DCOPs [17]. In contrast, incomplete algorithms, such as MGM [13], DSA [23], and Max-Sum [6], efficiently find approximate solutions. For instance, the Max-Sum algorithm approximates the objective function through message passing to find solutions. Although these DCOP algorithms solve DCOPs effectively, most rely on an assumption that all agents correctly adhere to their prescribed procedure.

In practical systems, however, agents may malfunction and deviate from the algorithm. Moreover, malicious attackers may control agents to disrupt the optimization process. In the context of distributed systems, such unintended behaviors of agents are modeled as *Byzantine faults* [10], where agents can take arbitrary actions.

These Byzantine faulty agents can significantly compromise the performance of existing DCOP algorithms. For example, let us consider an event scheduling problem where multiple trucks (agents) must coordinate schedules to avoid concentration at particular terminals [1, 20]. During algorithm execution, faulty agents may send erroneous messages to correct agents, which can yield suboptimal solutions that violate constraints, leading to infeasible schedules for trucks or excessive concentration. As such results can cause critical economic losses and degrade system reliability, the vulnerability to Byzantine faults hinders practical applications of DCOP algorithms.

To achieve fault tolerance against such Byzantine faults in distributed systems, researchers have explored several approaches. One major approach is *replication*, which involves replicating computational processes across multiple agents to mask the effects of faulty agents. Rust et al. [21] proposed a method for incorporating this technique into DCOP algorithms. Their work, however, focused on handling agent removal such as network disconnection and crashes, rather than tolerating Byzantine faults. Moreover, that work assumed that correct agents can identify the faulty agents before responding, which is restrictive in the presence of Byzantine faults, as such identification is non-trivial.

This paper addresses Byzantine fault tolerance in DCOP algorithms. In contrast to previous work, we assume that correct agents do not know the identities of faulty agents in advance. To the best of our knowledge, this is the first study to tackle Byzantine faults in DCOP algorithms where agents may deviate arbitrarily from their prescribed procedures.

Our contribution is fourfold: (1) we introduce a novel DCOP class for handling faulty agents; (2) we establish an impossibility

This work is licensed under a Creative Commons Attribution International 4.0 License.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus. © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). https://doi.org/10.65109/JZVR6522

result for solving that problem; (3) we propose a fault-tolerant DCOP algorithm based on the Max-Sum algorithm; and (4) we demonstrate the vulnerability of Max-Sum to Byzantine faults and the resilience of the proposed algorithm through experiments.

First, we introduce a Fault-Tolerant DCOP (FT-DCOP), a novel class of DCOPs designed to accommodate faulty agents. This new formulation extends the standard DCOP model by explicitly incorporating the notion of faulty agents. In FT-DCOPs, correct agents aim to find optimal solutions while tolerating interference from faulty agents. In addition to the problem definition, we also define the concept of an ideal fault-tolerant algorithm for solving FT-DCOPs.

Furthermore, we prove an impossibility result for FT-DCOPs. It demonstrates that for any utility function, more than half of the agents with knowledge of that function must be correct. This impossibility result is crucial for clarifying the fundamental limitations of the problem, as is common in research on Byzantine fault-tolerant distributed systems, such as consensus problems [10, 12].

Additionally, we propose a fault-tolerant DCOP algorithm for solving FT-DCOPs, namely Replicated-Max-Sum (Repl-Max-Sum). This algorithm incorporates replication into the Max-Sum algorithm, while avoiding the complexity and strong assumptions of naive replication approaches under the Byzantine fault model. In particular, general replication methods [3, 16, 22] require complex Byzantine consensus protocols and rely on restrictive assumptions to circumvent impossibility results for consensus [10, 12]. To avoid these requirements, Repl-Max-Sum employs synchronous message passing and introduces a simple verification step for replication. This design effectively masks the influence of faulty agents, allowing correct agents to obtain approximate solutions.

Finally, we experimentally evaluate the performance of Repl-Max-Sum and the standard Max-Sum in a setting where Byzantine faulty agents send random messages to correct agents. In the experiments, we used two types of problems: graph coloring problems, which are common benchmarks for DCOP algorithms, and truck appointment scheduling, which represents a more practical scenario. The results demonstrate that Repl-Max-Sum masked the effects of faulty agents and obtained solutions identical to those in fault-free cases, while incurring manageable communication overhead. In contrast, the standard Max-Sum was disrupted by faulty agents, resulting in low-quality solutions that violate given constraints. These results highlight the significance of fault tolerance in DCOP algorithms.

The remainder of this paper is organized as follows. Section 2 reviews related work. Next, Section 3 presents the necessary preliminaries. Subsequently, Section 4 defines our novel problem class and presents an associated impossibility result. Then, Section 5 details the proposed algorithm, and Section 6 evaluates its performance. Finally, Section 7 concludes the paper and discusses future work.

2 RELATED WORK

This section reviews existing work on DCOPs involving faulty or adversarial agents, as Byzantine faults encompass both behaviors.

Several studies have explored DCOP algorithms for environments where agents may cease to function. Self-stabilizing Distributed Pseudo-tree Optimization Procedure (S-DPOP) [18] and Support-Based Distributed Optimization (SBDO) [2] handle DCOPs with dynamic configurations, such as varying sets of participating agents. These algorithms can achieve (near-)optimal solutions even when some agents stop functioning. From a communication perspective, Zivan et al. [24] analyzed the effects of message delays and loss on Max-Sum variants, which is relevant to scenarios where agents suffer from link failures.

Rust et al. [21] introduced a replication technique into DCOP algorithms to achieve resilience to the dynamic nature of smart home environments, where devices can be added or removed. To address this challenge, they proposed the concept of *k-resilience*, a property ensuring that a system can be repaired even after up to k agents are removed. The authors also proposed methods for determining an efficient distribution of replicas across agents to guarantee k-resilience.

These previous studies, however, only considered a limited type of fault, such as agents that stop functioning. Consequently, the algorithms are vulnerable to Byzantine faults, where agents can maliciously exhibit incorrect behaviors.

Other studies have addressed DCOPs with adversarial agents. For example, a Quantified DCOP (Q-DCOP) [15] is a class of DCOPs that models systems with adversarial agents that do not cooperate with others. In this model, variables associated with the adversarial agents can adopt arbitrary assignments. The authors proposed algorithms for calculating bounds on the utilities (or costs) of optimal solutions under the assumption of such arbitrary variable assignments. Furthermore, Lass et al. [11] proposed a method for obtaining robust solutions against faulty agents that adopt worst-case assignments, assuming prior identification of faulty agents.

Although the aforementioned studies provide a degree of robustness against faults, they rely on strong assumptions that hinder their practical application in systems requiring high reliability. First, the existing algorithms assume that no agents send incorrect messages. That is, they lack resilience to Byzantine faulty agents that can deviate from their intended procedure. Second, they assume that correct agents can identify faulty (or adversarial) agents in advance. However, identifying Byzantine faults during the executions of DCOP algorithms is non-trivial.

In contrast to previous studies, this paper addresses Byzantine faults that are not known to correct agents beforehand and proposes a DCOP algorithm to tolerate them. Moreover, to clarify the objective of such fault-tolerant algorithms, we define a novel class of DCOPs that explicitly models the presence of faulty agents.

3 PRELIMINARIES

In this section, we define standard DCOPs and one of their primary representations, namely factor graphs. Additionally, we describe the Max-Sum algorithm, which serves as the foundation for our proposed algorithm.

3.1 DCOP

A DCOP is formally defined as a tuple (A, X, D, F, α). Here, $A = \{a_1, \ldots, a_{|A|}\}$ is a finite set of agents, and $X = \{x_1, \ldots, x_{|X|}\}$ is a finite set of variables. $D = \{D_1, \ldots, D_{|X|}\}$ is a set of finite domains, where D_i is the domain for variable x_i. $F = \{f_1, \ldots, f_{|F|}\}$ is a finite set of utility functions, where each utility function f_i :

$\prod_{x_j \in \mathbf{x}^i} D_j \to \mathbb{R}^+ \cup \{0\}$ has a scope $\mathbf{x}^i \subseteq X$. Finally, the function $\alpha : X \to A$ assigns each variable to an agent. In the standard DCOP model, variable x_i is controlled exclusively by agent $a_j = \alpha(x_i)$.

The objective in a DCOP is to find a variable assignment, σ^*, that maximizes the sum of all utility functions. This optimal solution is formally defined as:

$$\sigma^* \in \underset{\sigma \in \prod_{x_j \in X} D_j}{\operatorname{argmax}} \sum_{f_i \in F} f_i\left(\sigma_{\mathbf{x}^i}\right), \quad (1)$$

where $\sigma_{\mathbf{x}^i}$ is the projection of the assignment σ onto the variables in the scope of f_i.

3.2 Factor Graph

A factor graph [6, 7] is a primary graphical representation of a DCOP. It is a bipartite graph composed of two types of nodes: variable nodes and function nodes. For a given DCOP (A, X, D, F, α), its corresponding factor graph $G = (V, E)$ must satisfy the following conditions:

(1) The set $V = X \cup F$ of nodes consists of variable nodes X and function nodes F.
(2) The set E of edges consists of undirected edges, where an edge (x_i, f_j) exists in E if and only if the variable x_i is in the scope of the utility function f_j (i.e., $x_i \in \mathbf{x}^j$).

For simplicity, we reuse symbols from the DCOP definition for the factor graph. The sets of variable nodes and function nodes are denoted by X and F, respectively. Accordingly, an individual variable node is denoted by x_i and a function node by f_j. The set of neighboring nodes of a node $v \in V$ is denoted by $N(v) \subseteq V$.

3.3 Max-Sum

The Max-Sum algorithm [6] is a well-known incomplete DCOP algorithm that operates on factor graphs. In this algorithm, variable and function nodes, each operated by its respective agent, iteratively exchange messages to find an optimal solution. Max-Sum is theoretically proven to converge to an optimal solution on acyclic factor graphs, although convergence is not guaranteed for those with cycles.

During its execution, both types of nodes send distinct messages to their neighbors. For an assignment $d_i \in D_i$ of a variable x_i, two types of messages are recursively defined. The message $q_{i,j}(d_i)$ is sent from a variable node x_i to a function node f_j, while the message $r_{j,i}(d_i)$ is sent from a function node f_j to a variable node x_i. These messages are defined as follows:

$$q_{i,j}\left(d_i\right) = c_{i,j} + \sum_{f_{j'} \in N(x_i) \setminus \{f_j\}} r_{j',i}\left(d_i\right), \quad (2)$$

where $c_{i,j}$ is a normalization constant that ensures $\sum_{d \in D_i} q_{i,j}(d) = 0$, and

$$r_{j,i}\left(d_i\right) = \max_{\sigma \in \Sigma^j_{-i}} \left(f_j\left(d_i, \sigma\right) + \sum_{x_{i'} \in N(f_j) \setminus \{x_i\}} q_{i',j}\left(\sigma_{x_{i'}}\right) \right), \quad (3)$$

where $\Sigma^j_{-i} = \prod_{x_{i'} \in \mathbf{x}^j \setminus \{x_i\}} D_{i'}$ and $\sigma_{x_{i'}}$ represents the assignment of variable $x_{i'}$ within σ. Each node iteratively updates its outgoing messages based on incoming messages from its neighbors and sends them until termination. Finally, each variable node x_i computes $d^* \in \arg\max_{d \in D_i} \sum_{f_j \in N(x_i)} r_{j,i}(d)$, which yields an approximate solution to the DCOP.

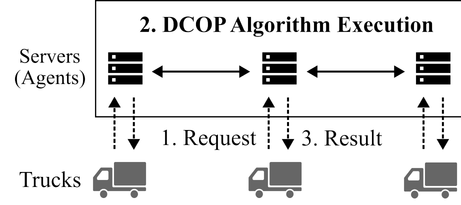

Figure 1: Truck Appointment Scheduling via DCOP

4 FAULT-TOLERANT DCOP

In this section, we define a novel class of DCOPs, termed a Fault-Tolerant DCOP (FT-DCOP), to introduce the concept of fault tolerance. First, we describe a motivating scenario in this paper. Subsequently, we present the definition of FT-DCOPs and then discuss its applicability and limitations. Furthermore, we establish an impossibility result for FT-DCOPs.

4.1 Motivating Scenario

We first describe a motivating scenario: truck appointment scheduling [1, 20], an event scheduling problem for delivery trucks. Let us consider a system where distributed servers, referred to as agents, execute a DCOP algorithm to determine when trucks visit their destination terminals. After the algorithm execution, agents inform trucks of the computed schedules, and the trucks deliver according to them. Figure 1 illustrates the system architecture.

In this system, agents may deviate from the DCOP algorithm by the effects of malfunction or malicious attacks. We refer to agents exhibiting such deviation collectively as *faulty agents*. They may stop functioning and send incorrect information to other correct agents, which can severely compromise the algorithm performance.

To avoid this disruption, fault-tolerant DCOP algorithms enable correct agents to compute optimal solutions and offer them to trucks. This tolerance can be acquired because faulty agents can only affect the procedure of the DCOP algorithm, that is, they cannot directly control the decision of trucks. In this paper, we assume systems with such characteristics and focus on the tolerance to faulty behaviors during the execution of DCOP algorithms.

4.2 Definition of FT-DCOP

An FT-DCOP is defined as a tuple $(A, X, D, F, \alpha, \gamma)$. The components X, D and F are identical to those in the standard DCOPs. In an FT-DCOP, agents are classified as either *correct* or *faulty*. An agent is considered correct if it properly executes the given algorithm; otherwise, it is faulty. We assume that faulty agents exhibit Byzantine behaviors, i.e., they can behave arbitrarily. The sets of correct and faulty agents are denoted by $H \subseteq A$ and $B \subset A$, respectively, where $A = H \cup B$ and $H \cap B = \emptyset$. Furthermore, we modify the definition of the function α to map a variable to a set of agents, i.e., $\alpha : X \to 2^A$. In this formulation, the function represents a set of agents responsible for a variable, rather than an agent exclusively controlling it. Additionally, we define $\gamma : F \to 2^A$ as a function that maps each utility function to a set of agents that know its definition.

This notion is essential for proving our impossibility result. As with the standard DCOPs, correct agents aim to find an optimal solution defined in Eq. (1).

We adopt the standard communication model for DCOPs. Specifically, we assume that messages are eventually delivered to their intended recipients within a finite delay, and that recipients can reliably identify the sender of each message.

Applicability. FT-DCOPs can be applied to various systems where faulty agents are unable to manipulate the final variable assignments. This property holds if agents executing DCOP algorithms are distinct from actors that make decisions based on the obtained solutions, as in the aforementioned scenario. Such separation allows our model to employ replication, where multiple agents perform computations for the same variables. Although this approach differs from standard DCOP applications, it is essential for Byzantine fault tolerance. Representative examples with the property include edge computing systems that optimize action plans and schedules for multiple robots and IoT (the Internet of Things) devices [4, 8, 9].

Limitations. If faulty agents can directly influence variable assignments, the FT-DCOP formulation may be unsuitable. For instance, in systems where mobile robots execute DCOP algorithms, faulty robots can ignore their obtained solutions and adopt other assignments for their variables (e.g., positions), potentially affecting utilities of correct robots. Handling such scenarios would require fundamentally different algorithmic mechanisms. Thus, we focus on the constrained setting above, which enables us to establish essential theoretical foundations for fault-tolerant DCOP algorithms.

4.3 Impossibility of FT-DCOP

This subsection establishes a fundamental impossibility on solving FT-DCOPs. Intuitively, an optimal solution cannot be guaranteed if, for any given utility function, there is not a sufficient majority of correct agents with knowledge of it.

To formalize this impossibility, we first define the notion of an ideal fault-tolerant algorithm for solving FT-DCOPs, which we term a (g, k)*-complete FT-DCOP algorithm.* This definition relies on two parameters: $g \in \mathbb{N}$, which represents the lower bound on the number of agents possessing knowledge of any utility function, and $k \in \mathbb{N}$, which represents the upper bound on the number of faulty agents within that set.

Definition 4.1. A (g, k)-complete FT-DCOP algorithm is an algorithm that guarantees an optimal solution to any FT-DCOP $(A, X, D, F, \alpha, \gamma)$ in the presence of a set $B \subset A$ of faulty agents, provided that for any utility function $f_i \in F$, the conditions $g \leq |\gamma(f_i)|$ and $k \geq |\gamma(f_i) \cap B|$ both hold.

This definition allows us to state the impossibility result formally, as presented in Theorem 4.2. We show its proof in Section A of the supplementary material[1].

THEOREM 4.2. *For any g and k such that $g \leq 2k$, no (g, k)-complete FT-DCOP algorithm exists.*

This theorem reveals a fundamental limitation: to guarantee optimality, the set of agents that know any utility function must have size at least $2k + 1$, so that a strict majority are correct. Accordingly, the proposed algorithm in the next section is developed under this necessary assumption.

[1]The supplementary material is available at https://mas.cs.tsukuba.ac.jp/~noshiro/.

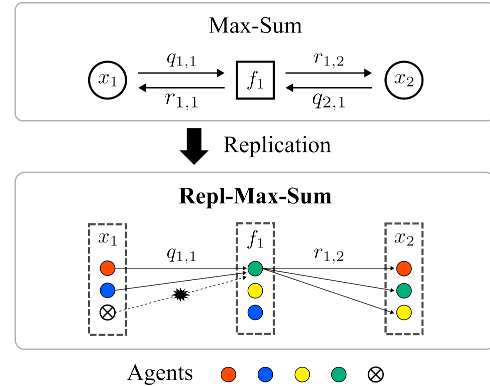

Figure 2: Overview of Repl-Max-Sum

5 REPLICATED-MAX-SUM

We propose a fault-tolerant DCOP algorithm, namely Replicated-Max-Sum (Repl-Max-Sum), for approximately solving FT-DCOPs. In this section, we introduce the key idea of the algorithm, describe its design, and present its properties and analysis.

5.1 Key Idea

Repl-Max-Sum aims to overcome the vulnerability of the Max-Sum algorithm to Byzantine faults. In the standard Max-Sum, an agent is assigned to each node in the factor graph. Under this design, even if faulty agents send erroneous messages, correct agents cannot detect these errors, because they do not observe the inputs of faulty agents and cannot rule out the possibility that the messages were computed correctly. Such undetected erroneous messages can disrupt the entire optimization process.

Conversely, Repl-Max-Sum replicates the computational processes of nodes across multiple agents. In this algorithm, correct agents, or *replicas*, send identical messages when they receive identical inputs, owing to the deterministic nature of Max-Sum. Then, receiving replicas can identify the correct messages by taking the majority, a standard mechanism in replication.

Figure 2 illustrates Repl-Max-Sum. Each node (i.e., x_1, x_2, and f_1) is replicated across three agents. For example, replicas of f_1 run on the green, yellow, and blue agents; the blue agent also runs a replica of x_1. Although a faulty replica of x_1 (depicted as a crossed circle) may send incorrect messages, receiving replicas identify and ignore them via majority-based verification, thereby sending correct subsequent messages.

In general, replication methods require complex procedures and stringent assumptions under the Byzantine fault models [3, 16, 22]. Specifically, correct replicas must reach consensus on the processing order of received messages to ensure identical state transitions. This is typically achieved by Byzantine consensus protocols, which involve intricate designs. Moreover, according to the impossibility results for Byzantine consensus [10, 12], the replica set size must exceed $3k$, where k bounds the number of faulty replicas.

Algorithm 1: Repl-Max-Sum (replica $a_i \in R(v)$ of node $v \in V$)

```
1  s_i ← 1; L_i ← ∅;
2  procedure When message M = (m, (v′, v), s)_j is received
3      store M in L_i;
4  procedure ReplMaxSum
5      initialize q and r messages to zero;
6      while termination condition is not satisfied do
7          update q (resp. r) messages by Eq. (2) (resp. Eq. (3));
8          let m_{v′} be q or r messages to send to v′ ∈ N(v);
9          send m_{v′} to all R(v′) for all v′ ∈ N(v);
10         wait until correct(m′_{v′}, v′, s_i) holds for all v′ ∈ N(v)
             and some m′_{v′};
11         store the received content m′_{v′} for all v′ ∈ N(v);
12         s_i ← s_i + 1;
13     return an optimal assignment if v ∈ X;
```

By contrast, Repl-Max-Sum mitigates these complexities and prerequisites by modifying the message-passing mechanism in Max-Sum. In Repl-Max-Sum, replicas send messages synchronously after receiving verified messages from all neighboring nodes, whereas the standard Max-Sum can proceed asynchronously [24]. This synchronicity enables replication without the need for consensus on message ordering via Byzantine consensus protocols. As a result, Repl-Max-Sum reduces complexity and relaxes the requirement so that the number of replicas need only exceed $2k$.

5.2 Algorithm Design

This subsection describes the detailed design of the replicas and the procedure of Repl-Max-Sum. Algorithm 1 presents the pseudocode.

5.2.1 Replicas. As previously mentioned, multiple agents are assigned as replicas to each node $v \in V$ in the factor graph. Let $R(v)$ denote the set of replicas of node v.

We make the following assumptions about where replicas are placed. For a variable node $x_i \in X$, replicas are assigned to the agents responsible for that variable (i.e., $R(x_i) = \alpha(x_i)$), whereas for a function node $f_j \in F$, replicas are assigned to the agents possessing knowledge of the function (i.e., $R(f_j) = \gamma(f_j)$).

Furthermore, we assume that the number of faulty agents within the replica set $R(v)$ for any node $v \in V$ is at most k. Accordingly, we configure replica sets so that $|R(v)| \geq 2k + 1$. This assumption is consistent with the condition $g \geq 2k + 1$, which is necessary to avoid the impossibility result established in Theorem 4.2.

5.2.2 Algorithm Procedure. The fundamental procedure of Repl-Max-Sum follows that of the standard Max-Sum. Replicas generate messages as described in Section 3.3 and multicast them to the replicas of neighboring nodes.

Message Format. To enable majority-based verification under the replication scheme, all correct replicas of a node must send identical messages. For that purpose, replicas attach additional information to each message. A message in Repl-Max-Sum has the form $(m, (v, v'), s)_i$, where m is the message content, v is the source node, v' is the destination node, s is a sequence number, and i indexes the sending replica $a_i \in R(v)$. Each replica a_i maintains a sequence number s_i, initialized to one and incremented by one in each iteration.

Message Sending. Replicas operate analogously to the corresponding nodes in the standard Max-Sum. They compute messages using Eqs. (2) and (3). When a replica $a_i \in R(v)$ computes a message content m (i.e., either q or r message) for a neighboring node $v' \in N(v)$, it multicasts $(m, (v, v'), s_i)_i$ to all replicas in $R(v')$.

Message Reception. Upon receiving a message, a replica a_i stores it in its message log L_i. Subsequently, the replica verifies the correctness of the message by comparing it with other messages in the log. To formalize this verification, we define the predicate $correct(m, v', s)$ as follows:

$$correct\left(m, v', s\right) \equiv \left|\left\{\left(m, \left(v', v\right), s\right)_j \in L_i \mid a_j \in R\left(v'\right)\right\}\right| \geq k + 1.$$

Replica a_i waits until, for each neighboring node $v' \in N(v)$, there exists a message content $m_{v'}$ such that $correct(m_{v'}, v', s_i)$ holds. When this condition is met, the replica increments its sequence number s_i, computes its outgoing messages based on these verified contents, and multicasts them. This verification and synchronization step is essential to ensure the correctness of Repl-Max-Sum; we present that guarantee in the next subsection.

5.3 Properties and Analysis

In this subsection, we present the properties and analysis of Repl-Max-Sum. First, we explain how the algorithm guarantees that correct replicas exchange proper messages while preventing interference from faulty replicas. Then, we analyze the communication complexity of Repl-Max-Sum.

5.3.1 Correctness Guarantee. The correctness of Repl-Max-Sum is established by the following proposition. It ensures that the correct replicas of any node send identical messages for a given sequence number, and therefore, they execute the Max-Sum procedure properly without being disrupted by faulty replicas.

Proposition 5.1. *Given any node $v \in V$, any neighboring node $v' \in N(v)$, and any sequence number s, if the algorithm has not terminated at s, then every correct replica $a_i \in R(v) \cap H$ sends exactly one identical message $M_i = (m_{v'}, (v, v'), s)_i$ to each replica in $R(v')$, where $m_{v'}$ is the message content for node v' computed according to Eq.* (2) *or* (3).

Proof. We prove the proposition by induction.

Base Case (for $s = 1$). All correct replicas initialize the q and r messages to zero (Line 5). By Eqs. (2) and (3), they deterministically compute a message content $m_{v'}$ for node v' (Lines 7 and 8). Thus, each replica a_i sends an identical message $M_i = (m_{v'}, (v, v'), 1)_i$. After waiting for messages from neighboring nodes, the replica increments its sequence number (Line 12), which ensures it sends only M_i for the previous sequence number, $s = 1$.

Induction Step. As the induction hypothesis, suppose that all correct replicas $a_j \in R(v'') \cap H$ of any neighboring node $v'' \in N(v)$ send messages $M''_j = (m''_{v''}, (v'', v), s)_j$ to replicas in $R(v)$. Since the number of faulty replicas in $R(v'')$ is at most k and $|R(v'')| \geq 2k+1$, there are at least $k+1$ correct replicas in $R(v'')$. Hence, each replica

$a_i \in R(v)$ eventually receives the messages M''_j and satisfies the predicate $correct(m''_{v''}, v'', s)$ (Line 10). Conversely, by the induction hypothesis, any messages with sequence number s and content different from $m''_{v''}$ can be sent by at most k faulty replicas, and thus cannot satisfy the *correct* predicate.

Therefore, each replica a_i stores the message content $m''_{v''}$ for every node $v'' \in N(v)$ (Line 11). Subsequently, the replica increments the sequence number (Line 12), computes the next message content $m_{v'}$ (Lines 7 and 8), and sends the identical message $M_i = (m_{v'}, (v, v'), s + 1)_i$ to replicas in $R(v')$ (Line 9). □

5.3.2 Communication Complexity. We next analyze the communication complexity of Repl-Max-Sum. In the minimum configuration, Repl-Max-Sum places $2k + 1$ replicas for each node in the factor graph. Because replicas multicast to all replicas of neighboring nodes, in each iteration, they transmit $O(k^2)$ messages per original message defined in the standard Max-Sum. Therefore, letting ℓ denote the maximum degree of the factor graph, the total number of message transmissions per iteration is $O(k^2\, \ell\, |V|)$.

Note that Repl-Max-Sum is practically more efficient than adopting general replication methods for Max-Sum. First, Repl-Max-Sum requires only $2k + 1$ replicas per node, whereas general replication methods require $3k + 1$. Furthermore, unlike those methods, Repl-Max-Sum does not need Byzantine consensus protocols for message ordering, which typically incur a heavy coordination process such as the three-phase commit [3].

6 EXPERIMENTAL EVALUATION

This section experimentally compares the performance of the standard Max-Sum algorithm and the proposed Repl-Max-Sum algorithm in the presence of Byzantine faulty agents[2]. First, we show that, when faults are present, Max-Sum suffers substantial degradation in solution quality, and its convergence is impeded by incorrect messages, potentially increasing runtime. Next, we show that Repl-Max-Sum masks the effects of faults via replication and consistently achieves high-quality approximate solutions whose behavior is identical to that of Max-Sum in the fault-free case.

6.1 Experimental Setup

We evaluate the algorithms on two problem domains: graph coloring problems (GCPs), a standard DCOP benchmark, and truck appointment scheduling (TAS), a realistic scenario described in Section 4.1.

In the DCOP formulation of both problems, we define utility functions based on their domain-specific constraints. Let X_i be a subset of variables, $C_i(X_i)$ a constraint, and $\varepsilon_i(X_i)$ a function returning a small value for tie-breaking. A utility function f_i is defined as follows:

$$f_i(X_i) = \begin{cases} 1 + \varepsilon_i(X_i) & \text{if } C_i(X_i) \text{ holds,} \\ \varepsilon_i(X_i) & \text{otherwise.} \end{cases} \quad (4)$$

This formulation assigns higher utilities to variable assignments that satisfy more constraints.

[2] The source code is available at https://github.com/nossie0360/ft-dcop-aamas2026.

6.1.1 Graph Coloring Problems. A graph coloring problem assigns a color to each node in a graph such that adjacent nodes do not share the same color. In our DCOP model, variables represent colors assigned to nodes, and a binary utility function is defined for each edge. Specifically, the scope of a utility function f_i is $X_i = \{x_j, x_{j'}\}$, where x_j and $x_{j'}$ denote the variables at the edge's endpoints, and the constraint prevents those two nodes from sharing the same color, i.e., $C_i(x_j, x_{j'}) \equiv x_j \neq x_{j'}$. Therefore, the optimal DCOP solution yields a coloring with the minimum number of color conflicts.

For data generation, we used $n \in \{12, 24, 36, 48\}$ nodes and, for each n, generated 50 random graphs with an average degree of 3 that are 3-colorable. The number of available colors (i.e., the domain size of each variable) was fixed at 3. Since all graphs are 3-colorable, the number of violations in the optimal solution is 0.

6.1.2 Truck Appointment Scheduling. Truck appointment scheduling plans the visit time (time slot) of each truck to the terminals, considering truck availability and terminal capacity. Each truck has two delivery tasks, each visiting a single terminal. Each terminal has T time slots, and at most one task can be assigned to a slot. Moreover, each task has a set of available time slots, and the two tasks belonging to the same truck cannot be scheduled at the same time. The goal is to find a schedule that satisfies these constraints.

In the DCOP formulation, each variable x_i represents the time slot of task i. Based on the generic utility function in Eq. (4), we encode three types of constraints as utilities:

- *Terminal capacity constraints.* For the set X_i of variables corresponding to tasks associated with the same terminal, all variable assignments must be pairwise distinct.
- *Intra-truck collision constraints.* For a pair of tasks x_j and $x_{j'}$ belonging to the same truck, $x_j \neq x_{j'}$ must hold.
- *Per-task availability constraints.* For a single variable x_i, x_i must be within the set of available time slots for task i.

For generating problem instances, we set the number of trucks to $n \in \{12, 24, 36, 48\}$, the number of terminals to $n/2$, and the number of time slots per terminal to $T = 8$. Thus, the total number of tasks is $2n$ and the total number of time slots is $(n/2) \cdot 8 = 4n$, keeping the ratio of tasks to time slots fixed at 0.5. For each task, we first chose the number of available time slots uniformly at random from $\{2, 3, 4, 5, 6\}$ and then sampled that number of time slots uniformly at random to form the allowable set. For each n, we randomly generated 50 feasible instances.

6.1.3 Algorithms and Fault Model. We set the number of agents executing the algorithms to n (equal to the number of graph nodes for GCPs and the number of trucks for TAS). Each agent is assigned to at least one node in the factor graph and executes the algorithmic procedures of its assigned nodes in parallel. Specifically, for Max-Sum, we assign one agent to each node. In contrast, for Repl-Max-Sum, we partition the agents into $n/3$ sets of three and assign one set to each node as its replica set. Each replica set tolerates at most one Byzantine fault ($k = 1$). Under this bound, we varied the total number of faulty agents, b, up to $n/3$ relative to the problem size n.

A Byzantine faulty agent generates each value of q and r messages uniformly at random from the range $[0, 1)$. To avoid trivial fault detection by correct agents, faulty agents then normalize q

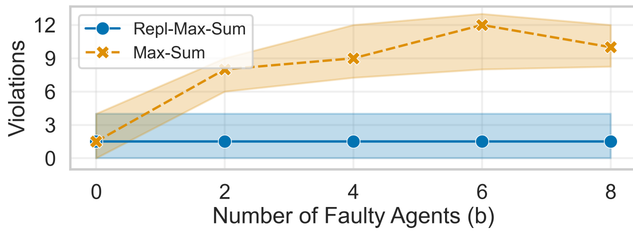

Figure 3: Number of Constraint Violations on GCPs ($n = 24$)

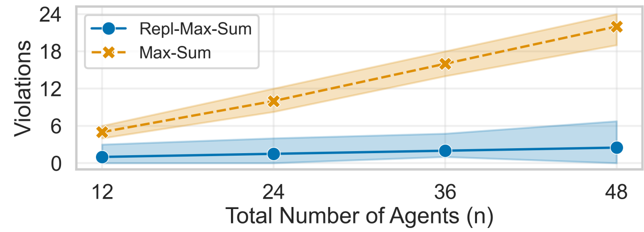

Figure 4: Number of Constraint Violations on GCPs ($b = n/3$)

values as in the Max-Sum procedure. This constitutes a stringent setting that models hard-to-detect faults.

Furthermore, to reduce redundant messages and enable a fair comparison, we also employed synchronous message passing for Max-Sum, identical to Repl-Max-Sum: in each iteration, an agent sends messages only after it has received messages from all neighbors. The termination condition uses an iteration limit of 1000 and terminates early if the updates of all q and r values fall below a prescribed threshold. For GCPs, a classical Max-Sum implementation performed poorly; therefore, we introduced damping with a damping factor of 0.9 [5]. For TAS, sufficient performance was obtained without damping.

6.1.4 Evaluation Metrics. We evaluate the algorithms in terms of solution quality, iterations to convergence, and communication load. We measure solution quality by the total number of constraint violations under the obtained assignment. For both problem domains, the optimal solution has zero violations. Iterations to convergence are measured as the number of iterations until all correct agents terminate. Communication load is measured as the total number of exchanged messages per iteration, thereby quantifying the overhead due to replication and multicast in Repl-Max-Sum.

We report medians and interquartile ranges (IQRs), which are more robust than means and standard deviations. We adopt these statistics because the randomly generated problem instances have highly heterogeneous characteristics, which can induce large standard deviations and mask true performance gaps.

6.2 Results on Graph Coloring Problems

We systematically varied the number of faulty agents (b) and the number of agents (n) and compared solution quality, iterations to convergence, and communication load. All statistics were computed over 50 instances for each condition.

6.2.1 Solution Quality. Figure 3 presents the number of constraint violations of the algorithms while fixing $n = 24$ and varying $b \in \{0, 2, 4, 6, 8\}$. Max-Sum without faults ($b = 0$) achieved a median violation count of 1.5 and obtained optimal solutions (zero violations) on 40% of the instances. In contrast, when faults are present ($b > 0$), the violation counts clearly increase. Notably, even with a small number of faults ($b = 2$, i.e., fewer than 10% of agents), at least 75% of the instances exhibited 6 or more violations, indicating a pronounced deterioration in solution quality. Conversely, Repl-Max-Sum showed the same violation distribution as Max-Sum with $b = 0$, regardless of b, demonstrating that replication effectively masks the impact of faults.

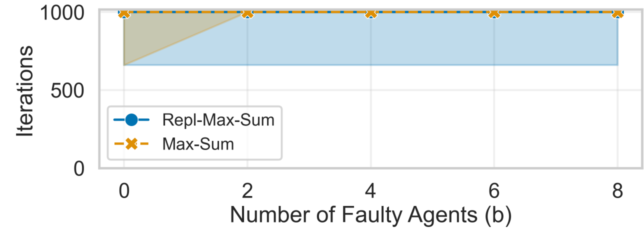

Figure 5: Terminated Iterations on GCPs ($n = 24$)

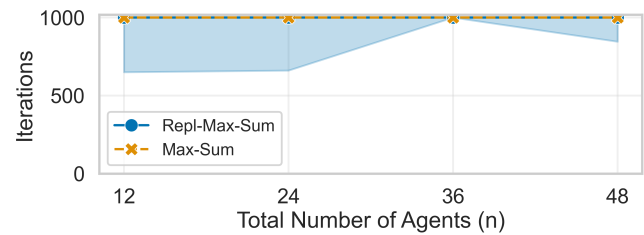

Figure 6: Terminated Iterations on GCPs ($b = n/3$)

Next, Figure 4 shows the number of constraint violations when varying $n \in \{12, 24, 36, 48\}$ and fixing the fault scale at $b = n/3$. In this figure, Repl-Max-Sum achieves lower violation counts than Max-Sum for all n. Although both algorithms show increasing violations with growing n, the growth rate is markedly higher for Max-Sum. For example, in medians from $n = 12$ to $n = 48$, Repl-Max-Sum increases by about 2.5 times, whereas Max-Sum increases by more than 4 times. These results indicate that Max-Sum, without any fault-tolerant mechanism, becomes increasingly sensitive to faults as the problem size grows, causing significant degradation in solution quality.

We also compared the violation counts of the algorithms on each problem instance. The results show that in over 80% of the instances with $n = 24$ and $b \in \{2, 4, 6, 8\}$, Repl-Max-Sum obtained better solutions than Max-Sum. We discuss this comparison in detail in Section B of the supplementary material.

6.2.2 Iterations to Convergence. Under synchronous message passing, we measured the number of iterations until all correct agents terminated, as shown in Figures 5 and 6. Repl-Max-Sum (and Max-Sum with $b = 0$) converged on some instances and stopped before reaching the iteration limit of 1000. In contrast, Max-Sum with faults ($b > 0$) did not converge under any tested condition and

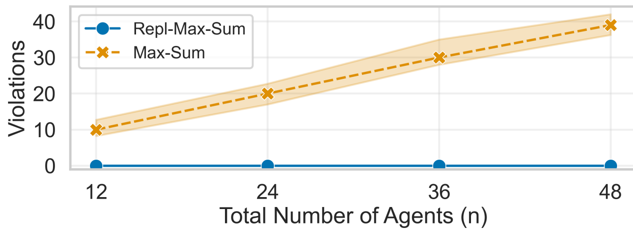

Figure 7: Number of Constraint Violations on TAS

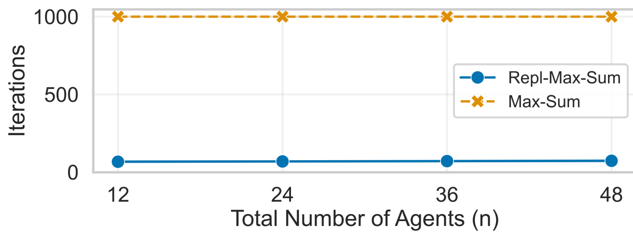

Figure 8: Terminated Iterations on TAS

always reached the iteration limit. Random messages from faulty agents destabilize the updates of q and r messages in Max-Sum, effectively preventing convergence. Conversely, Repl-Max-Sum blocks such adverse effects through replication and suppresses the impact of faults on convergence.

6.2.3 Communication Load. We compared the number of messages per iteration to assess communication overhead. For the representative condition $n = 24$ and $b = 0$, Max-Sum averaged 144.6 messages per iteration, while Repl-Max-Sum averaged 1301.5, i.e., approximately ninefold. This aligns with the algorithm design and experimental setting, where each node in a factor graph is replicated threefold and each replica multicasts to the three replicas of its neighbor, yielding a theoretical scaling factor of $3 \cdot 3 = 9$.

6.3 Results on Truck Appointment Scheduling

We compare the algorithms in TAS, a realistic scenario, in the presence of Byzantine faults. In this scenario, we vary the number of agents, $n \in \{12, 24, 36, 48\}$, and fix the fault scale at $b = n/3$.

6.3.1 Solution Quality. The number of constraint violations are presented in Figure 7. Across $n \in \{12, 24, 36, 48\}$, Repl-Max-Sum achieves the optimal solution on almost all instances. Specifically, only one instance for $n = 36$ and three instances for $n = 48$ had 1 or 2 violations. By contrast, Max-Sum exhibits violations in all instances, with magnitudes roughly proportional to n. For $n = 48$, the median violation count reaches 39, representing severe degradation compared to Repl-Max-Sum. In TAS terms, these violations correspond to infeasible time-slot assignments for tasks or schedules that concentrate excessive arrivals at terminals, undermining operational reliability. Therefore, deploying Max-Sum directly in real systems can carry high risk, highlighting the importance of a fault-tolerant method such as Repl-Max-Sum.

6.3.2 Iterations to Convergence. Figure 8 shows the terminated iterations of the algorithms. Repl-Max-Sum terminated within 100 iterations in more than 90% of the instances for all n. Conversely, Max-Sum did not converge and reached the iteration limit of 1000 in all instances. Thus, random messages generated by faulty agents destabilize the computation in Max-Sum, significantly increasing the number of iterations to termination. As in GCPs, Repl-Max-Sum can block these effects via replication and therefore provide fault tolerance with respect to both solution quality and convergence.

6.4 Discussion

We compared the behavior of Max-Sum and Repl-Max-Sum with Byzantine faults present on two DCOP domains with distinct characteristics, GCPs and TAS. The results are consistent across domains: when faults are present, Max-Sum suffers substantial degradation in solution quality and fails to converge within the iteration limit, whereas Repl-Max-Sum effectively masks the impact of faults, reproducing the solutions and iteration counts of fault-free Max-Sum.

Additionally, we observe a communication cost of Repl-Max-Sum: it increases messages per iteration by roughly ninefold due to three replicas per node and replica-to-replica multicast. Repl-Max-Sum messages are, however, small (in our implementation for GCPs, below 256 bytes each), and send/receive operations across replicas can be processed with high parallelism. Preliminary measurements on our testbed with two physical servers (see Section B of the supplementary material for details) showed that even when the number of concurrent messages increased ninefold, the total runtime increased by less than a factor of two. These results imply that the practical runtime overhead of Repl-Max-Sum can scale more favorably than the increase in message count.

Overall, while Repl-Max-Sum incurs a higher communication load, its improvements in solution quality and convergence dominate. This yields an acceptable trade-off, particularly in realistic systems such as TAS, where the risk of deploying Max-Sum without fault tolerance is substantial.

7 CONCLUSION

This paper addressed Byzantine fault tolerance in DCOPs. We first introduced a novel DCOP class, FT-DCOP, to handle the presence of Byzantine faults. We also established an impossibility result for solving FT-DCOPs, implying a strict majority of agents with knowledge of each utility function must be correct. Building on this, we proposed Repl-Max-Sum, which integrates replication into Max-Sum while avoiding Byzantine consensus and relaxing assumptions. Experiments on graph coloring and truck appointment scheduling showed that, with Byzantine faults present, the standard Max-Sum degrades markedly in solution quality and fails to converge within iteration limits, whereas Repl-Max-Sum masks faults and matches the fault-free performance with manageable communication overhead. Future work includes more extensive experiments on real systems and further development of algorithms robust to settings where faulty agents can directly manipulate variable assignments.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number JP25KJ0655.

REFERENCES

[1] Ahmed Mohssen Abdelmagid, Mohamed Samir Gheith, and Amr Bahgat Eltawil. 2022. A Comprehensive Review of the Truck Appointment Scheduling Models and Directions for Future Research. *Transport Reviews* 42, 1 (Jan. 2022), 102–126. https://doi.org/10.1080/01441647.2021.1955034

[2] Graham Billiau, Chee Fon Chang, and Aditya Ghose. 2012. SBDO: A New Robust Approach to Dynamic Distributed Constraint Optimisation. In *Principles and Practice of Multi-Agent Systems (Lecture Notes in Computer Science)*, Nirmit Desai, Alan Liu, and Michael Winikoff (Eds.). Springer, Berlin, Heidelberg, 11–26. https://doi.org/10.1007/978-3-642-25920-3_2

[3] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, USA, 173–186.

[4] Zhuoqing Chang, Shubo Liu, Xingxing Xiong, Zhaohui Cai, and Guoqing Tu. 2021. A Survey of Recent Advances in Edge-Computing-Powered Artificial Intelligence of Things. *IEEE Internet of Things Journal* 8, 18 (Sept. 2021), 13849–13875. https://doi.org/10.1109/JIOT.2021.3088875

[5] Liel Cohen, Rotem Galiki, and Roie Zivan. 2020. Governing Convergence of Max-sum on DCOPs through Damping and Splitting. *Artificial Intelligence* 279 (Feb. 2020), 103212. https://doi.org/10.1016/j.artint.2019.103212

[6] A. Farinelli, A. Rogers, A. Petcu, and N. R. Jennings. 2008. Decentralised Coordination of Low-Power Embedded Devices Using the Max-Sum Algorithm. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '08, Vol. 2)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 639–646.

[7] Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. 2018. Distributed Constraint Optimization Problems and Applications: A Survey. *Journal of Artificial Intelligence Research* 61 (March 2018), 623–698. https://doi.org/10.1613/jair.5565

[8] Linghe Kong, Jinlin Tan, Junqin Huang, Guihai Chen, Shuaitian Wang, Xi Jin, Peng Zeng, Muhammad Khan, and Sajal K. Das. 2022. Edge-Computing-Driven Internet of Things: A Survey. *ACM Comput. Surv.* 55, 8 (Dec. 2022), 174:1–174:41. https://doi.org/10.1145/3555308

[9] Xiangjie Kong, Yuhan Wu, Hui Wang, and Feng Xia. 2022. Edge Computing for Internet of Everything: A Survey. *IEEE Internet of Things Journal* 9, 23 (Feb. 2022), 23472–23485. https://doi.org/10.1109/JIOT.2022.3200431

[10] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

[11] Robert N. Lass, Evan A. Sultanik, Rachel Greenstadt, and William C. Regli. 2009. Robust Distributed Constraint Reasoning. *Distributed Constraint Reasoning* 75 (2009).

[12] Nancy A Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann, Oxford, England.

[13] Rajiv T. Maheswaran, Jonathan P. Pearce, and Milind Tambe. 2004. Distributed Algorithms for DCOP: A Graphical-Game-Based Approach. In *Proceedings of the Isca 17th International Conference on Parallel and Distributed Computing Systems*, David A. Bader and Ashfaq A. Khokhar (Eds.). ISCA, the Canterbury Hotel, San Francisco, California, Usa, 432–439.

[14] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. 2004. Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Multi-Event Scheduling. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '04, Vol. 1)*. IEEE Computer Society, USA, 310–317.

[15] Toshihiro Matsui, Hirohsi Matsuo, Marius Călin Silaghi, Katsutoshi Hirayama, Makoto Yokoo, and Satomi Baba. 2010. A Quantified Distributed Constraint Optimization Problem. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, Vol. 1. Citeseer, 1023–1030.

[16] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399

[17] Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. 2005. Adopt: Asynchronous Distributed Constraint Optimization with Quality Guarantees. *Artificial Intelligence* 161, 1-2 (Jan. 2005), 149–180. https://doi.org/10.1016/j.artint.2004.09.003

[18] Adrian Petcu and Boi Faltings. 2005. S-DPOP: Superstabilizing, Fault-Containing Multiagent Combinatorial Optimization. In *Proceedings of the National Conference on Artificial Intelligence*. 449–454.

[19] Adrian Petcu and Boi Faltings. 2005. A Scalable Method for Multiagent Constraint Optimization. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, Leslie Pack Kaelbling and Alessandro Saffiotti (Eds.). Professional Book Center, Edinburgh, Scotland, UK, 266–271.

[20] Mai-Ha Phan and Kap Hwan Kim. 2016. Collaborative Truck Scheduling and Appointments for Trucking Companies and Container Terminals. *Transportation Research Part B: Methodological* 86 (April 2016), 37–50. https://doi.org/10.1016/j.trb.2016.01.006

[21] Pierre Rust, Gauthier Picard, and Fano Ramparany. 2022. Resilient Distributed Constraint Reasoning to Autonomously Configure and Adapt IoT Environments. *ACM Transactions on Internet Technology* 22, 4 (Nov. 2022), 100:1–100:31. https://doi.org/10.1145/3507907

[22] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM, Toronto ON Canada, 347–356. https://doi.org/10.1145/3293611.3331591

[23] Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. 2005. Distributed Stochastic Search and Distributed Breakout: Properties, Comparison and Applications to Constraint Optimization Problems in Sensor Networks. *Artificial Intelligence* 161, 1-2 (2005), 55–87.

[24] Roie Zivan, Ben Rachmut, Omer Perry, and William Yeoh. 2023. Effect of Asynchronous Execution and Imperfect Communication on Max-Sum Belief Propagation. *Autonomous Agents and Multi-Agent Systems* 37, 2 (Sept. 2023), 40. https://doi.org/10.1007/s10458-023-09621-w