



HAL
open science

Adaptive Consumption by Continuous Negotiation

Ellie Beauprez, Anne-Cécile Caron, Maxime Morge, Jean-Christophe Routier

► **To cite this version:**

Ellie Beauprez, Anne-Cécile Caron, Maxime Morge, Jean-Christophe Routier. Adaptive Consumption by Continuous Negotiation. 21th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS), PAAMS, Jul 2023, Guimarães, Portugal. pp.28-39, 10.1007/978-3-031-37616-0_3 . hal-04163732

HAL Id: hal-04163732

<https://hal.science/hal-04163732>

Submitted on 17 Jul 2023


HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Adaptive consumption by continuous negotiation

Ellie Beauprez, Anne-Cécile Caron^[0000-0001-6672-5686], Maxime Morge ^[0000-0003-2139-7150], and Jean-Christophe Routier^[0000-0001-8032-6323]

Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRISTAL, F-59000 Lille, France
maxime.morge@univ-lille.fr

Abstract. In this paper, we study the problem of allocating concurrent jobs composed of situated tasks, underlying the distributed deployment of the MapReduce design pattern on a cluster. In order to implement our multi-agent strategy which aims at minimising the mean flowtime of jobs, we propose a modular agent architecture that allows the concurrency of negotiation and consumption. Our experiments show that our reallocation strategy, when executed continuously during the consumption process: (1) improves the flowtime; (2) does not penalise the consumption; (3) is robust against execution hazards.

Keywords: Distributed problem solving, Agent cooperation and negotiation

1 Introduction

Data sciences exploit large datasets on which computations are performed in parallel by different nodes. These applications challenge distributed computing in terms of task allocation and load-balancing. This is the case for the practical application that we consider in this paper: the most common model for processing massive data on a cluster, i.e. the MapReduce design pattern [8]. The execution of jobs, that need be completed as soon as possible, consists of processing resources located on nodes. Since multiple resources are required to perform a task on a node, its execution may require the retrieval of resources available on other nodes, thereby incurring additional cost.

Many works adopt the multi-agent paradigm to address the problem of task reallocation and load-balancing [9]. The individual-centred approach allows the distribution of heuristics for problems that are impractical due to the combinatorial scheduling, thus allowing for scaling. Moreover, multi-agent reallocation methods are inherently reactive and adapt to imprecise estimates of runtimes and to perturbations (e.g. node slowdowns).

In [2], we have proposed a multi-agent reallocation strategy for a set of jobs to be executed as soon as possible. In order to minimise the flowtime, the agents, which are cooperative, negotiate to determine the next tasks to delegate or even swap. This strategy requires the distributed deployment of autonomous agents that consume tasks and continuously exchange some of them to balance the current allocation. In this paper, we formalize the task consumption and reallocation operations and we propose a modular agent architecture that allows for the concurrency of negotiation and consumption.

According to the principle of separation of concerns, a first component agent is dedicated to the consumption (i.e. the execution) of tasks, a second to the negotiations and a third one to the local coordination of these operations through the management of the task bundle. The difficulty lies in designing the behaviour of agents that do not share a global state of the system (e.g. allocation) but they have local and partial knowledge. Our experiments show that our reallocation strategy, when executed continuously during the consumption process, does not penalise consumption and it can improve the flowtime by up to 37%, even when the agents have an imperfect knowledge of the environment, such as execution hazards.

After a review of related work in Section 2, we recall in Section 3 the formalization of the problem of job allocation composed of situated tasks. Section 4 formalizes the consumption/reallocation operations. Then, we describe in Section 5 how the consumption and reallocation processes are intertwined. We detail our agent architecture in Section 6. Section 7 presents our experimental results. Section 8 summarises our contribution and presents our perspectives.

2 Related work

Many papers have addressed the problem of task reassignment. The individual-centred approach overcomes the limitations of centralised solutions: the impossibility of solving large-scale problems and the low responsiveness to changes [9]. In particular, the dynamic task allocation problems require to propose processes that continuously adapt to changes in the execution environment or the performance of the executors [10]. Most of these works are based on the consensus-based bundle algorithm [5] which is a multi-agent task assignment method that: (a) selects the tasks to be negotiated through an auction process; (b) determines the bids that win these auctions by resolving potential conflicts. In particular, our modular agent architecture is largely inspired by [1]. However, our agents do not aim at minimising the makespan but the flowtime. Furthermore, we prefer here a bilateral protocol which allows, through the choice of the interlocutor, to make targeted proposals and thus to reduce the computational and communication costs associated with the negotiation. Finally, the simulation of the execution environment allows us to control its perturbations.

Chen *et al.* consider dynamic task allocation problems where tasks are released at uncertain times [4]. They propose to continuously adjust the task allocation by combining the local rescheduling of agents with task reallocation between agents. Similarly, our multi-agent strategy relies on a consumption strategy to define the local task scheduling and on a negotiation strategy for the tasks to be reallocated. Contrary to [4], we assume that the set of jobs is initially known, but our agents may have imperfect knowledge of the execution environment.

Most of the work, that considers that perturbations in the execution environment vary the task costs, rely on operations research techniques such as sensitivity analysis to assess the robustness of optima to perturbations, incremental methods to repair the initial optimal allocation when costs change, or combinatorial optimisation to exploit the measures of degradation [12]. Similarly, our strategy measures the gap be-

tween expected and observed progress in order to modify the allocation. However, our individual-centred approach allows us to solve large-scale problems.

Creech *et al.* address the problem of resource allocation and task hierarchy in distributed multi-agent systems for dynamic environments [6]. They propose an algorithm that combines updating and prioritisation algorithms, as well as reinforcement learning techniques. Contrary to learning techniques, our solution requires no prior model of either the data or the environment, and no exploration phase as this would not be relevant for the practical applications we are concerned with. In fact, the volume of data makes preprocessing and exploration too expensive. Moreover, the variability of the data makes it quickly obsolete.

Our previous experiments have shown that the flowtime achieved by our strategy is better than that achieved with distributed constrained optimisation (DCOP) techniques and remains close to that obtained with a classical heuristic, with in all cases a significantly reduced rescheduling time [2]. In this paper, we show how to deploy this strategy in a continuous way during the consumption process.

3 Problem

This section recalls the formalisation introduced in [2] of the task allocation problem with concurrent jobs composed of situated tasks. A distributed system consists of a set of computing nodes. These tasks require transferable and non-consumable resources that are distributed among different resource nodes.

Definition 1. A *distributed system* is quadruple $\mathcal{D} = \langle \mathcal{P}, \mathcal{N}_r, \mathcal{E}, \mathcal{R} \rangle$ where:

- \mathcal{P} is a set of p computing nodes;
- \mathcal{N}_r is a set of r resource nodes;
- $\mathcal{E} : \mathcal{P} \times \mathcal{N}_r \rightarrow \{\top, \perp\}$ is a neighborhood property that evaluates whether a computing node of \mathcal{P} is local to a resource node in \mathcal{N}_r ;
- $\mathcal{R} = \{\rho_1, \dots, \rho_k\}$ is a set of resources of size $|\rho_i|$. The location of resources, which are eventually replicated, is determined by the function $l : \mathcal{R} \rightarrow 2^{\mathcal{N}_r}$.

A resource can be local or remote to a computing node, depending on whether it is present on a resource node in the vicinity of the node. From this, we define the locality predicate: $\forall v_c \in \mathcal{P}, \forall \rho \in \mathcal{R}, \text{local}(\rho, v_c)$ iff $\exists v_r \in l(\rho)$ s.t. $\mathcal{E}(v_c, v_r)$. Resources are accessible to all computing nodes, including those on remote resource nodes.

A job is a set of independent, non-divisible and non-preemptible tasks. The execution of each task requires access to resources distributed on the nodes of the system. The execution of a job (without a deadline) consists of the execution all its tasks.

Definition 2. Let \mathcal{D} be a distributed system. We consider a set of ℓ **jobs** $\mathcal{J} = \{J_1, \dots, J_\ell\}$. Each job J_i , with the release time $t_{J_i}^0$, is a non-empty set of k_i **tasks** $J_i = \{\tau_1, \dots, \tau_{k_i}\}$.

We denote $\mathcal{T} = \cup_{1 \leq i \leq \ell} J_i$ the set of n tasks for \mathcal{J} and $\mathcal{R}_\tau \subseteq \mathcal{R}$ the set of resources required by the task τ . For the sake of brevity, we note $\text{job}(\tau)$ the job containing the task τ . We assume that that the number of jobs is negligible compared to the number of tasks, $|\mathcal{J}| \ll |\mathcal{T}|$.

The cost of a task for a node v_i is an estimate *a priori* of its runtime.

Definition 3. Let \mathcal{D} be a distributed system and \mathcal{T} be a set of tasks. The **cost function** $c : \mathcal{T} \times \mathcal{N} \mapsto \mathbb{R}_+^*$ is such that:

$$c(\tau, \mathbf{v}_j) = \sum_{\rho_j \in \mathcal{R}_\tau} c(\rho_j, \mathbf{v}_j) \text{ with } c(\rho_j, \mathbf{v}_i) = \begin{cases} |\rho_j| & \text{if local}(\rho_j, \mathbf{v}_i) \\ \kappa \times |\rho_j| & \text{with } \kappa > 1 \text{ otherwise.} \end{cases} \quad (1)$$

Since gathering remote resources is an additional cost, a task is more expensive if the resources required are "less local". The cost function can be extended to a set of tasks : $\forall \mathbf{T} \subseteq \mathcal{T}, c(\mathbf{T}, \mathbf{v}_i) = \sum_{\tau \in \mathbf{T}} c(\tau, \mathbf{v}_i)$.

Essentially, we consider the problem of allocating jobs consisting of situated tasks.

Definition 4. A **situated task allocation problem** is a quadruple $STAP = \langle \mathcal{D}, \mathcal{T}, \mathcal{J}, c \rangle$ where:

- \mathcal{D} is a distributed system of m computing nodes;
- $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$ is a set of n tasks;
- $\mathcal{J} = \{J_1, \dots, J_\ell\}$ is a partitioning of tasks in ℓ jobs;
- $c : \mathcal{T} \times \mathcal{N} \mapsto \mathbb{R}_+^*$ is the cost function.

A task allocation is an assignment of sorted bundles to different nodes.

Definition 5. An **allocation** for a STAP problem at time t is a vector of m sorted bundles $\vec{A}_t = ((B_{1,t}, \prec_1), \dots, (B_{m,t}, \prec_m))$ where each bundle $(B_{i,t}, \prec_i)$ is the set of tasks $(B_{i,t} \subseteq \mathcal{T})$ assigned to the node \mathbf{v}_i at time t , associated with a strict and total scheduling order $(\prec_i \subseteq \mathcal{T} \times \mathcal{T})$. $\tau_j \prec_i \tau_k$ means that if $\tau_j, \tau_k \in B_{i,t}$ then τ_j is executed before τ_k by \mathbf{v}_i . The allocation \vec{A}_t is such that:

$$\forall \tau \in \mathcal{T}, \exists \mathbf{v}_i \in \mathcal{N}, \tau \in B_{i,t} \quad (2)$$

$$\forall \mathbf{v}_i \in \mathcal{N}, \forall \mathbf{v}_j \in \mathcal{N} \setminus \{\mathbf{v}_i\}, B_{i,t} \cap B_{j,t} = \emptyset \quad (3)$$

All the tasks are assigned (Eq. 2) and each task is assigned to a single node (Eq. 3). For the sake of brevity, we denote $\vec{B}_{i,t} = (B_{i,t}, \prec_i)$, the sorted bundle of \mathbf{v}_i ; $\min_{\prec_i} \vec{B}_{i,t}$, the next task to be executed by \mathbf{v}_i ; and $\mathbf{v}(\tau, \vec{A}_t)$, the node whose bundle contains τ in \vec{A}_t .

In order to assess the quality of a job allocation, we consider the mean flowtime which measures the average time elapsed between the release date of the jobs and their completion date.

Definition 6. Let STAP be a problem and \vec{A}_t be an allocation at time t . We define :

- the waiting time of the task τ in the bundle $\vec{B}_{i,t}$,
 $\Delta(\tau, \mathbf{v}_i) = \sum_{\tau' \in B_{i,t} | \tau' \prec_i \tau} c(\tau', \mathbf{v}_i)$
- the completion time of the task $\tau \in \mathcal{T}$ for the allocation \vec{A}_t ,
 $C_\tau(\vec{A}_t) = \Delta(\tau, \mathbf{v}(\tau, \vec{A}_t)) + t - t_{job(\tau)}^0 + c(\tau, \mathbf{v}(\tau, \vec{A}_t))$
- the completion time of the job $J \in \mathcal{J}$ for \vec{A}_t ,
 $C_J(\vec{A}_t) = \max_{\tau \in J} \{C_\tau(\vec{A}_t)\}$
- the **mean flowtime** of \mathcal{J} for \vec{A}_t ,

$$C_{mean}(\vec{A}_t) = \frac{1}{\ell} C(\vec{A}_t) \text{ with } C(\vec{A}_t) = \sum_{J \in \mathcal{J}} C_J(\vec{A}_t) \quad (4)$$

The waiting time measures the time from the current time t until the task τ is executed.

4 Operations

Here we formalise the operations of task consumption and reallocation. A task consumption by a node consists in the latter removing this task from its bundle to execute it. The completion of a task is a disruptive event that changes not only the allocation of tasks but also the underlying problem.

Definition 7. Let $STAP = \langle \mathcal{D}, \mathcal{T}, \mathcal{J}, c \rangle$ be a problem and \vec{A}_t be an allocation. The **consumption at time t by the node v_i** , whose bundle is not empty ($B_{i,t} \neq \emptyset$), leads to the allocation $\vec{A}_t' = \lambda(v_i, \vec{B}_{i,t})$ for the problem $STAP' = \langle \mathcal{D}, \mathcal{T}', \mathcal{J}', c \rangle$ where:

$$\mathcal{T}' = \mathcal{T} \setminus \{\min_{\prec_i} B_{i,t}\} \quad (5)$$

$$\mathcal{J}' = \begin{cases} \mathcal{J} \setminus \{\text{job}(\min_{\prec_i} B_{i,t})\} & \text{if } \text{job}(\min_{\prec_i} B_{i,t}) = \{\min_{\prec_i} B_{i,t}\} \\ \mathcal{J} & \text{otherwise} \end{cases} \quad (6)$$

In the latter case:

$$\forall J_j \in \mathcal{J} \exists J'_j \in \mathcal{J}' \text{ s.t. } J'_j = \begin{cases} J_j \setminus \{\min_{\prec_i} B_{i,t}\} & \text{if } \text{job}(\min_{\prec_i} B_{i,t}) = J_j \\ J_j & \text{otherwise} \end{cases} \quad (7)$$

$$\text{and } \vec{A}_t' = (\vec{B}_{1,t}', \dots, \vec{B}_{m,t}') \text{ with } \vec{B}_{j,t}' = \begin{cases} \overrightarrow{B_{i,t} \ominus \min_{\prec_i} B_{i,t}} & \text{if } j = i \\ \vec{B}_{j,t} & \text{otherwise} \end{cases} \quad (8)$$

When a task is consumed, it is removed not only from the resulting problem in the task set but also from the corresponding job. The latter can also be removed if the task was the only (last) task in the job. The resulting allocation is also changed. The task is removed from the bundle it was in. The tasks are intended to be consumed one by one until the empty allocation is reached.

A task consumption causes the *flowtime* to decrease locally, at time t : $\sum_{J \in \mathcal{J}} C_J(\lambda(v_i, \vec{B}_{i,t})) < \sum_{J \in \mathcal{J}} C_J(\vec{B}_{i,t})$. This is not always the case over time since the effective costs of tasks may differ from the estimated costs. If a task turns out to be more expensive than expected when it is performed, the flowtime may increase after a task has been consumed, as in Ex. 1.

Example 1. Let $STAP = \langle \mathcal{D}, \mathcal{T}, \mathcal{J}, c \rangle$ be a problem with:

- $\mathcal{D} = \langle \mathcal{P}, \mathcal{N}_r, \mathcal{E}, \mathcal{R} \rangle$, a distributed system with a single computing node $\mathcal{P} = \{v_1\}$ associated with a single resource node $\mathcal{N}_r = \{v_1^r\}$, such that $\mathcal{E}(v_1, v_1^r) = \top$ and a single resource $\mathcal{R} = \{\rho_1\}$ over the resource node v_1^r ;
- two tasks $\mathcal{T} = \{\tau_1, \tau_2\}$;
- a single job $\mathcal{J} = \{J_1\}$ released at $t_{J_1}^0 = 0$ composed of the two tasks $J_1 = \{\tau_1, \tau_2\}$;
- the cost functions c such that $c(\tau_1, v_1) = 2$ and $c(\tau_2, v_1) = 4$.

The allocation $\vec{A}_0 = (\vec{B}_{1,t})$ with $\vec{B}_{1,t} = (\tau_1, \tau_2)$. According to Eq. 4, the flowtime is $C_{mean}(\vec{A}_0) = C_{J_1}(\vec{A}_0) = C_{\tau_2}(\vec{A}_0) = \Delta(\tau_2, \nu_1) + t + t_{J_1}^0 + c(\tau_2, \nu_1) = c(\tau_1, \vec{A}_t) + 0 + 0 + c(\tau_2, \nu_1) = 2 + 4 = 6$.

If the consumption of τ_1 ends at time $t_1 = 3$, it means that this task turns out to be more expensive than expected when running. Therefore, the flowtime of $\vec{A}_{t_1} = (\vec{B}_{\nu_1, t_1})$ with $B_{\nu_1, t_1} = (\tau_2)$ is $C_{mean}(\vec{A}_{t_1}) = C_{J_1}(\vec{A}_{t_1}) = t_1 + t_{J_1}^0 + c(\tau_2, \nu_1) = 3 + 0 + 4 = 7 > C_{mean}(\vec{A}_0)$.

We consider an operation where some tasks are moved from one bundle to another.

Definition 8. Let $\vec{A}_t = (\vec{B}_{1,t}, \dots, \vec{B}_{m,t})$ be an allocation of the problem $STAP = \langle \mathcal{D}, \mathcal{J}, \mathcal{J}, c \rangle$ at time t . The **bilateral reallocation** of the non-empty list of tasks T_1 assigned to the proposer ν_i in exchange for the list of tasks T_2 assigned to the responder ν_j in \vec{A}_t ($T_1 \subseteq B_{i,t}$ and $T_2 \subseteq B_{j,t}$) leads to the allocation $\gamma(T_1, T_2, \nu_i, \nu_j, \vec{A}_t)$ with m bundles s.t.:

$$\gamma(T_1, T_2, \nu_i, \nu_j, \vec{B}_{k,t}) = \begin{cases} \overline{B_{i,t} \ominus T_1 \oplus T_2} & \text{if } k = i, \\ \overline{B_{j,t} \ominus T_2 \oplus T_1} & \text{if } k = j, \\ \vec{B}_{k,t} & \text{otherwise} \end{cases} \quad (9)$$

If T_2 is empty, the reallocation is called a *delegation*. Otherwise, it is a *swap*.

We restrict ourselves here to bilateral reallocations, but multilateral reallocations deserve to be explored.

Contrary to most other works (e.g. [7]), our agents are not individually rational but they have a common goal that overrides their individual interests: to reduce the flowtime.

Definition 9. Let \vec{A}_t be an allocation at time t for the problem $STAP = \langle \mathcal{D}, \mathcal{J}, \mathcal{J}, c \rangle$. The bilateral reallocation $\gamma(T_1, T_2, \nu_i, \nu_j, \vec{A}_t)$ is **socially rational** iff the flowtime decreases, $C(\gamma(T_1, T_2, \nu_i, \nu_j, \vec{A}_t)) < C(\vec{A}_t)$.

An allocation is said to be **stable** if there is no socially rational reallocation. In [2], we have shown the termination of the process that iterates this type of reallocation.

5 Process

In order to carry out the consumption and reallocation processes simultaneously, we consider two types of agents: (a) node agents, each of which represents a computing node (cf Section 6); (b) the supervisor, which synchronises the phases of the negotiation process.

The consumption process consists of the concurrent or sequential execution of the different tasks by the computing nodes under the supervision of their agent. The reallocation process consists of multiple local reallocations that are the results of bilateral negotiations between node agents, performed sequentially or concurrently. These processes are complementary. While consumption is continuous, agents negotiate their task

bundles up to the point where a stable allocation is reached. A task consumption can make an allocation unstable and thus trigger new negotiations. The consumption process ends when all tasks have been executed. It is worth noting that this multi-agent system is inherently adaptive. Indeed, if a task turns out to be more expensive than expected, because the runtime was underestimated or the running node is slowed down, then the reallocation process, which runs continuously, allows the allocation to be balanced by taking into account the actual cost of the task.

The **consumption strategy** of node agents, detailed in [2], specifies the scheduling of tasks executed by the node for which they are responsible. In order to reduce the flowtime, this strategy executes the tasks of the least expensive jobs before those of the most expensive jobs.

The **negotiation strategy** of node agents, also detailed in [2], which is based on a peer model, in particular a belief base built from the messages exchanged, determines whether a reallocation is socially rational according to the agent's beliefs. The agents have: (a) an offer strategy which proposes bilateral reallocations; (b) an acceptance rule that evaluates whether a proposal is socially rational before accepting or rejecting it; and (c) a counteroffer strategy that selects a counterparty to a delegation to propose a task swap.

The negotiation process consists of two successive stages: (1) agents propose the delegations which they consider socially rational and which are accepted or rejected by their peers; (2) agents propose delegations which are not necessarily socially rational but which are likely to trigger counter-offers and thus socially rational swap. The stages of negotiation alternate successively in a way that is concurrent with consumption.

6 Architecture

For the design of a node agent, we adopt a modular architecture that allows concurrent negotiation and consumption. A node agent is a composite agent consisting of 3 component ones (cf Fig. 2a), each with a limited role: the **worker** executes (consumes) tasks; the **negotiator** updates a belief base for negotiating tasks with peers; the **manager** handles the task bundle of the computing node to schedule the worker by adding or deleting tasks according to the bilateral reallocations bargained by the negotiator.

In order to prioritise task consumption, as soon as the manager is informed that the worker is free, the manager gives to the worker the next task to run in accordance with the consumption strategy, even if this means cancelling the reallocation of this task during the negotiation. This task is then no longer eligible for a potential reallocation.

We represent here the interactions between the component agents with interaction diagrams where solid arrow heads represent synchronous calls, open arrow heads represent asynchronous messages, and dashed lines represent reply messages.

After the manager has given to the worker a task, the worker informs the manager when it has been completed (cf Fig. 1a). In order to give priority to the consumption over the negotiation, the query of the worker for the next task to run takes priority and preempts the manager's interactions with the negotiator. To refine its estimate of the waiting time for the tasks in its bundle, the manager can ask the worker for an estimate of the remaining runtime for the current task (cf Fig 1a).

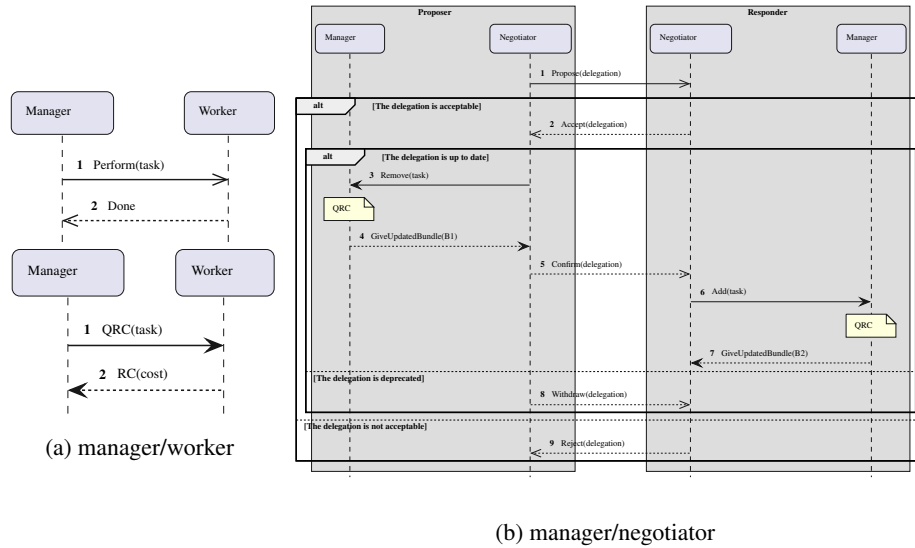


Fig. 1: Interactions between the manager, the worker and the negotiator

In a first negotiation stage, the agents negotiate delegations (cf Fig. 1b). To confirm such a bilateral reallocation, the proposer’s negotiator synchronously asks the manager to update the task bundle so that it can update its belief base before engaging in new negotiations. After this confirmation, the responder’s negotiator does the same. The QRC tag indicates that the manager interacts with the worker according to the protocol in Fig. 1a in order to take into account the remaining runtime for the current task. In a second negotiation stage, the agents bargain task swapping and the interactions are similar.

Despite our modular architecture, the main difficulty lies in the design of the behaviours of the agents, which are specified in [3] by automata¹, and whose complexity is measured in Tab. 2b with the number of states, transitions and lines of code.

The **worker** is either free or busy to run a task and it can therefore estimate the remaining runtime of the current task.

The **manager** handles the task bundle and coordinates the task consumptions of the worker with the reallocations bargained by the negotiator. When the latter informs the manager that there is no more socially rational delegations to propose and that the negotiator is waiting for proposals from peers, the manager informs the supervisor. The manager also continues to distribute tasks to the worker until the bundle is empty. Informed that no node agent detects a reallocation opportunity, the supervisor triggers the next negotiation stage. Finally, the supervisor completes the process when it learns from the managers that all tasks have been consumed.

The **negotiator** responds to the proposals of its peers and updates its belief base, which make it possible to detect reallocation opportunities. After proposing a delega-

¹ <https://gitlab.univ-lille.fr/maxime.morge/smastaplus/-/tree/worker/doc/specification>

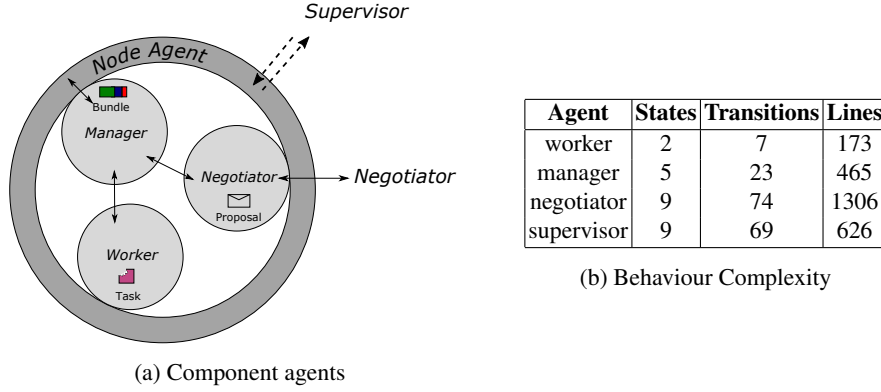


Fig. 2: Node Agent Architecture

tion, the negotiator waits for an acceptance, a rejection or a counter-proposal before a deadline. When the negotiator has accepted a proposal or made a counter-proposal, it waits for a confirmation or a withdrawal from its interlocutor (if the task has been consumed). When the negotiator has confirmed its acceptance of a counteroffer, it also waits for a double-confirmation. When the offer strategy does not suggest delegation, the belief base is updated until a new opportunity is found.

7 Experiments

Our experiments aim to validate that the strategy of reallocation, when applied continuously during the consumption: (1) improves the flowtime; (2) does not penalise the consumption; (3) is robust against execution hazards (i.e. node slowdowns). We present here our metrics, our experimental protocol and our results².

Instead of the expected execution time (Eq. 1), we consider here $c^S(\tau, v)$, the effective cost for the node v to perform the task:

- with a perfect knowledge of the computing environment, $c^{SE}(\tau, v_i) = c(\tau, v_i)$
- with the slowing down of half of the nodes,

$$c^{SH}(\tau, v_i) = \begin{cases} 2 \times c(\tau, v_i) & \text{if } i \bmod 2 = 1 \\ c(\tau, v_i) & \text{otherwise} \end{cases} \quad (10)$$

Therefore, we distinguish: the **simulated flowtime** $C_{mean}^S(\vec{A}_t)$ according to the effective costs; the **realised flowtime** $C_{mean}^R(\vec{A}_t)$ according to the task completion times which are measured. We define the **rate of performance improvement**: $\Gamma = \frac{C_{mean}^R(\vec{A}_0) - C_{mean}^R(\vec{A}_e)}{C_{mean}^R(\vec{A}_0)}$ where \vec{A}_e is the allocation when the tasks are performed and \vec{A}_0 is the initial allocation.

² The experiments are reproducible using the following instructions:
<https://gitlab.univ-lille.fr/maxime.morge/smastaplus/-/tree/master/doc/experiments>

The rate of performance improvement is positive if the realised flowtime obtained by the reallocation process is better (i.e. lower) than that of the initial allocation.

Our prototype [3] is implemented using the Scala programming language and the Akka library [11] which is suitable for highly concurrent, distributed, and resilient message-driven applications. Experiments have been conducted on a blade with 20 CPUs and 512Go RAM. The fact that, in our experiments, the difference between the realised flowtime and the simulated flowtime of the initial allocation ($C_{mean}^R(\vec{A}_0) - C_{mean}^S(\vec{A}_0)$), which measures the cost of the infrastructure, is negligible supports this technological choice.

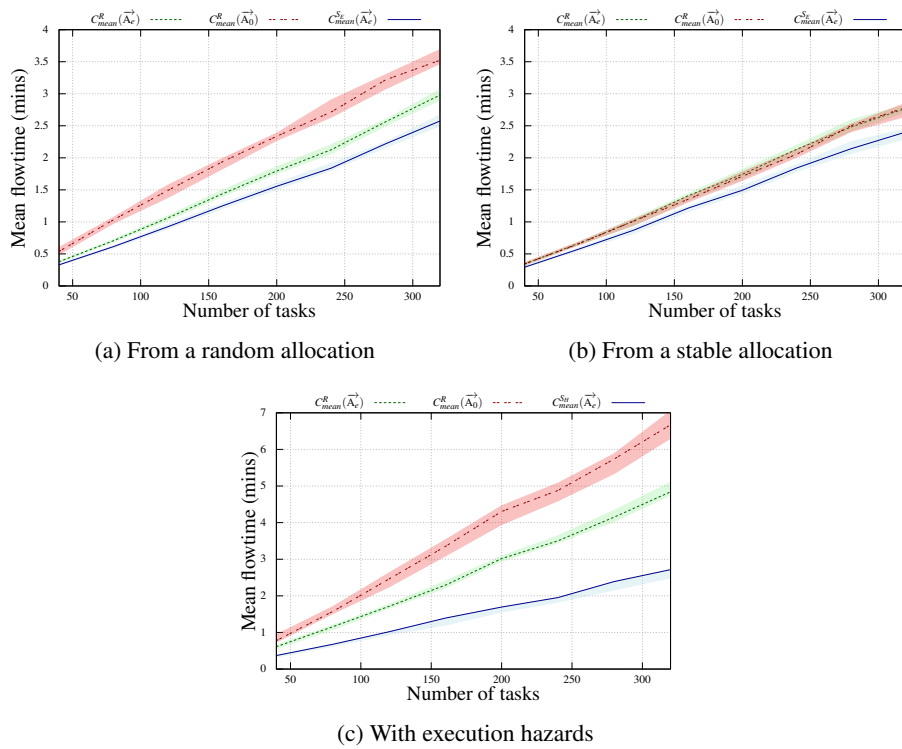


Fig. 3: The strategy of reallocation improves the flowtime

The experimental protocol consists of randomly generating 25 initial allocations for different *STAP* problems. We have empirically chosen $\kappa = 2$ as a realistic value to capture the overhead of fetching remote resources in a homogeneous network. We consider $m = 8$ nodes, $l = 4$ jobs, $n \in [40; 320]$ tasks with 10 resources per task. Each resource ρ_i is replicated 3 times and $|\rho_i| \in [0; 500]$. In order to avoid unnecessary negotiations due to the asynchronicity of the consumption operations, we assume in our experiments

that a bilateral reallocation is socially rational if it decreases the flowtime by at least one second.

Hypothesis 1: The reallocation strategy improves the flowtime. We assume here that the initial allocations are random and that agents have perfect knowledge of the environment (c^{SE}). Fig. 3a shows the medians and standard deviations of our metrics as functions of the number of tasks. We observe that the realised flowtime of the reallocation is better than the realised flowtime of the initial allocation and it is bounded by the simulated flowtime of the reallocation (if an oracle computes the reallocation in constant time). Our strategy improves the *flowtime* by continuously reallocating the remote tasks whose delegation reduces their costs during the consumption process. The rate of performance improvement (Γ) is between 20 % and 37 %.

Hypothesis 2: The reallocation strategy does not penalise the consumption. We assume here that the initial allocations are stable. In Fig. 3b, the realised flowtime of the reallocation is similar to the realised flowtime of the initial allocation. The negotiation overhead is negligible since no negotiation is triggered if the agents consider the allocation to be stable.

Hypothesis 3: The reallocation strategy is robust against execution hazards. We consider here the effective cost of the tasks, which simulates the slowing down of half of the nodes, c^{SH} . In Fig. 3c, the flowtimes have doubled due to execution hazards. Furthermore, the realised flowtime of the reallocation remains better than the realised flowtime of the initial allocation despite imperfect knowledge of the computing environment. Taking into account the effective runtimes of the tasks already executed, the rate of performance improvement (Γ) is between 30 % and 37 %.

8 Discussion

In order to design autonomous agents that simultaneously perform consumption and reallocation, we have proposed a modular agent architecture composed of three component agents: the worker which performs the tasks; the negotiator which bargains reallocations with peers; and the manager which locally coordinates these operations by managing the task bundle. Without knowing the global state of the system, i.e. the allocation, these agents have local knowledge (e.g. the current task, the task bundle) and beliefs that guide their behaviour.

Our experiments show that the rate of performance improvement due to our reallocation strategy, when used continuously during the consumption process, can reach 37 %. Furthermore, the negotiation overhead is negligible since it is suspended when the allocation is stable. Finally, even if some nodes are slowed down, our strategy of reallocation adapts to the execution context by distributing more tasks to the nodes that are not slowed down since it takes into account the effective runtime of the tasks already executed, without requiring a learning phase.

A sensitivity analysis of the influence of the replication factor, of the remote resource fetch overhead (κ) and of the negotiation timeout is beyond the scope of this article, but would deserve a thorough study. We would also like to evaluate the responsiveness of our strategy to the release of jobs over time.

More generally, our future work will focus on integrating task reallocation into a provisioning process that adds or removes computing nodes at runtime according to user needs in order to propose an elastic multi-agent strategy for scalability.

References

1. Baert, Q., Caron, A.C., Morge, M., Routier, J.C., Stathis, K.: An adaptive multi-agent system for task reallocation in a MapReduce job. *Journal of Parallel and Distributed Computing* **153**, 75–88 (2021)
2. Beauprez, E., Caron, A.C., Morge, M., Routier, J.C.: Task Bundle Delegation for Reducing the Flowtime. In: *Agents and Artificial Intelligence, 13th International Conference, ICAART 2021, Online streaming, February 4-6, 2021, Revised Selected Papers, Lecture Notes in Computer Science*, vol. 13251, pp. 22–45. Springer International Publishing (2022)
3. Beauprez, E., Morge, M.: Scala implementation of the Extended Multi-agents Situated Task Allocation. <https://gitlab.univ-lille.fr/maxime.morge/smastaplus> (2020)
4. Chen, Y., Mao, X., Hou, F., Wang, Q., Yang, S.: Combining re-allocating and re-scheduling for dynamic multi-robot task allocation. In: *Proc. of SMC*. pp. 395–400 (2016)
5. Choi, H.L., Brunet, L., How, J.P.: Consensus-based decentralized auctions for robust task allocation. *IEEE transactions on robotics* **25**(4), 912–926 (2009)
6. Creech, N., Pacheco, N.C., Miles, S.: Resource allocation in dynamic multiagent systems. *CoRR* **abs/2102.08317** (2021)
7. Damamme, A., Beynier, A., Chevaleyre, Y., Maudet, N.: The power of swap deals in distributed resource allocation. In: *Proc. of AAMAS*. pp. 625–633 (2015)
8. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: *Proc. of OSDI*. pp. 137–150 (2004)
9. Jiang, Y.: A survey of task allocation and load balancing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems* **27**(2), 585–599 (2016)
10. Lerman, K., Jones, C., Galstyan, A., Matarić, M.J.: Analysis of dynamic task allocation in multi-robot systems. *The International Journal of Robotics Research* **25**(3), 225–241 (2006)
11. Lightbend: Akka is the implementation of the actor model on the JVM. <http://akka.io> (2020)
12. Mayya, S., D’antonio, D.S., Saldaña, D., Kumar, V.: Resilient task allocation in heterogeneous multi-robot systems. *IEEE Robotics and Automation Letters* **6**(2), 1327–1334 (2021)