Distributed Multiagent Resource Allocation using Reservations to Improve Handling of Dynamic Task Arrivals

by

Graham Kendall Pinhey

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In the artificial intelligence subfield of multi-agent systems, there are many applications for algorithms which optimally allocate a set of resources among many available tasks which demand those resources. In this thesis we present a distributed algorithm to solve this problem which adapts well to dynamic task arrivals, where new work arises at short notice. This algorithm builds on prior work which focused on finding the optimal allocation in a closed environment with a fixed number of tasks. Our algorithm is designed to leverage preemption if it is available, revoking resource allocations to tasks in progress if new opportunities arise which those resources are better suited to handle. However, interrupting tasks in progress is rarely without cost, and our algorithm both respects these costs and may reserve resources to avoid unnecessary costs from hasty allocation.

Our multi-agent model assigns a task agent to each task which must be completed and a proxy agent to each resource which is available. These proxy agents are responsible for allocating the resource they manage, while task agents are responsible for learning about their environment and planning out which resources to request for their task. The distributed nature of our model makes it easy to dynamically introduce new tasks with associated task agents. Preemption occurs when a task agent approaches a proxy agent with a sufficiently compelling need that the proxy agent determines the newcomer derives more benefit from the proxy agent's resource than the task agent currently using that resource. We compare to other multi-agent resource allocation frameworks which permit preemption under more conservative assumptions, and show through simulation that our planning and learning techniques allow for improved allocations through more permissive preemption. Our simulations present a medical application which models fallible human resources, though the techniques used are applicable to other domains such as computer scheduling.

We then revisit the model with a focus on opportunity cost, introducing resource reservation as an alternative method to preemption for addressing expected future changes in the task allocation environment. Simulations help identify the scenarios where opportunity cost is a significant concern. The model is then further expanded to account for switching costs, where interrupting tasks in progress is worse than simply delaying tasks, and the logical extreme where resource allocation is irrevocable thus encouraging careful decisions about where to commit resources.

This thesis makes three primary contributions to multi-agent resource allocation. The first is an improved distributed resource allocation framework which uses Transfer-of-Control strategies and learning to rapidly find good allocations in a dynamic environment.

The second is a discussion of the importance of opportunity cost in resource allocation, accompanied by a simple "dummy agent" implementation which validates the use of resource reservation to address scenarios vulnerable to opportunity cost. Finally, the effectiveness of this resource allocation framework with reservation is extended to environments where preemption is costly or impossible.

Acknowledgements

I would like to thank the many people who helped me over the course of this degree.

To my supervisor Robin Cohen, I thank you for my topic as well as your continuous reviews, drawing this thesis forward from my first blank pages through reams of unhewn text to become the product it is today. This work would not exist without your advice and support, and while another supervisor might have drawn a similar text from me I am not confident it would have been as polished.

To my readers Pascal Poupart and Grant Weddell, I thank you for the time you spent reading and, ultimately, affirming this work. Your feedback and insights kept me honest and validated the time I spent exploring these ideas.

To my colleague John A. Doucette, I thank you for the work you left behind, the guidance you provided through my other attempts at publication, and the companionship you shared. It has been an honor and a pleasure to work with you.

To my other officemates and colleagues Cecylia Bocovich, John Champaign, Lachlan Dufton, Hadi Hosseini, Noel Sardana, Dean Shaft, and Alan Tsang, I thank you for making my stay in Waterloo a less lonely one.

To my immediate and nearby family, Scott Pinhey, Suzanne Pinhey, Theresa Pinhey, and Raymond Comeau, thank you for your support and understanding throughout both of my degrees. I become aware at times of the magnitude of the opportunities that you have afforded me, and I am humbled to accept them.

I acknowledge the generous financial support of the University of Waterloo and its Graduate Studies Office through the Presidents Scholarship and the David R. Cheriton Graduate Scholarship as well as teaching assistant and travel funding; the Government of Ontario through the Ontario Gradute Scholarship program; and the National Sciences and Engineering Research Council (NSERC) through the Alexander Graham Bell Canada Graduate Scholarship and the hSITE program. The stress I felt during this degree was never due to my financial situation.

To administrative support Wendy Rush, I thank you for shielding me from the brunt of academic paperwork. My incredible experiences from traveling to conferences were financed in no small part through your efforts.

To my other friends near and far, thank you for sharing entertainment with me over these years. We all need something to help keep us sane.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Artificial Intelligence (AI) is a broad field concerned with the application of algorithms to solve complex problems which often do not admit efficient worst-case solutions. Many real-world problems have practical constraints that limit the effectiveness of theoretically optimal algorithms, for example any problem where decisions need to be made before the implications are clear. If an algorithm does not have all the information about a problem before it starts, then choosing the best course of action may not always be possible, necessitating a decision-making procedure that adapts to information as it arrives in order to do the best it can. Resource allocation problems are a good example, where there are many tasks which need to be completed with the assistance of a small set of resources. Deciding which resources are allocated to which tasks and for how long is a difficult problem with a large search space, and in real life there is often uncertainty about exactly when new tasks will arise.

1.1 Definitions

This thesis focuses on the use of a multi-agent model to address resource allocation problems. In order to describe the specific aims of this thesis, several central concepts in AI must be defined.

1.1.1 Multi-agent Systems

Agents in AI are software entities endowed with some degree of autonomy, usually each running a local algorithm that may interact with the external environment and other agents [20]. These agents act on behalf of the interests of users, performing automated reasoning to make good decisions with respect to those interests.

Multi-agent systems are useful for solving some problems addressed by AI [8]. The emergent behaviour from having multiple entities (agents) acting according to local information can solve interesting problems in an intuitive way; a simple real-world example arises when distributed robots act individually towards a common goal. In a software environment, individual agents focused on potentially-competing goals can interact with each other to form compromises that improve the welfare of the system as a whole. Designing agent interactions that generate useful results is the challenge. Just as greedy algorithms often provide a cheap reasonable solution to problems, making the correct local improvements can provide good practical solutions.

In this thesis, agents represent both the tasks that enter a system as well as the resources present in the system, and these agents communicate in order to decide on an optimal allocation, as measured using a utility function.

1.1.2 Utility Theory

In order to evaluate and compare solutions to problems, AI techniques frequently make use of utility functions. A utility function accepts solutions to a problem and reduces them to numbers which can be compared, with larger numbers indicating better solutions [20]. A simple utility function for the problem of which move to make in chess would be to count the number of pieces the player will have on the board by their next turn minus the number of pieces that the opponent will have at that time, which would give a favourable score to moves that capture the opponent's pieces without endangering the player's pieces. This simple example also illustrates the importance of selecting an appropriate utility function to maximize; the example function does not distinguish between the value of pieces or their positions on the board, and maximizing this function will not necessarily help the player to win the game which is likely the real goal.

In a multi-agent system, the utility of the system as a whole is usually some function of the well-being of each individual agent [8]. A simple and useful approach here is to define a utility function at the agent level, then add the utility values of all agents together (possibly weighted according to agent priority) to produce the utility of the complete solution. This

allows direct comparison in utility values between individual agents independent of which agents are being compared, simplifying calculation and allowing a more local approach. However, the nature of this utilitarian approach allows for local sacrifices to improve overall utility as long as the loss in utility by one agent is compensated by improvements in the utility of other agents. Finding these tradeoffs is usually the intention, but this must be considered when designing utility functions at the agent level to ensure that the intended behaviour is rewarded. Alternatively, an egalitarian approach would have the goal of maximizing the minimum utility value among individual agents, ensuring some baseline level of welfare for all agents.

1.1.3 Resource Allocation

Resource allocation problems deal with a collection of tasks that must be completed through use of a set of available resources. The paradigm of allocating resources to tasks can apply to problems in many different domains; one particular domain examined at length in this thesis is medical treatment, where resources are doctors and tasks are treatments for patients. At varying timescales, this general framework can represent problems such as computer scheduling (where resources are processors and tasks are programs), industrial manufacturing (where resources are factories and tasks are production orders), and fire control (where resources are fire fighting crews and tasks are fires that demand attention).

Depending on the domain, different assumptions may apply towards the treatment of tasks and resources. The concept of preemption is common in domains such as computer scheduling, where resources can be removed from a task in progress if there is another task with greater need for those resources. Tasks are frequently prioritized, encouraging solutions that put more effort into resolving higher-priority tasks. Some tasks may require more effort to complete than others, making the best allocation decisions less clear.

1.2 Overview

This thesis approaches the problem of multiagent resource allocation in environments with dynamic task arrivals, expanding on prior work in static task environments. We focus on a distributed approach for multiagent resource allocation, as solutions which rely only on local information can easily handle a changing environment where new tasks arrive over time. Environments allowing preemption grant a great deal of flexibility in these allocations, but our solution performs servicably when preemption is impractical. Techniques for resource reservation become useful in these irreversable allocation environments.

The proposed model operates as follows: Each task to be completed is controlled by a distinct task agent, while each available resource is controlled by a distinct proxy agent. All proxy agents are known to all task agents, but individual task agents are only known by the proxy agents they interact with; as a result, whenever a new task enters the system it is provided with a new task agent which can operate immediately without disrupting any agents already existing within the system. Each task agent interested in a resource will approach the associated resource's proxy agent and request the resource. Each proxy agent will then compare all of the task agents which have approached it, as well as any task agent currently using the proxy agent's resource, and decide which task agent should receive the resource. This decision is made greedily, based on optimizing local utility. In order to help connect this local estimate to the global utility function, each task agent learns two pieces of information about the allocation environment: Congestion (the level of competition for resources) and Churn (the rate of change in the environment). This information is used by task agents to decide which resources are best for their task, to determine how far ahead to plan, and to inform proxy agents of how well the task agent will be able to operate if it loses a resource.

The ability for proxy agents to revoke resources from tasks in progress is called preemption, which is used to recover from bad allocations which may arise from how a new task arrival changes the relative priority of tasks. The best decisions in one situation can become poor decisions in retrospect after the situation changes. An alternate approach to handling this problem of hasty decisions is to reserve resources for later use instead of allocating them immediately and revoking the allocation later. When preemption has some cost and the system has some awareness of future changes, reservations can provide more benefit than preemption. We examine a variation of our multi-agent resource allocation model which explicitly concerns itself with these switching costs, where allocating a resource only to take it back is worse than not allocating the resource at all. We also examine the most extreme case where these costs rise so high that tasks cannot be interrupted once they begin, thus making allocation irreversable.

Chapter 2 discusses some of the background material useful for understanding this thesis. Chapter 3 describes the initial multi-agent resource allocation problem with preemption as presented in prior works, along with algorithmic improvements and application to environments with dynamic task arrivals. Chapter 4 discusses the concept of opportunity cost and how it can be used to analyze the cost of reserving a resource for future use. Chapter 5 drops the assumption of preemption and explores the effects of resource reservation in this restricted environment.

The contributions of this thesis will be of interest to designers of multi-agent systems, as we provide a reference approach for handling dynamic task arrivals quickly with and

without preemption. The discussion and reference implementation for resource reservation indicates the cases where this technique benefits system designers.

Chapter 2

Background

A certain degree of background understanding is necessary to describe the techniques used in later chapters. Important concepts will be outlined here. Readers experienced in AI techniques relevant to multi-agent systems may use this chapter as a reference for any individual terms in later chapters that are unfamiliar.

2.1 Multi-Agent Resource Allocation

Multi-agent resource allocation approaches resource allocation problems by constructing a multi-agent system where agents are given responsibility for the resolution of tasks using a limited set of resources. Each agent may be responsible for one or more tasks, and the resources may have agents of their own to participate in negotiation.

Chevaleyre et al. [8] provide a useful introduction and definition of multi-agent resource allocation, describing it as "the process of distributing a number of items amongst a number of agents," further defined by what items are being distributed, how they are being distributed, and why they are being distributed. An allocation is then a specific distribution of resources among agents. Solutions to multi-agent resource allocations can vary widely based on the answers to these questions. What resources are being allocated matters, as divisible resources (like gasoline for cars) must be handled differently than indivisible resources (like spare tires for cars), just as resources which can be shared (like multitasking servers for programs) are different from resources which are only useful for one agent at a time (like pieces of clothing). The end goal of an allocation also affects design decisions, as simply finding a feasible allocation (e.g. picking volunteers with specific time constraints

to fill a set of time slots) requires a different approach from optimizing an allocation for some criteria (e.g. picking employees with time constraints to fill a set of shifts without overworking or underworking anyone).

The techniques in this thesis are best suited to indivisible resources which cannot be shared among multiple requesting agents. This gives meaning to reserving resources for later arrivals, as an allocated resource cannot be split between agents. These techniques are described in a fully distributed environment, where the resource allocation is slowly built up out of individual negotiations; resource reservations are made in a way that implicitly coordinates among resources and agents without the need for a centralized manager. The goal of this allocation is to maximize the utility of the system by minimizing costs, completing tasks as quickly as possible while accounting for differences in priority between tasks.

Certain domains make different assumptions, but in this thesis tasks require use of a single resource for a given length of contiguous time in order to be resolved, so changes in resource allocation may restart progress on interrupted tasks. Resources may be better suited to some tasks than others; a task may require less time to resolve if it is allocated a compatible resource. Some tasks may be more important than others.

While it is possible to have each agent be truly self-interested, as in marketplace environments where the agents are seeking to rent resources to resolve their tasks, this thesis does not perform the necessary game theoretic analysis to ensure that agents cannot benefit through misrepresenting themselves and their tasks. Such work is available elsewhere [19].

There are benefits to a multi-agent approach even if the agents are assumed to be truthful. Provided that agents make decisions based on a subset of the total system state (i.e. agents may need to know which resources are in the system, but not necessarily the specifics of tasks for which they are not responsible) then a multi-agent approach can be distributed and thus gain resiliency against single points of failure. Distributing this decision-making renders state changes less disruptive to the algorithm, which has similar practical benefits. Consider a centralized algorithm which consumes all available state information and outputs an optimal allocation. If this algorithm must be re-run for every new task that enters the system, there is potential for large amounts of wasted work, especially if the resulting allocation only differs in small ways. However, suddenly requiring large changes in an existing allocation may be realistically infeasible even if it results in a better overall solution. Localizing resource allocation decisions helps to resolve these concerns.

Multi-agent systems are largely defined by the protocols for interaction between agents [8]. Cooperative autonomous agents share only those pieces of information which

are necessary to resolve a dispute, but trust that the values reported by other agents are accurate. Agents can thus decide which resources are best to acquire based on available information, then confirm that this information was not too out-of-date by approaching the agents currently responsible for those resources. This idea is described by Michael Cheng [7] in terms of information-sharing agents that must coordinate. Cheng describes four types of agents which each make use of shared resources which can be overused, with each type of agent taking a different approach to communicating their resource usage to the other agents. Cheng's Type IV agents decide which actions to take based on possibly-stale local information, then include projected benefits and acceptable error thresholds when actually approaching the resources indicated by the plan. If the actual usage level of the resource indicates that the Type IV agent's choice was based on an incorrect estimate, then it informs the Type IV agent of the changes in the environment, but still allows the action to proceed as long as the action lost less value than the acceptable error threshold. If the acceptable error threshold is set as the difference between an agent's first-place and second-place choices, then this saves retry attempts as the Type IV agent only needs to change its decision when the environment has changed enough to make a different decision worthwhile.

2.2 Resource Preemption

A notable point of difference between resource allocation problems is whether the environment permits preemption, where a resource is taken from a task currently in progress and assigned to a different task. If the preempting task receiving the resource has significantly greater importance than the preempted task which loses the resource, then the overall welfare of the system is improved by allowing this preemption to take place. Note that the preempting task may not have been present in the system when the resource was first allocated; preemption thus serves as a fallback mechanism for responding to changes in the problem environment, and is especially important in environments with dynamic task arrivals that change the optimal allocation.

Preemption is a powerful tool for any resource allocation problem, but in many domains there are weaknesses to its use. For example, interrupted tasks may need to restart completely, unable to preserve any accumulated progress. These switching penalties naturally arise from resources that hold state relevant to the task, such as storage space in a network cluster, or the working memory of a human resource who is interrupted halfway through solving a problem. Even when some work can be preserved, there is a cost associated with recording the current progress in such a way that the task can be resumed later (whether

by the same resource or a different resource).

The costs associated with preemption prevent an algorithm from reallocating all resources every time the optimal allocation changes. This transforms the choice of initial allocation into a selection between mutually exclusive alternatives, which is best analyzed through the concept of opportunity cost [15] which is the value of the best alternative forgone. For example, if an algorithm can choose between gaining 20 utility by allocating resource A to task B before task C or gaining 10 utility by allocating A to C before B, then the decision to take the first allocation has an opportunity cost of 10 utility because the alternative allocation can no longer be taken. Reasoning with opportunity cost reminds system designers to consider all possible alternative actions, especially in stochastic environments where conditions can change. If the task B is currently not present in the system but could arrive shortly, and the utility of allocating resource A to task B before task C depends on the arrival time of B, then the decision to allocate A to C could be a better idea depending on when B actually arrives. Anticipating future arrivals can thus help improve resource allocation decisions.

A notable competing multiagent resource allocation system using preemption is presented by Paulussen et al. [17] for the medical domain, where agents (representing patients) are granted schedule slots with resources (representing doctors or hospital equipment, like operating rooms or diagnostic machines) on a first-come-first-served basis. From this initial allocation, agents can negotiate trades with each other to rearrange the schedule. The goal of each agent is to minimize the time they spend waiting for treatment; the lack of hard deadlines allows agents to compromise if doing so improves overall utility. Agents gain utility by being treated earlier, and this gain in utility is used as currency to compensate the agent which is treated later due to the trade. This currency could presumably be used later by the disrupted agent to secure better resources.

Examining Paulussen's model in more detail, the utility measure used is years of well-being, encapsulating both the time a patient spends waiting for treatment as well as any complications that a patient may face after receiving treatment. For practical reasons, degradation in patient health state is assumed to be linear; this also assists in practical implementation, as two points of evaluation are sufficient to fit a line for the patient's model. Paulussen's utility formula also incorporates stochastic treatment duration, though the end goal is still to minimize the amount of time before a patient is treated. In order to do so, the patient's agent may attempt to negotiate rescheduling, following this procedure [16]:

- Identify the resource slot with the highest increase in utility for your task, and approach the resource's agent.

- The resource's agent reserves that slot, and informs the agents of all patients who overlap that slot about the initiating agent's request.
- These affected agents attempt to secure replacement slots following this procedure, with the restriction that reserved resource slots cannot be requested. This repeats recursively, eventually finishing when the last patient agents secure resource slots which are unoccupied (i.e. at the end of the waiting list).
- The cost incurred by each displaced patient is reported back to the original resource agent, which compares the net cost to the benefit claimed by the initiator. If the benefit is higher, then the preemption occurs (and presumably currency changes hands); otherwise, the resource agent rejects the preemption and the requester cannot attempt to claim this slot again unless its ownership changes. Either way, the negotiation is complete.

This procedure is followed by one patient agent at a time until no further improvements are possible. Negotiations are assumed to not require any time, allowing an allocation to settle to a steady state whenever a patient's situation changes. Since the cost of giving up the current resource slot can only be determined by this recursive rescheduling procedure, this procedure cannot find swaps where a patient agent gives up its current resource slot in exchange for another resource slot, as a cycle of resource transfers cannot be evaluated.

2.3 Online Problems

This thesis offers an online approach to resource allocation. An online problem refers to a problem where not all information is available at the start of execution, with this information then usually spread out over the course of execution. This is in contrast to offline algorithms which have access to all the relevant information and are thus free to attempt brute-force solutions where the benefit of every possible solution is calculated in order to choose the best solution. Algorithms for online problems are evaluated by comparing their results to the optimal offline algorithm, using this best-case scenario as a reference point [12].

The quality of online algorithms can be quantified as a competitive factor, as described by Baruah et al. [4] in their analysis of real-time scheduling. The competitive factor r ($0 < r < 1$) of an algorithm indicates that the algorithm will achieve at least r times the value of an optimal offline algorithm on any arbitrary problem instance. Calculating the

bounds on a competitive factor is reminiscent of worst-case runtime analysis for algorithms, using an adversary to construct problem instances designed to exploit flaws in the online algorithm's decision-making process. In the proof bounding the competitive factor of an arbitrary real-time scheduler, the series of tasks described uses a carefully constructed series of 'bait' tasks which the adversary stops providing if the scheduling algorithm ever attempts to make use of them. The offline algorithm can then take advantage of these bait tasks once task arrivals are locked in, performing better than the optimal online algorithm. These absolute worst-case scenarios illustrate the power of hindsight, showing how the best decisions based on available information cannot always be the truly best decisions.

Randomness can help defeat this adversarial analysis. Awasthi and Sandholm [3] provide a similar proof that limits the effectiveness of online algorithms where an adversary manipulates the problem based on the algorithm's decisions. However, as long as the adversary must decide on the problem before it knows what random draws the algorithm receives, a randomized online algorithm can achieve a better competitive factor than a deterministic online algorithm (which the adversary could perfectly simulate before in order to construct a worst-case problem). The authors then go on to describe this generalized random case as a prior-free algorithm which acts without any knowledge of the distribution of arrivals into the system. Incorporating some awareness of arrivals, justified through use of medical records for the medical domain discussed in their paper, allows for online algorithms to make reasoned decisions based on the expected future arrivals as well as the current problem state.

For example, consider matching problems. An offline matching problem would present the algorithm with a bipartite graph (a collection of nodes connected by edges such that there are two groups of nodes where nodes are only connected to nodes in the other group; an example is where there is one group of job positions and one group of applicants, with edges connecting jobs to qualified applicants) and request a solution that creates as many matched pairs as possible, with each pair consisting of one vertex from each side of the graph connected to each other by an edge but with no vertex in more than one pair. Online matching problems [12] are provided with one half of a bipartite graph at the start of execution, and introduce one vertex at a time with its associated edges from the other half of the graph, only progressing when the algorithm has decided which vertex to match to the incoming vertex. This necessarily means that it is possible to construct online matching problems which are only solvable through random chance, such as a bipartite graph which is fully connected except for one vertex from the hidden side which has a single connection and arrives last. There is no way for an online matching algorithm to guarantee a solution to this case. Best-effort solutions may be necessary.

Online resource allocation problems usually withhold information about the tasks that

must be accomplished, as in real-world scenarios where an organization has some control over the supply of resources it has available but the work they perform is unpredictable. In the medical example, a hospital knows how many doctors are on staff but cannot control the rate at which patients appear. An offline version of this problem which was aware of task arrivals could carefully allocate resources to ensure that good resources are always available when high-priority tasks arrive, even in cases where the good resource provides less benefit to the system until after it has the opportunity to resolve the high-priority task.

2.4 Planning with Transfer-of-Control Strategies

Planning in AI refers to the selection of future actions which best solve a problem [20]. Plans may be conditional, providing a clear procedure to follow which decides on an action to take once necessary information has been collected. An example of a plan is the choice of roads to follow when driving from one city to another, which may be conditional if there is uncertainty over whether certain roads may be available. Planning techniques incorporate uncertainty to decide which courses of action are expected to be optimal after accounting for the risks associated with unfavourable conditions.

A transfer-of-control (TOC) strategy is described by Cheng in [6] as a sequence of transfer-of-control actions interspaced with wait times, where each transfer-of-control action is a request for another agent to assume the responsibility of a decision. This is based on the idea of "adjustable autonomy" [21], where agents have some ability to make decisions on their own as well as the ability to decide when to seek help from other agents or human users. An agent following a TOC strategy will approach the first agent in their strategy, wait for the specified time, then if no response has been received the agent will move on to the second agent in the strategy until the strategy ends. Only one step in the strategy is expected to provide any practical benefit, as the first agent that responds takes on responsibility for the decision being made. The value of these actions is thus judged by the expected improvement which each approached agent could provide, weighed against the probability of response and the wait time required for a response. TOC strategies are useful when agents are heterogenous, with some agents better suited to certain tasks than others; for example, the "Electric Elves" paper [21] describes agents responsible for scheduling meetings that make requests of humans when this improves decision quality, balancing the superior general reasoning ability of a human against the speed of a software agent.

TOC strategies can be used as a resource acquisition guide, as introduced in [10]. Here,

proactive agents are each responsible for resolving a task but have no power on their own, and must thus delegate this responsibility to a resource capable of resolving the task. Knowing when to give up on acquiring a valuable resource with a long wait time in favour of acquiring a reliable resource is important, and TOC strategies neatly encapsulate this decision process.

For an example of a TOC strategy for resource acquisition, consider an agent who plans to request $r1$ in the current tick, then $r2$ in the next tick if the first request is unsuccessful, then another attempt to request $r2$ in the tick after that if neither request is successful; this plan can be denoted $[r1, r2, r2]$. Each request has an associated change in utility, with the second request for $r2$ having a different utility change than the first request as time has passed between the two requests. Let the utility gain for each request respectively being successful be $[20, 16, 10]$. Each request also has a probability of success, which may change depending on how much competition there is for a given resource. Let these probabilities respectively be $[0.5, 0.75, 0.8]$. The value of this plan is not simply the sum of the utility at each step weighted by probability, as the plan will end at the first successful step; the total value of this plan is accumulated by weighting later steps by the probability that they are reached at all, for a final value of $0.5 \times 20 + 0.5 \times (0.75 \times 16 + 0.25 \times (0.8 \times 10)) = 10 + 0.5 \times (12 + 2) = 17$ utility.

Note that the TOC strategies provided in this example rely on discrete time steps, distinct from the TOC strategies described in Cheng's work [6] which spend a great deal of computation on finding the optimal length of continuous time to wait before declaring that the currently requested entity is unresponsive. Our work uses a synchronized clock to make decisions, so requested resources are never completely unresponsive; if an agent has not received a resource by the time the next resource request cycle begins, then the agent definitively knows that it has not received a resource.

2.5 Bother Modelling

AI systems often interact with humans, usually because the AI system produces a solution to a problem humans have (meeting schedule, car plan, movie recommendation, etc.). If the system is attempting to respond to a changing real-world scenario, then the optimal solution may fluctuate as conditions change, but if the magnitude of these changes is small then there is little incentive for the human users to spend the necessary time communicating with the system in order to use these optimizations. Similarly, solutions that require subjective evaluation (like recommendations) have an incentive to validate interim results in order to provide better solutions overall. However, computers are fast, and the benefit

humans gain from AI systems diminishes if the AI systems require micromanagement in order to provide useful solutions. A system which is frustrating to use does not serve well.

Bother modelling [9] attempts to quantify the strain placed on the users of an intelligent system and thus avoid excessively bothering them. A simple way to approach this is in terms of utility, assuming that humans incur some cost to switch between different solutions or provide feedback. Simply defining a tolerance value which must be exceeded in order for an improvement to be worth the user's time can help respect the user. It is natural to model this value exponentially, spacing out the times a user is bothered by the system and allowing users to relax and forget previous interruptions before requesting something from the user again.

The formula provided by Fleming [11] for calculating bother cost models several interim steps, building from the cost $c(I)$ of individual interactions I, discounted by the time since those interactions occurred $t(I)$, to accumulate a bother-so-far (BSF) value:

$$BSF = \sum_{I} c(I)\beta^{t(I)}$$

This is an initial value, with β as a discount factor $0 < \beta \leq 1$ indicating how quickly users forgive the system for past interactions. The example provided by Fleming examines a system that bothered the user three times, once each at 2 time steps ago, 7 time steps ago, and 13 time steps ago. If all interactions are equally penalized (with $c(I) = 1$) and the discount factor is set at $\beta = 0.95$, then the BSF value is $0.95^2 + 0.95^7 + 0.95^{13} = 2.11$.

Note that this value is purely based on the interactions themselves, before accounting for how some users may be more or less comfortable with interacting with the system. This value is further transformed into bother cost by the following formula:

$$bother = Init + \frac{1 - \alpha^{BSF}}{1 - \alpha}$$

The values for $Init$ and α are determined by rating the user's willingness w on a scale of 0 (for users that dislike the system) to 10 (for users that are very willing to help the system), and computing $\alpha = 1.26 - 0.05w$ and $Init = 10 - w$. These recommended formulas are intended to give a linear increase in bother cost for moderate users while unwilling users have an exponential increase in cost and willing users have a logarithmic increase in cost; the values for $Init$ can also be used to capture the inherent disruption posed by a task in order to differentiate unobtrustive requests (e.g. for the time) from demanding requests (e.g. helping someone move). For example, a willing user with $w = 9$ (and thus

$\alpha = 1.26 - 0.05w = 0.81$ and $Init = 10 - w = 1$) that has been bothered as above for a BSF value of 2.11 will have the cost of bothering calculated as:

$$bother = Init + \frac{1 - \alpha^{BSF}}{1 - \alpha} = 1 + \frac{1 - 0.81^{2.11}}{1 - 0.81} = 2.89$$

By contrast, an unwilling user with $w = 1$ (and thus $\alpha = 1.26 - 0.05w = 1.21$ and $Init = 10 - w = 9$) would have the cost of bothering calculated as:

$$bother = Init + \frac{1 - \alpha^{BSF}}{1 - \alpha} = 9 + \frac{1 - 1.21^{2.11}}{1 - 1.21} = 11.36$$

Adjusting these parameters allows the bother model to recognize differences in how users react to interruptions by an intelligent system.

An alternate approach taken by Doucette [10] models bother probabilistically, indicating that busy users are less alert to changes in the AI system's solution. This approach is less straightforward to interpret, but intuitively captures the idea that theoretically optimal solutions may not translate well into practice. In this framework, every interaction with users has a probability of failure determined by a function of the utility improvement from the action the system recommends, as well as the BSF value calculated similarly as in Fleming's work. This indicates the reasonable decision by a human to refuse a course of action which only provides a tiny improvement in system utility.

Both bother modelling and TOC strategies are useful in describing mixed-initiative systems [1] where an AI system decides whether to attempt to solve problems directly or whether it should request the expert opinion of a human. However, these techniques are applicable to any application which has not been entirely automated, as at some point the AI system's solution must be implemented in reality, and as conditions inevitably change the AI system must be able to make a reasoned tradeoff between the ideal solution and the practically attainable solution.

Chapter 3

Multiagent Resource Allocation with Dynamic Arrivals

3.1 Introduction

Multiagent resource allocation problems arise when there are multiple parties with conflicting needs that must compete for a limited pool of resources. In dynamic environments, new agents may arrive with their own needs, which may be more or less important than the needs of parties already present. A multiagent system represents each user with a software agent that negotiates on their behalf; if there is some shared assumption about the relative importance of tasks, then the agents can generate an efficient allocation to satisfy the most important needs as quickly as possible.

When taking a resource from a task in progress, viewing this transfer in a vacuum leaves out important information. Both the requesting task and the resource-holding task have other options available to them, and ignoring these fallback options may prevent optimal allocation decisions. Our proposed solution relies on allowing for effective communication of resource needs and backup plans between agents.

3.2 Overview of Proposed Techniques for Resource Allocation

3.2.1 Coordination Structure

Our proposed solution for multiagent resource allocation rests on mapping out communication between Task Agents (responsible for obtaining resources in order to complete user-provided tasks) and Resource Proxy Agents (responsible for providing resources to the best available tasks). The proposed system handles dynamic task arrivals by reasoning about which actions to take at each time tick, with new arrivals participating in this reasoning immediately. Task Agents reason about which resources to request based on plans that are formulated (and reformulated under certain conditions) by learning about the environment through interactions with other agents. Resource Proxy Agents deal with the requests of Task Agents at each time tick, ultimately deciding which Task Agent receives the resource by reasoning about the utility which that Task Agent will gain by receiving the resource (balanced against the utility lost by the resource's current owner, if any). The Resource Proxy Agent is able to make this decision in part due to knowledge of the underlying task's Type (revealed by Task Agents when requesting resources) and in part due to the expected utility of the backup plan for any Task Agent which may be displaced (a value which is revealed honestly by a Task Agent when a resource that the Task Agent owns is requested).

The coordination process is as follows:

1. Each task agent computes a plan of how to complete its task. This plan consists of a series of resources which the task agent will request in turn over several time ticks, planned out to a certain horizon.

2. At each time tick, before making a planned request, the request is verified through a quick check with the resource proxy agent. If the resource proxy agent reveals that the request cannot possibly succeed, the task agent adjusts the plan before requesting a resource for that tick.

3. After verification, each task agent requests a resource from a resource proxy agent.

4. Resource proxy agents each evaluate all incoming requests, selecting a single task agent which shall receive the resource. If the resource is already owned by a task agent, then requests are evaluated with respect to preempting that resource from its owner.

5. A task agent whose request is granted will immediately compute a new plan based on ownership of the resource which the task agent received. A task agent whose request is refused will move on to the next step in its current plan once the next time tick arrives.

In addition to evaluating resource requests based on utility, Resource Proxy Agents may deny consideration of requests based on accumulated bother. This represents a resource which can be overworked by the resource allocation system. Such a resource may disengage from the decisions of the system and continue working without listening to recommended optimization. Subsequent chapters discuss the logical extension of this bother modelling, where resources cannot be interrupted while working thus making resource allocations irreversible.

3.2.2 Coordination Details

Our solution relies on certain parameters which are known by task agents and resource proxy agents. In particular, we assume that the time required for a given resource to complete a given type of task is known (provided that the task has uninterrupted access to the resource for that time; interruptions are assumed to clear all accumulated progress toward completing a task). Each resource is assumed to complete some task types faster than others. When a task agent receives a resource, it begins tracking a countdown timer to the task's completion; the time remaining before completion is reported along with task type when a task agent requests a resource. Resource proxy agents track a similar countdown timer beginning when the resource is allocated to a task, thus allowing the resource proxy agent to know at all times how much time is left before the resource will finish the current task.

When a resource proxy agent is evaluating resource requests, the calculation is simple if the resource is not currently owned; the utility gain for each requester is calculated, and the task agent with the highest utility gain is selected. Utility is based on minimizing the total time a task spends in the system, weighted by task type. If the resource is owned, then three utility values are involved in this calculation: the utility gained by the winner, the utility lost by the loser (the current owner in the case where the resource is taken), and the utility which the loser could regain by making its own resource requests. If the total change in utility after accounting for these three values is positive, then the resource proxy agent will allow the preemption, taking the resource from its current owner and granting that resource to the requester. The gain and loss in utility can be calculated directly from the type of each task and the known time required to complete each task.

The potential utility regain is reported by the task agent currently holding the resource to the resource proxy agent, which requests this information when another task agent requests that resource. The task agent calculates this regain value from the expected utility of its plan.

Task agents compute plans which indicate the resources which the task agent will request in order to improve their utility. In this thesis, these plans are described as Transfer-of-Control strategies [21, 6]: A series of resource requests over time. Notably, the goal of such a plan is to acquire a single resource; once a single request succeeds, then the plan has served its purpose, as that resource is capable of completing the task if given enough time. If the first request in a plan does not succeed, then the task agent will request the second resource specified in its plan at the next time step, and so on until a request succeeds or the plan requires revision. Task agents will continue to generate plans for task completion until the task is complete. The primary conditions for plan revision are:

1. A successful resource request. Once the task agent possesses a resource, it will generate a new plan, allowing the task agent to recover if that resource is taken away at a later time.

2. An unsuccessful verification. If a plan includes a resource request which cannot succeed, this indicates that the resource allocation environment has changed substantially since the plan was constructed, so a new plan must be constructed.

3. Reaching the end of the plan. If there are no more steps in a task agent's plan, then that task agent must compute a new plan to follow.

Task agents compute plans to optimize expected utility, based on both utility gain and the probability of success for each request. Resource proxy agents provide the information necessary for planning. Our solution uses the optimizing algorithms detailed in [10] to compute these plans.

The final element involved in resource request evaluation is bother. When task agents request a resource, the resource proxy agent will always deny the requests from all task agents other than the task agent with the highest positive increase in utility. However, the resource proxy agent does not always grant the request from even the utility-maximizing task agent, and may reject all requests on the basis of the bother model. This is used to emulate scenarios such as humans who are too busy or annoyed to check the recommendations of the system. A Bother-So-Far (BSF) value is tracked by resource proxy agents, accumulating whenever the resource is interrupted and slowly decaying over time as the

resource forgives and forgets. A P_{stop} function uses this BSF value to determine the probability with which the resource proxy agent rejects all requests. Resource proxy agents use this function to report this probability value to task agents during planning in order to determine the probability of success when requesting a given resource.

3.2.3 Summary of Agent Communication

In short, the information transferred between task agents and resource proxy agents is as follows:

Task Agent to Resource Proxy Agent: Report task type and time until task completion when making resource requests or verification requests. Report plan value when asked. Request expected utility of gaining a resource when constructing plans.

Resource Proxy Agent to Task Agent: Report expected utility of resource requests and probability of request failure when asked. Respond to requests for verification immediately, indicating if the request could possibly succeed (yes or no). Respond to requests for a resource after collecting all requests during a tick, indicating whether the requester receives the resource (yes or no).

We now provide our proposed resource allocation solution in full, clarifying both the model components required and the algorithms describing the exact reasoning of this allocation process.

3.3 Model

The general approach of this model is to represent the decision-making process when many tasks that need to be completed compete for a limited set of resources in a scenario where some tasks are legitimately more important. Agents representing each task will pick a resource to approach, then all contenders for a resource (including the agent currently using the resource, if any) are compared, and the resource is awarded to the most important task. The reader is advised to keep this basic overview in mind as the following fills out all of the details over how this process is modelled and conducted.

The model used in this work is composed of four primary components: **Tasks**, **Task Agents**, **Resources**, and **Resource Proxy Agents**. This is consistent with the model design in [10] and [18], with differences highlighted later in Section 3.6. Each of these components is explained in more detail below.

Tasks represent the actual work that must be performed by the system for its users. Resources are allocated to agents in order for them to perform tasks on behalf of their users: for example, in a medical domain each task may represent a patient with some injuries or illnesses that must be resolved by performing work. The term 'task' is used in computer scheduling to represent discrete bundles of work, but can represent work in other domains. While some literature discussing multi-agent resource allocation discusses the allocation of resources to agents directly [8], we define tasks separately to clearly distinguish between the agents responsible for performing work and the agents responsible for managing resources. In our framework, a task is characterized by:

- A task type θ_t, used in expected utility calculation. This allows different tasks to behave differently, requiring different resources and holding different priority levels in the system. Important tasks may be referred to as being high-value or high-severity, reflecting the increased attention they receive.
- A resource r currently assigned to the task, or $NULL$ if there is no such resource.
- The length of time *until_complete* which shows when the task will be completed if it continues to use its current resource. This is a state variable, tracked in order to determine the potential losses from removing the current resource from that task.

Task Agents are agents that each act on behalf of a task in the system, and negotiate with resource proxy agents to acquire resources. Every task has its own task agent, which is assigned when the task first enters the system. Task agents learn about the resource allocation environment they inhabit, building plans about which resource proxy agents to approach as well as making and refining estimates used during planning and negotiation. A task agent is characterized by:

- The task t which the task agent represents.
- A Transfer of Control (TOC) plan *plan*, consisting of a list of resource proxy agents which the task agent will approach in sequence. When following a plan, the task agent approaches the next resource proxy agent in the list, then removes it from the list if the request fails; if the request succeeds, then the task agent rebuilds the plan based on the new resource it has acquired for the task.
- The cached valuation eu_{plan} of the task agent's plan; while this can be recomputed at any time from *plan* by using the expected utility of each step in the plan, naming this valuation allows for easier discussion. This value is reported during negotiation with

other task agents to acquire resources, modified by the task agent's view of system congestion (see below).

- Learned values $m_{congestion}$ and m_{churn} recording the task agent's experiences in the resource allocation environment. System congestion is measured by the number of times the task agent loses its resource to another task agent, and churn is measured by the number of times the task agent's plans are rejected due to changes between the plan's creation and execution. These are described in more detail later, along with how they alter task agent planning.

Resources represent the resources available which can be used to resolve tasks if a resource is allocated to a task. It is assumed that each resource continues to provide work at full capacity over the lifetime of the system; however, there are allowances in the model for unreliable resources which occasionally ignore or miss updates from the software system (see BSF below). Resources are characterized by:

- A resource type θ_r, used to determine how much time a resource requires to complete a given task. This allows different resources to offer varying levels of work, and also permits specialists which perform certain types of tasks much better than others.

- A task t currently being performed by the resource, or $NULL$ if the resource is idle. Note that the task t's value *until_complete* can also be used to determine how long the resource will be occupied, since the resource will only be idle again when the task completes.

- The bother-so-far value BSF recording the mental effort expended by a human resource in attempts to follow the system's advice. A high value here increases the chance that the resource may ignore requests from the system, whether because the system's reported gain in expected utility is too small to be worth the trouble of switching tasks or simply because of information overload. This value exponentially decays over time, but different resources may tolerate different levels of bother. Bother modelling can be used to represent any resource which is disrupted by frequent changes in ownership, not just humans.

Resource Proxy Agents are agents that each act on behalf of a resource in the system, listening to requests made by task agents and choosing which requests to pass on to the resource (who may or may not be able to respond - see above). It is assumed that resource proxy agents are visible to all task agents in the system, though the converse is not necessarily true as task agents are only known by the resource proxy agents they interact

with; as a result, it is much easier to add task agents to the system as no-one needs to be updated on the existence of new tasks except the task agent responsible for that task. Resource proxy agents are characterized by:

- The resource r which the resource proxy agent represents.
- The task agent a which currently has ownership of the resource. This means that the resource is currently working on the task $a.t$ represented by that task agent.

The elements of the model above leave a number of key functions undefined, which must be specified to fit the domain being modelled. The necessary functions are:

- $completion(\theta_r, \theta_t)$ for calculating the time required for a resource of type θ_r to complete a task of type θ_t. This value is for a contiguous block of time spent working on the task, with the assumption that any interruption completely resets progress.
- $EU(old\_completion, new\_completion, \theta_t)$ for calculating the change in expected utility between a task of type θ_t being completed in *new_completion* time rather than in *old_completion* time. For example, tasks could cost the system a constant amount based on their type as long as they are in the system, thus prioritizing fast task completion.
- $P_{stop}(\Delta EU, BSF)$ for calculating the probability that a resource with bother level BSF will stop considering a system request which changes total expected utility by ΔEU (calculated by evaluating the EU function above for every task whose completion time is changed by the request and summing these values).
- $bother\_increase(\Delta EU, BSF, \theta_r)$ for calculating the increase in bother level for a resource considering a request which changes total expected utility by ΔEU, when the resource has bother level BSF and type θ_r.

3.3.1 Motivating Example

With the fundamentals of the model described above, consider this instantiation of the model in the medical domain. The following will help clarify what needs to be specified in order to apply the abstract model to a concrete scenario.

The tasks will represent patient treatments, where completion of a task corresponds to the patient being cured. Task types represent conditions; in this model, task types are natural numbers, with larger numbers corresponding to more severe patients. To reflect this, the

EU function is defined as $(old\_completion - new\_completion) \times \theta_t$ to capture how shorter completion times are preferred, weighted by the task type (thus prioritizing patients with severe conditions). For the special case where completion time is unbounded (for patients not receiving any assistance), let $EU(\infty, new\_completion, \theta_t) = (12 - new\_completion) \times \theta_t$, thus placing an implicit upper limit on the length of time a patient could possibly require for treatment in this model. To demonstrate a wide spread in patient severity, take types ranging from 1 to 500, thus providing some grounding for the meaning of a point of utility as the minimum amount of harm a patient may incur in a given unit of time.

The resources will represent doctors, who may specialize in the treatment of particular conditions. Resource types represent those skill sets; for a model with N different task types, let resource types be drawn from the set $\mathbb{N}^N$ so that each resource has a skill level denoted by a natural number for each task type. To use these resource types to reflect expertise, define the *completion* function as $11 - \theta_r[\theta_t]$, where $\theta_r[x]$ is the resource skill value for the x^{th} task type, and skill levels range from 0 to 10. Thus, completion times will vary from 1 to 11 steps based on how skilled the doctor is at treating the patient's condition. For example, in a system with 5 patient types there may be a skilled doctor with skill vector $[8, 6, 9, 7, 10]$ who can treat severity-5 patients in a single step while still being able to treat other patient types quickly, while the operator of a specialized piece of medical equipment may have the skill vector $[0, 0, 10, 0, 0]$ indicating that severity-3 patients have a condition which the equipment can help treat quickly but the equipment isn't useful to any other patient types. With the *EU* function described above, differences in skill level mean that some resources can complete tasks faster, thus improving utility. Differences in skill for higher-severity tasks can provide more of a utility difference, but skill with low-severity tasks prevents a resource from needing to spend lots of time resolving those low-severity tasks.

Next, as the resources in this system are human, the P_{stop} and *bother_increase* functions should be defined to model the effects of request overload. Each request should raise the doctor's bother level, but large improvements should mitigate this bother increase as the doctor sees the good that the system has to offer. Consider the following sigmoid P_{stop} function and piecewise *bother_increase* function:

$$P_{stop} = 1 - (1.5 - S(BSF)) \times (0.5 + 0.5 \times S(\Delta EU))$$

$$\text{where } S(t) = \frac{1}{1 + e^{-t}}$$

$$bother\_increase = \begin{cases} 1.25 * BSF + 1 & : \Delta EU < -50 \\ 0.75 * BSF + 1 & : \Delta EU > 50 \\ BSF + 1 & : otherwise \end{cases}$$

The example P_{stop} function handles a large scale of values, introducing the probability that the doctor will view incoming requests (ranging from 1 at low bother to 0.5 at high bother) and multiplying this by the probability the doctor will accept based on the proposed utility increase (ranging from 0 for negative utility changes to 1 for large positive changes); subtracting the probability of accepting the request from 1 produces the probability that the request will be rejected. The example *bother_increase* function allows obviously-beneficial requests to place less mental burden on resources, while placing an additional penalty on obviously-bad requests that are somehow issued (possibly from task agents acting on out-of-date information).

Taken together, the definition of the four functions along with the domain definitions for task types and resource types provides a full system which can interact with the algorithms used in this chapter.

3.4 Formal Problem Specification

Consider a resource allocation problem as a tuple (T, R) of tasks and resources, and the following functions:

1. $completion : T, R \mapsto \mathbb{N}$, taking a task and a resource and returning the amount of time required for that task to be completed by that resource.

2. $EU : \mathbb{N}, \mathbb{N}, T \mapsto \mathbb{R}$, taking the old and new completion ticks of a task and returning the change in utility from this change in completion time.

3. $arrival : T, \mathbb{N} \mapsto \{true, false\}$, taking a task and a tick and returning whether that task has entered the system by that tick.

4. $P_{stop} : R, \mathbb{N}, T \mapsto [0, 1]$, taking a resource at a given tick requested for a given task and returning the probability that the resource will be too bothered to consider the request. Further constraints may be placed on P_{stop} to model the accumulation of bother over time and incorporate the expected utility change from allocating the resource to the given task.

5. $random : R, \mathbb{N} \mapsto [0, 1]$, providing a random value for each resource at each tick which is used to test whether the resource ignores requests as indicated by P_{stop}.

The solution to such a resource allocation problem is an *allocation*, defined as a function over $R, \mathbb{N} \mapsto \{T \cup \emptyset\}$ taking a resource and a tick and returning the task (if any) that the resource is allocated to at that tick. An allocation must satisfy the following constraints:

- $\forall t \in T, \quad \exists r \in R, n \in \mathbb{N} \, [allocation(r, n+i) = t \; \forall i \in [0, completion(t, r)) \wedge$
  $\forall q \in R, m \in \mathbb{N} \, [m \geq n + completion(t, r) \implies allocation(q, m) \neq t]]$
  (all tasks are completed, and not allocated further resources after completion)
- $\forall t \in T, r \in R, n \in \mathbb{N}, \neg arrival(t, n) \implies allocation(r, n) \neq t$ (tasks cannot be allocated before they arrive)
- $\forall t \in T, r \in R, n \in \mathbb{N}, (allocation(r, n) \neq \emptyset \wedge random(r, n) < P_{stop}(r, n, t)) \implies allocation(r, n) = allocation(r, n+1) \vee allocation(r, n+1) = \emptyset$ (sufficiently bothered resources are not reallocated to different tasks, though they may finish tasks)

Further, an allocation's utility can be determined by:

$$\sum_{t \in T} EU(\arg\min_{n \in \mathbb{N}} (arrival(t, n) = true), \arg\max_{n \in \mathbb{N}} (\exists r \in R \; allocation(r, n) = t) + 1, t)$$

This accumulates the change in utility for each task, from instantaneous completion (completion at time of arrival to the system) to the actual completion time (one tick after the last tick at which a resource is allocated to the task). An optimal allocation maximizes utility. Provided that utility is maximized when this difference in completion time is minimized, the theoretical optimal utility (which may not be reachable if there are insufficient resources) is reached when every task is completed in minimal completion time. This is given by:

$$\sum_{t \in T} EU(\arg\min_{n \in \mathbb{N}} (arrival(t, n) = true), \arg\min_{n \in \mathbb{N}} (arrival(t, n) = true) + \min_{r \in R} (completion(t, r)), t)$$

As an online problem, the functions *arrival* and *random* cannot be evaluated until the time tick used as the natural number argument has arrived.

Table 3.1: A table of algorithm dependencies.

Algorithm	Calls	Is Called By
Main Loop (1)	2, 7	n/a
step_agent (2)	3, 4, 5, 6	1
update_models (3)	n/a	2
gen_plan (4)	n/a	2
allow_transfer_attempt (5)	9	2
request_transfer (6)	9	2
step_resource (7)	8	1
commit_transfer (8)	9	7
compute_preemption_cost (9)	n/a	5, 6, 8

3.5 Algorithm

In discussing the model, there have been several mentions of how the individual agents act in order to solve resource allocation problems. An algorithmic description of this behavior follows; Appendix A has a fully-detailed example of how these algorithms operate in a small system. Table 3.1 summarizes how these algorithms interact.

At base, the algorithm formally describes a series of negotiations between groups of agents. Some agents hold resources already, but have developed *contingency plans* describing what they would do if the resources were preempted. Other agents want to preempt these resources. When a preempting agent requests a resource, the proxy agent associated with the resource, which has a model of the utilities for both allowing and denying the preemption, makes a decision about whether or not the utility of the whole system is served by that action. Throughout, agents learn models of the global environment to facilitate their local decision making when requesting new resources. New requests for resources are continually admitted into the system, with the aforementioned reasoning occurring at each time step.

Algorithm 1 describes our proposed solution's overall behavior: each round, new tasks enter the system and receive agents; task agents submit requests for resources to the proxy agents; resources are each allocated to the task agent which made the best request for them as determined by their proxy agent; and completed tasks leave the system. While described here as a central loop, this simulates the functioning of a distributed system using a shared clock to synchronize negotiation rounds; task agents act in random order (see line 10) to illustrate that no order constraint is imposed as long as all task agents act before all resource proxy agents. To allow for dynamic task arrivals, we have a function

Algorithm 1 The **main loop**, called to begin processing a new resource allocation task.

LET A be the set of agents.
LET T be the set of tasks.
LET R be the set of resources.
LET k be the maximum allowable plan length.
LET $T_a(tick)$ be the list of tasks that arrive at a given tick.
LET $tick$ be 0.
LET $T_{complete} = \emptyset$ and $T_{added} = \emptyset$
while $\exists t \in T$ s.t. t is incomplete **do**
    ASSIGN every task in $T_a(tick)$ to an agent in A.
    for all $a \in permute(A)$ **do**
        $step\_agent(a)$ **(alg. 2)**
    end for
    for all $r \in R$ **do**
        $step\_resource(r)$ **(alg. 7)**
    end for
    for all $t \in T$ **do**
        if $processing\_complete?(t)$ **then**
            Mark t as complete.
            Record total cost incurred by t for later evaluation.
        end if
    end for
    $tick \leftarrow tick + 1$
end while

Algorithm 2 An algorithm describing **step_agent(a)**, the behaviors of agents at each time step in the main loop.

1: UPDATE a's environmental model **(alg. 3)**
2: **if** $a.request\_success = TRUE$, **or** $a.plan = NULL$, **or** $|a.plan| = 0$ **then**
3: $a.plan \leftarrow gen\_plan(a, R/a.task.current\_resource)$ **(alg. 4)**
4: $a.request\_success \leftarrow FALSE$
5: **end if**
{Verify that the next step in the plan produces a net increase in expected utility.}
6: LET r be the next resource in $a.plan$.
7: **if not** $allow\_transfer\_attempt?(r.proxy, a.task)$ **(alg 5) then**
{If our plan is poor, make a new one.}
8: $a.plan \leftarrow gen\_plan(a, R/a.task.current\_resource)$ **(alg. 4)**
9: SET $r \leftarrow$ the next resource in $a.plan$.
10: **end if**
{Attempt to acquire a resource...}
11: **if** $allow\_transfer\_attempt(r.proxy, a.task)$ **(alg. 5) then**
12: $request\_transfer(r, a.task)$ **(alg. 6)**
13: **if** $a.best\_request = r$ **then**
14: $a.request\_success \leftarrow TRUE$
15: **end if**
16: **end if**
{Update and reappraise $a.plan$}
17: $a.plan \leftarrow rest(a.plan)$
18: $a.eu_{plan} \leftarrow appraise(a.plan)$ **(eq. 3.5)**

Algorithm 3 An algorithm describing **update_models(a)**, the learning process for the agent's learning models.

LET $a.m_{churn}$, $a.m_{congestion}$ be the corresponding models.
SET $a.m_{churn}.value \leftarrow a.m_{churn}.value \times \alpha$
{If our plan was updated in the last time step **(alg. 4)**}
if $a.m_{churn}.updated\_plan = TRUE$ **then**
    LET c be the cost of regenerating a plan.
    SET $a.m_{churn}.value \leftarrow a.m_{churn}.value + c$
    SET $a.m_{churn}.updated\_plan = FALSE$
end if
SET $a.m_{congestion}.value \leftarrow a.m_{congestion}.value \times \gamma$
{If our resource was preempted recently **(alg. 6, 8)**}
if $a.task.current\_resource = \emptyset$ **and** $a.m_{congestion}.preemption$ **then**
    LET d be the cost of a preemption event.
    SET $a.m_{congestion}.value \leftarrow a.m_{congestion}.value + d$
    SET $a.m_{congestion}.preemption = FALSE$
end if

Algorithm 4 An algorithm describing **gen_plan(a,R)**, the task agents' plan generation methodology

LET $plan\_length \leftarrow$ **(eq. 3.2)**
LET *method* be a TOC strategy generating algorithm.
LET $plan \leftarrow method(R, a.plan\_length)$ **(maximizing eq. 3.1)**
SET $a.m_{churn}.updated\_plan \leftarrow TRUE$
SET $a.eu_{plan} \leftarrow appraise(plan)$ **(eq. 3.5)**
return *plan*

parameter $T_a(tick)$ (see line 5) which lists the new tasks that arrive in a given tick of the simulation. All the associated subroutines are discussed below.

The behavior of task agents is found in Algorithm 2, and is summarized as four overall operations. First, the agent generates a plan for satisfying the resource needs of its user, provided that the agent has no such plan in the present step; this could be either because the previous plan succeeded and needs to be replaced, or the simulation is just starting and the agent has no plan yet, or the previous plan ran out of steps without succeeding (lines 2–5). Note that if the agent requests a resource and is successful, it will still generate a "backup plan" (the utility of which can be revealed to resource agents if the task agent is a candidate for preemption). This plan may include resources that are even more valuable than the new resource, so the agent never stops attempting to improve the expected utility of its task. The agent's plan may make resource requests that fail when other task agents with more important tasks also attempt to acquire the same resource. Second, the agent verifies that the action specified in its plan for the current time step can still increase utility (thus determining whether the information used to build the plan is still current), regenerating the plan if this is not the case (lines 6–10). Third, the agent requests ownership of the resource specified by the current step of its plan, allowing the resource's proxy agent to decide whether to grant this request (lines 11–16). Fourth, the agent removes the completed action from its plan and re-evaluates the expected utility of the remaining steps in the plan (lines 17–18).

Algorithm 4 contains the process for generating agents' plans. In this work we model each plan as a Transfer-Of-Control (TOC) strategy [21]: a series of attempts to acquire resources, with each new step only executed if the previous step has not been successful. In our implementation we use an unconstrained dynamic program to generate these TOC plans [10]; however, any reasonable planning technique could be used here provided that it can generate plans that maximize the expected utility. The appraise function called on line 5 computes the valuation function V which the planning technique maximizes:

$$V(X) = EU(X_i) \times P(X_i) + (1 - P(X_i)) \times V(X \setminus X_i) \tag{3.1}$$

where X is a list of actions, X_i is the first action in that list, EU is the expected gain in utility if the action succeeds, and P is the probability of the action succeeding (based on the bother model). Both EU and P can be computed from the domain-specific functions EU and P_{stop} mentioned above; note that the total expected utility of an action is the sum of the utility changes for each task that experienced a change in expected completion time. This utility value is calculated by the resource proxy agent using Algorithm 9 and provided to the task agent.

Algorithm 3 describes the learning procedures agents use to model their environment. The environmental model consists of two learned parameters, each of which ranges from 0 (an ideal system, requiring no corrections) to ∞ (an overwhelmed system requiring vast corrections). The parameter $a.m_{churn}$ is agent a's estimate of the extent of *churn* in the system. This is learned using "churn events" [10]: interactions where the agent discovers its plans for acquiring resources are based on information that is no longer true because the environment has changed. These events are triggered on line 4 of Algorithm 4 every time the agent has to regenerate its plan. The model's internal value is used in Algorithm 4 at line 1 to determine how much effort should be put into planning out actions further into the future. In particular, the plan length prediction is given by:

$$plan\_length = \lfloor \frac{k-1}{1 + a.m_{churn}.value} \rfloor + 1 \tag{3.2}$$

where k is the maximum allowable plan length measured in time steps (limit of computational resources), and $a.m_{churn}.value$ is an exponentially weighted decaying sum of the number of churn events experienced by the agent, given by:

$$a.m_{churn}.value = \sum_{i \in I} c\,\alpha^{t_i} \tag{3.3}$$

where I is the set of churn events i, c is the cost of an event, α is a decay rate, and t_i is the time since the event (the parameters c and α can be adjusted to model different levels of sensitivity to changes in the environment). Thus, the more frequently the plan is regenerated, the higher the churn value is, and the shorter new plans are. If the environment is stable enough for plans to be reliable, then the churn value will decay over time and permit the creation of longer plans.

The congestion model is a similar system, but measures the degree to which resources the agent needs are in demand. The parameter $a.m_{congestion}$'s value is computed in a manner identical to that of the churn model in Equation 3.3, but using parameters γ and d in place of α and c respectively:

$$a.m_{congestion}.value = \sum_{i \in I} d\,\gamma^{t_i} \tag{3.4}$$

Congestion events occur when a held resource is taken before an agent's needs were completely satisfied. As a result, an agent will adjust the expected utility of its plan per Equation 3.5 below (using $V(X)$ from Equation 3.1).

$$a.eu_{plan} = appraise(X) = \lfloor \frac{V(X)}{1 + a.m_{congestion}.value} \rfloor \tag{3.5}$$

As with churn, the congestion model is self-balancing: An agent that frequently loses resources will build up high congestion, thus reporting small values for plan utility, lowering the chance that the agent will lose further resources.

The resource proxy agent has three behaviors. First, in Algorithm 5, it filters preemption requests for the associated resource so that only those with positive net utility are considered. Second, it can compute the expected utility of allowing a preemption to take place when given information about a task by a task agent requesting preemption, since it knows what its resource is currently doing (and the associated BSF value). This behavior is specified by Algorithm 9. The net gain in utility is computed from the change in expected utility for both the preempting and preempted task if the preemption is allowed, tempered by the change in expected utility for both if the preemption is not allowed (the case where the preempting task needs to wait for the preempted task to finish). Task agents can call upon proxy agents to provide this information when plans are being generated, though changes in resource ownership may result in these values becoming stale in later ticks. Third, these behaviors come together in Algorithm 6 which reasons about the incoming requests for a resource. At any given time tick, all agents interested in resources can thus make requests, and these requests compete against each other. The proxy agent must determine whether a preemption should be allowed, based on the task information provided by task agents attempting to acquire the proxy agent's resource. The requests are evaluated using Algorithm 9, and the request with the highest positive improvement in expected utility is then forwarded to the underlying resource. This batching of resource requests is particularly valuable when a resource has just completed a task, as otherwise low-priority tasks could otherwise give up a reliable resource to secure a valuable open resource, only to be displaced by the next-highest priority task. Forcing the tasks to compete against each other on each tick prevents spurious preemptions.

In Algorithms 7 and 8 we see the behaviors of resources within the system. The best request of the timestep (as evaluated by the resource proxy agent) is evaluated by the resource at this point. Since we are modelling human-like resources, we assume that a resource has some discretion about accepting a request for preemption. Request evaluation is specified in Algorithm 8, which begins by increasing the bother of the resource, simulating the mental effort required to assess the request. The resource then accepts the request unless the bother model prevents it as measured by the probability function P_{stop}, which is specific to the problem and abstracts the probability of a resource with a given bother level BSF failing to address a new task given that doing so has an expected utility of net_EU. If the request is accepted, appropriate state variables are adjusted to represent the transfer and the task's processing time is set. Note that the expected utility gain is estimated by the resource's proxy agent in Algorithm 9, using the eu_{plan} value provided by the potentially-

Algorithm 5 An algorithm describing **allow_transfer_attempt(proxy,task)**, the behaviors of a resource proxy agent confirming a preemption request.

1: **return** true if $compute\_preemption\_cost(proxy.owner, task, 0) \geq 0$, false otherwise. **(alg. 9)**

Algorithm 6 An algorithm describing **request_transfer(r, task)**, for recording the best request made each round.

{Check against the best request so far}
{If there are no requests yet, this is the best}
1: **if not** $exists(r.best\_request)$ **or**
$compute\_preemption\_cost(r, task, 0) >$
$compute\_preemption\_cost(r, r.best\_request, 0)$ **(alg. 9) then**
2: **if** $exists(r.best\_request)$ **then**
3: $r.best\_request.request\_success \leftarrow FALSE$
4: **end if**
5: $r.best\_request \leftarrow task$
6: **if** $exists(r.current\_task)$ **then**
7: $r.current\_task.a.m_{congestion}.preemption \leftarrow TRUE$
8: **end if**
9: **end if**

Algorithm 7 An algorithm describing **step_resource(r)**, the behavior of resources at each time step in the main loop.

{Commit the most valuable request received}
1: **if** $exists(r.best\_request)$ **then**
2: $commit\_transfer(r, r.best\_request)$ **(alg. 8)**
3: $r.best\_request \leftarrow NULL$
4: **end if**
{Decay current bother level}
5: $update\_bother\_model(r)$
6: LET $a \leftarrow r.current\_task$
7: **if** $exists(a)$ **then**
8: $a.time\_remaining \leftarrow a.time\_remaining - 1$
9: **end if**
10: **if** $exists(a)$ **and** $a.time\_remaining = 0$ **then**
11: $r.current\_task \leftarrow NULL$
12: **end if**

Algorithm 8 An algorithm describing **commit_transfer(r,task)**, the behaviors of resources in response to a request for preemption.

LET $net\_EU \leftarrow compute\_preemption\_cost(r, task, 0)$ **(alg. 9)**
{Update our bother model.}
SET $r.BSF = r.BSF + bother\_increase\_function(net\_EU, r.BSF, \theta_r)$
LET *draw* be a random draw in $U(0, 1)$.
if $draw \geq P_{stop}(net\_EU, r.BSF)$ **or not** $exists(r.current\_task)$ **then**
  {Alert the resource's old task to the preemption}
  if $exists(r.current\_task)$ **then**
    SET $r.current\_task.current\_resource \leftarrow NULL$
    SET $r.current\_task.time\_remaining \leftarrow \infty$
    SET $r.current\_task.a.m_{congestion}.preemption \leftarrow TRUE$
  end if
  SET $task.current\_resource.current\_task \leftarrow NULL$
  SET $r.current\_task \leftarrow task$
  SET $task.current\_resource \leftarrow r$
  SET $task.time\_remaining = processing\_time(r, task)$
  return TRUE
else
  return FALSE
end if

preempted agent. Lower-priority agents will report lower eu_{plan} values (because of higher congestion experienced). This will in turn increase their chances of keeping their current resources. Within Algorithm 7, once any incoming preemption request has been handled, we reduce the resource's BSF value via exponential decay to reflect the resource forgetting about prior interruptions [9], and then update the remaining processing time for the task assigned to this resource, potentially setting the resource's state to "free" if the current task is completed.

Algorithm 9 An algorithm describing **compute_preemption_cost(r, t, tau)**, a resource proxy agent evaluating the total utility of a preemption request to assign r to t in tau timesteps.

if $exists(t.current\_resource)$ **and** $t.time\_remaining \leq \tau$ **then**
  {If t will be finished by the time the request is made, there is no gain.}
  return 0
end if
{Compute the expected processing times for requesting task t}
LET $pt_{r,t} \leftarrow processing\_time(r, t)$
if $exists(t.current\_resource)$ **then**
  LET $pt_{old,t} \leftarrow t.time\_remaining - \tau$
else
  LET $pt_{old,t} \leftarrow processing\_time(null, t)$
end if
LET $owner \leftarrow r.current\_task$
if $exists(owner)$ **or** $owner.time\_remaining \leq \tau$ **then**
  {The resource is unowned, so there is no preemption.}
  return $EU(pt_{old,t}, pt_{r,t}, t.\theta_t)$ {Benefit to requester}
else
  {The resource is owned, so the effects of preemption are calculated.}
  LET $pt_{left} = owner.time\_remaining$
  LET $pt_{reset} = processing\_time(r, owner)$
  LET $pt_{old,t} \leftarrow \min(pt_{old,t}, pt_{left} + pt_{r,t})$
  return $EU(pt_{old,t}, pt_{r,t}, t.\theta_t)$ {Benefit to requester}
  $+owner.agent.eu_{plan}$ {Value of backup plan of preempted task}
  $+EU(pt_{left}, pt_{r,t} + pt_{reset}, owner.\theta_t)$ {Loss due to preemption}
end if

3.6 Design Notes

The algorithms designed above bear several important differences from their progenitor system as described in [10]. These refinements reflect lessons learned from analysis of particular cases which exploit the model in ways that do not make sense with the underlying assumptions; several such cases arose when adapting this model from its assumed static task setting to an environment with dynamic task arrivals.

The most important difference is in how the resource proxy agent filters incoming requests for its resource. Without this intermediary step, it is entirely possible for a low-priority task to acquire a valuable resource due to lucky position in the turn order immediately after the valuable resource completes a task. While this is a good and obvious allocation if the valuable resource is not being contested, the immediate processing of resource requests creates the unfortunate scenario where a higher-priority task can make its request during the same tick, preempting the resource from the low-priority task before that task has had the chance to use its newfound resource. By only acting on resource requests after all such requests for the current tick have been finalized, these wasted preemptions (note that tasks give up their previous resource on acquiring a new resource!) are eliminated.

Removing the problem of spurious preemptions also helps a system by reducing how often its resources are bothered. Forcing resources to perform unnecessary work results in higher bother values which understandably inhibit the system from making legitimate requests of resources later. One quirk of the proposed bother model arises here, however: With a probabilistic rate of request acceptance, the system can technically be exploited by spamming a resource with requests (Algorithm 8) until a request is accepted. This defeats the purpose of the bother model in function and spirit, indicating that requests must at least be rate-limited (as they currently are via Algorithm 6) if resource limitations are to be respected.

The simple decision to skip the bother check when a resource is currently idle (Algorithm 8) helps the expressiveness of the system by permitting the simulation of an environment where preemption is costly or rare. Drawing a distinction between taking on a new task and attempting to preempt a task allows a system to do useful (if suboptimal) work despite high P_{stop} values. While this prevents the system from modelling total resource disengagement, this is also an intuitive improvement when BSF is used to model the intellectual effort of deciding whether to accept a preemption request, as helping a task is obviously better for the system than doing nothing (provided the net expected utility change is positive, otherwise the "helping" resource would provide less help than the task's current resource; this is why proxy agents remove negative-valued requests).

Note that every task agent always has a plan of which resources to approach, even if its task already possesses the optimum resource for its resolution. While a plan in this situation may be referred to as a contingency plan, there is no difference to the system aside from how the resource requests in the plan will have negative value unless the task loses its resource.

3.7 Analytic Evaluation of Optimum Case

Given the example domain model defined in Section 3.3.1, in particular the *EU* function based on each task penalizing the system as long as it is present, consider the cost incurred from resolving tasks under perfect conditions where all resources can resolve all tasks in a single tick and never drop requests. These conditions simplify the hard problem of weighing resolution time against severity, ensuring that the optimal allocation algorithm is to resolve the most severe tasks first. For sake of analysis, assume a uniform distribution of task types.

Under these simplified conditions, the total cost incurred by the system is thus equal to the cost incurred by each task before it is resolved, which can be solely determined by the number of tasks of its severity level or higher and the number of available resources. If tasks arrive over time at such a rate that the system always has capacity to treat each new arrival immediately, then the total cost will be $250.5n$ for n tasks on average due to the assumed uniform distribution of 500 task types (as the mean value of $1, 2, 3, \ldots, 500$ is 250.5). However, if enough tasks are present to build a backlog (i.e. a set of tasks that have failed to acquire or hold resources and thus remain in the system), then the increase in cost per task becomes dependent on the number of tasks. Assume now that n tasks are present in the system at the start, with no further tasks arriving after the system starts. Now, the net cost incurred by the system is determined by multiplying the number of each type of task by the task's type and the average time that tasks of that type will remain in the system. The availability of optimal resources makes the decision of allocation trivial, as it is always better to resolve a higher-value task than a lower-value task when resolution time is equal. Thus, a task will wait for all tasks of types higher than itself to finish (limited by the number of resources per tick). The net cost of a system with n tasks, r resources, and

500 evenly-distributed task types can thus be calculated:

$$\begin{aligned}
&\sum_{\theta_t} \left(\theta_t \times \frac{n}{500} \times (501 - \theta_t) \times \frac{n}{500} \times \frac{1}{r} \right) \\
&= \frac{n^2}{500^2 r} \sum_{\theta_t} \left(\theta_t \times (501 - \theta_t) \right) \\
&= \frac{n^2}{500^2 r} \times \left(501 \sum_{\theta_t} \theta_t - \sum_{\theta_t} \theta_t^2 \right) \\
&= \frac{n^2}{500^2 r} \times \left(501 \frac{(500)(501)}{2} - \frac{(500)(501)(1001)}{6} \right) \\
&\approx \frac{n^2}{500^2 r} \times \left(\frac{500^3}{2} - \frac{500^3}{3} \right) \\
&= \frac{n^2}{r} \times \frac{500}{6}
\end{aligned}$$

Thus, for a system with 100 perfect resources, in the limit the cost to the system increases by approximately $0.84n$ for each new task added.

In the case where resources are resolved in random order, so any given resource expects to wait $\frac{n}{2} \times \frac{1}{r}$ steps for resolution on average (half the number of tasks divided by the number of resources), the net cost of the system is:

$$\begin{aligned}
&\sum_{\theta_t} \left(\theta_t \times \frac{n}{500} \right) \times \frac{n}{2} \times \frac{1}{r} \\
&= \frac{n^2}{1000 r} \sum_{\theta_t} \left(\theta_t \right) \\
&= \frac{n^2}{1000 r} \times \left(\frac{(500)(501)}{2} \right) \\
&\approx \frac{n^2}{r} \times \frac{500}{4}
\end{aligned}$$

Thus, random allocation allows the system with 100 perfect resources to see a cost increase of approximately $1.25n$ per task added; compared to an increase of $0.84n$ for the optimal allocation order, this shows the value of a proper allocation process.

These values are lower bounds on the performance of a realistic system that cannot perfectly resolve tasks in minimal time. Note that once disparities exist in resolution time,

the allocation problem becomes far more difficult. A 251-severity task that can be resolved in a single step outweighs a 500-severity task that requires two steps, as delaying the 500-severity task by one step costs 500 utility while delaying the 251-severity task by two steps costs 502 utility. These comparisons rapidly become intractable as every resource has a different skill profile and the set of available tasks changes (whether through arrival or resolution). Distributing the analysis across task agents helps solve this problem in an efficient way.

3.8 Comparison to Existing Preemption Approach

Determining when preemption is valuable is difficult. In simple cases, the change in utility can be calculated by simple resource swaps between tasks, but this approach does not generalize well: It is easy to find preemption cycles, where the benefit gained through permitting a preemption depends on other preemption swaps which loop around.

Paulussen et al. [16] handle the cycle problem by simply marking resources that have already been requested. When calculating the side effects of requesting a resource, any tasks displaced by that request may themselves request resources to determine the net impact, with the requirement that no resource can be requested more than once until the entire chain is resolved.

Algorithm 10 describes an instantiation of this approach in the language of our system, which would be used in place of eu_{plan} in Algorithm 9 to determine the predicted cost of losing the task's current resource. This is an appealing algorithm, as it produces a logical exact cost of preemption by following the full chain of displaced tasks, though the last task in this chain may receive nothing if all resources are occupied. However, this is a conservative estimate, as the resource currently held by the requesting task is not considered. Note that to keep computation feasible, the backup resource considered is pre-calculated by a planning method which uses Algorithm 9 to determine the optimal resource; if the desired resource is reserved then the best non-reserved resource is not considered, as finding such a resource relies on the output of Algorithm 9 which relies on 10 in this framework.

Conservative estimates limit the potential gains from preemption. Initial allocations have no guarantee of optimality if there is any reasonable limit on the number of resources a task agent can approach, as certain resources may be highly contested. Aggressive reallocation allows a system to move towards an optimal state. Note that the congestion learning model initially assumes an idealistic system, allowing for more flexible preemptions near

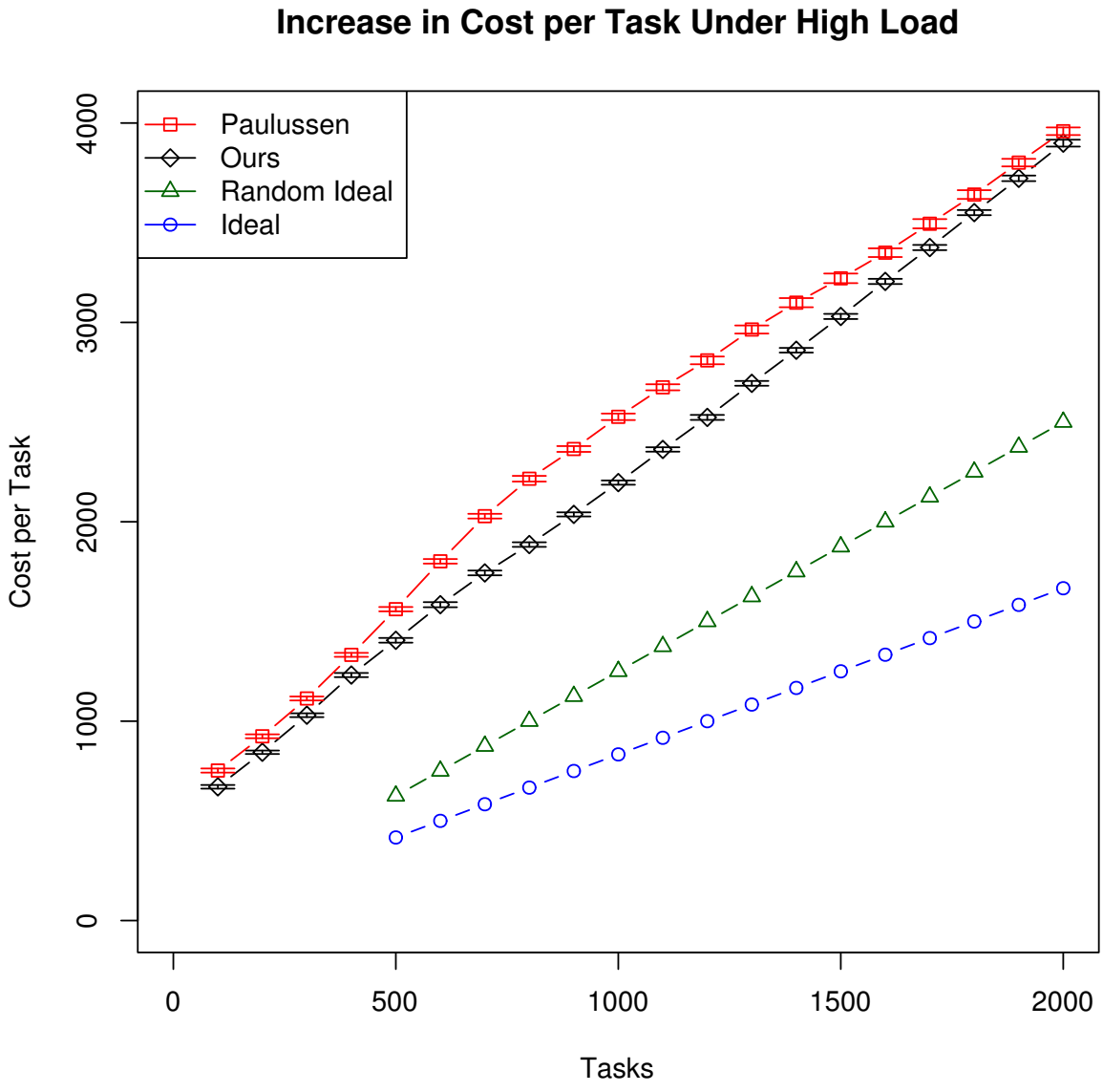

Figure 3.1: Cost per task when all tasks are present from start of system, compared against ideal conditions. (All error bars show a 95% confidence interval.)

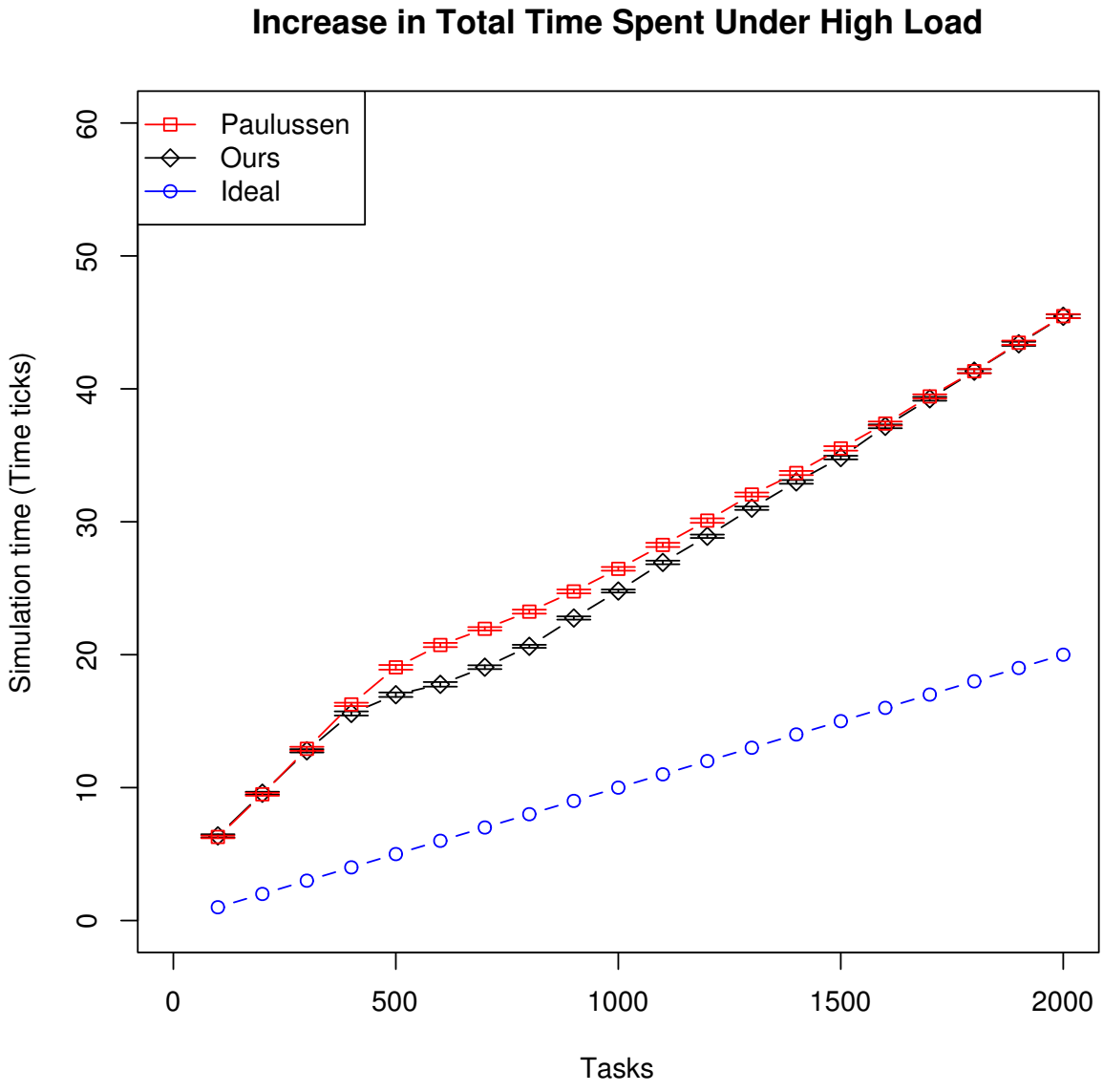

Figure 3.2: Simulation time when all tasks are present from start of system, compared against ideal conditions. (All error bars show a 95% confidence interval.)

Algorithm 10 An algorithm describing **paulussen_contingency_value(a)** which calculates the equivalent of eu_{plan} for a Paulussen-like system.

1: LET $utility \leftarrow 0$
2: LET *backup* be the preferred resource for t other than its current resource
3: **if not**$reserved(backup)$ **then**
4: $reserve(backup)$
{Calculate cost of switching to backup resource (contingency cost)}
5: SET $utility \leftarrow EU(processing\_time(backup, t), t.time\_remaining, t.\theta_t)$
6: **if** $exists(backup.current\_task$ **then**
{Recursively calculate contingency costs for displaced tasks}
7: SET $utility \leftarrow utility + paulussen\_contingency\_value(backup.current\_task)$
8: **end if**
9: $unreserve(backup)$
10: **else**
{If a cycle is reached, end the search assuming the last task gets nothing.}
11: SET $utility \leftarrow EU(processing\_time(NULL, t), t.time\_remaining, t.\theta_t)$
12: **end if**
13: **return** *utility*

the start of simulation when the system has had less time to find a reasonable allocation. Once preemptions start taking place, displaced task agents become less optimistic about the chances of securing new resources, helping them hold onto existing resources long enough to resolve tasks and keep system throughput up.

Figures 3.1 and 3.2 shows the relative performance of our system contrasted with a system using the Paulussen-like method of assessing preemption cost as described in Algorithm 10. The simulation used in this graph defines a maximum plan length of 4, and uses values of 1 for c and d and 0.95 for α and γ in the learning models for congestion and churn. Each point reflects the averaged results across 100 simulation runs, except for the "ideal" lines which are simply $\frac{5}{4}n$ for randomized completion and $\frac{5}{6}n$ for perfect completion in an environment with perfect resources, as computed above. The rate of increase in cost (loss of utility) per task is a relevant measure of how well the system handles heavy backlog of tasks (load); note that our system has a slope comparable to the rate of increase in idealized environments with perfect resources, while the Paulussen-like method climbs in cost steeply as system load increases.

Finding the right resources to resolve every task allows a system to approach the idealized case where all resources are perfectly suited to all tasks.

3.9 Dynamic Arrivals

Previous discussion has focused on a scenario where all tasks to be completed are present in the system from its initiation. This is a worst-case assumption under utility models where the total amount of time spent waiting is counted against the system, but the presence of all tasks reduces the complexity of the scheduling problem as all information is available from the start. When tasks can arrive over time, the system can make better use of its capacity by completing available tasks before new tasks demand attention, but preemption becomes much more important as new high-priority tasks can always arrive.

The distributed design of our algorithms is highly conducive to handling dynamic arrivals. While all task agents need to know about all resource proxy agents in order to effectively choose the best resource to approach, there is no need for resources to track tasks; resource proxy agents only need to use the information about tasks provided by those tasks that approach them. As a result, it is easy to add new task agents to the system as if they had always been present. A centralized brute-force allocation algorithm might be able to find an optimal allocation at great computational expense if all tasks are present from the beginning, but the need to recompute the optimal allocation every time a new task arrives makes such an approach problematic.

Further simulations were conducted, distributing the task arrivals evenly over the first 50 ticks of simulation. With 100 resources in the system, this means that the system is far under capacity for the majority of operation. Figure 3.3 clearly shows how spreading out the same number of tasks not only reduces the net cost (axes are not to scale with Figure 3.1) but also the rate at which this cost increases as the system is loaded. Figure 3.4 similarly shows that the primary limit on the simulation runtime is the arrival of the last few tasks.

Note that a cost disparity still exists between our solution and the Paulussen-like approach, despite the latter arguably providing a more accurate evaluation of the costs of preemption. The learned rate of flexibility in allowing preemption still wins out in this environment of dynamic task arrivals, with new arrivals cautiously optimistic about the chances of securing useful resources. Further, the total completion time required for both algorithms is essentially identical (due to the arrivals being spaced out over time), meaning that this cost improvement is purely from better allocations in the time available.

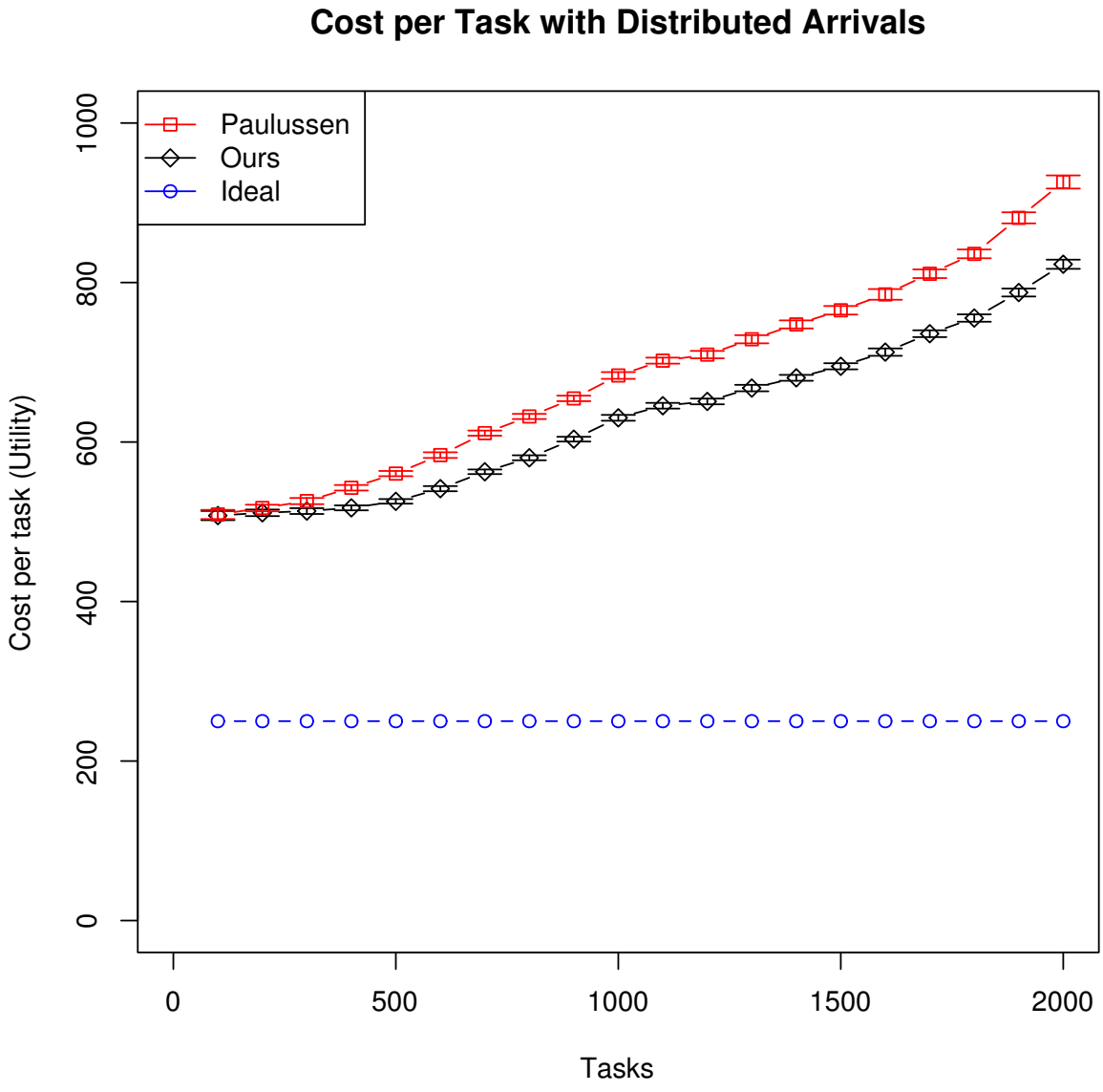

Figure 3.3: Cost per task when tasks arrive evenly distributed over the first 50 ticks of simulation. (All error bars show a 95% confidence interval.)

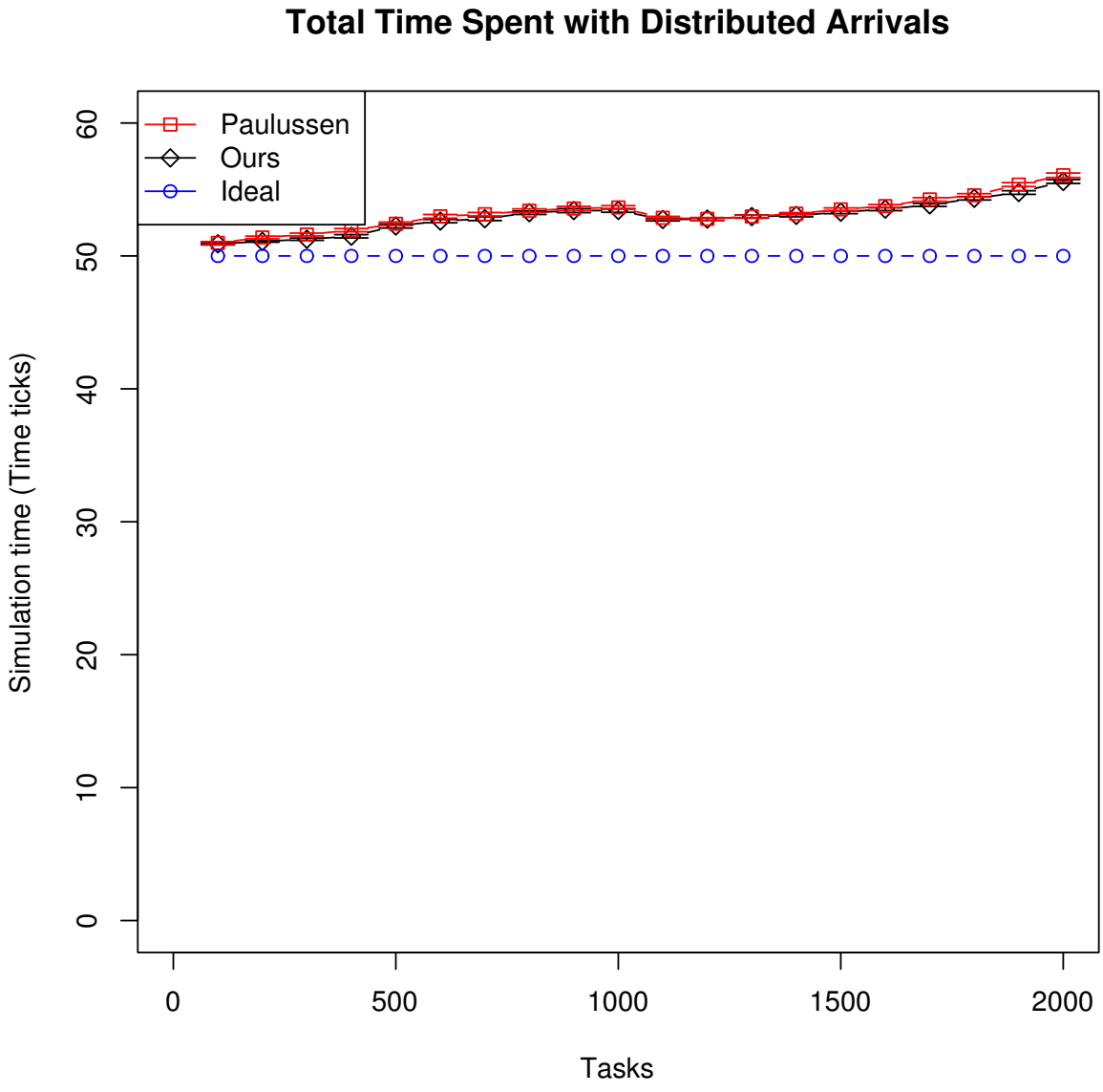

Figure 3.4: Simulation time when tasks arrive evenly distributed over the first 50 ticks of simulation. (All error bars show a 95% confidence interval.)

Chapter 4

Opportunity Cost and the Value of Waiting

4.1 Introduction

If there is a cost associated with reassigning resources, then there may be times where it is more efficient to keep a resource idle rather than locking it into an allocation where it provides less benefit than it might be able to provide in the future. However, the higher the cost, the easier it is to find situations where waiting can help.

Opportunity cost is the drop in utility due to losing the ability to perform an action; for example, consider a case where you have $10, you can buy coffee for $1 to gain 1 utility, and tomorrow you can buy a movie ticket for $10 to gain 20 utility. (Assume that having extra money left over has no value.) Buying coffee now gives more utility than not buying coffee, but it comes with the opportunity cost of 20 utility because you won't be able to afford the movie ticket later. The correct decision in this case is to keep money in reserve in order to take advantage of the future payoff. Note that opportunity cost is highly dependent on the expected changes in environment, so potential tasks which can use the reserved resource should feel free to challenge the decision to reserve a resource under the hope that the opportunity cost has lowered. For example, if you find out that other plans will prevent you from seeing the movie tomorrow, then the opportunity cost drops from 20 utility to 0 utility and so there is no need to conserve money now when you could use it to gain utility.

The opportunity cost in a preemption environment is usually smaller, but still exists. In the example above, imagine if you could borrow $1 from a friend at the cost of 2 utility

due to the headache of tracking debt. In this case, the opportunity cost of buying coffee now drops from 20 utility lost to 2 utility lost, because you can still see the movie by borrowing money. The opportunity cost is still higher than the utility gain from buying coffee, but if there is some probability that you won't be able to see the movie even if you can pay for it then not buying coffee now could be a wasted opportunity. With risk-neutral preferences, you should buy coffee if there is a 50% or higher chance of missing the movie if you can borrow money, compared to only getting coffee at a 95% or higher chance of missing the movie if you can't borrow money.

4.2 When Waiting is Valuable

Waiting is an expensive operation for resources in a rate-limited resource allocation environment, as cutting down on the amount of time that a resource is in use effectively reduces the pool of resources available, slowing the completion of all tasks currently in the system. However, there are still edge cases where hasty allocations can similarly waste resource capacity.

Consider the utility function from earlier examples, where each task applies a constant cost to the system for every time step that the task remains in the system. Each time step that a resource waits thus applies a potential cost to the system (i.e. a loss of utility from delaying a task) equal to the highest single step cost among available tasks, as the resource could have been applied towards finishing that task one step faster. This captures how there is no wait cost when there are already enough resources to handle all existing tasks. However, this is an approximation, as once the resource has waited for as many steps as there are resources, then the last step of the system (when the last tasks are resolved) is delayed by a step, raising the net cost to the system by the combined total cost of all tasks still present in that last step. Similarly, if the resource would have been preempted before finishing work on the currently-available task, then there is no cost from waiting.

For a concrete example: Consider a computing cluster with two servers (resources) available, r_1 and r_2. A task t_1 already exists in the cluster with a severity of 200, and in one tick a new task t_2 with severity 401 will enter the cluster. Let r_1 be a standard server that can complete either task in 6 ticks, while r_2 is a high-end server that can complete either task in 3 ticks. Before t_2 enters the cluster, it appears strictly better to assign t_1 to the high-powered r_2 to save on 3 ticks of processing time at 200 utility lost per tick. However, if the cluster makes this seemingly reasonable allocation, then one tick later t_2 enters the cluster and has the severity to pre-empt r_2 from t_1. This follows from how delaying the completion of t_1 by 4 ticks (instead of being completed in 2 more ticks

it would be completed in 6, whether by running on r_1 or by waiting for r_2 to complete t_2 and restarting on r_2) costs 804 utility while delaying the completion of t_2 by 2 ticks (waiting for t_1 to complete in 2 ticks before starting on r_2 rather than pre-empting r_2 and starting immediately) costs 802 utility. The first tick of computation spent by r_2 on t_1 is thus wasted. However, if the powerful r_2 had been reserved for t_2 in that first tick by allocating r_1 to t_1 instead, then t_1 would be completed in a total of 6 ticks (instead of 7 ticks, the first of which was wasted due to preemption) while t_2 would still be completed in 3 ticks, for a total savings of 200 utility. If there was an unacceptably high penalty to stopping a task in progress, then without reserving the powerful r_2 then the less-severe t_1 is completed in 3 ticks while the more-severe t_2 is completed in 5 ticks (for a total cost of $3 * 200 + 5 * 401 = 2605$ utility), as opposed to t_1 being completed in 6 ticks and t_2 being completed in 3 ticks (for a total cost of $6 * 200 + 3 * 401 = 2403$ utility, saving 202 utility).

Consider a case with only one resource and generalized tasks. Take two tasks:

- Task t_1, which costs x utility per step and will require y steps to finish, and exists in the system right now.
- Task t_2, which costs a utility per step and will require b steps to finish, and will enter the system in c steps.

The net cost for finishing the first task first thus comes to $x \times y + a \times (b + y - c)$, while the cost for waiting for the second task comes to $x \times (y + b + c) + a \times b$. In order for waiting to be worthwhile, the following relation must therefore hold:

$$\begin{aligned} x \times y + a \times (b + y - c) &> x \times (y + b + c) + a \times b \\ a \times (y - c) &> x \times (b + c) \\ \frac{a}{x} &> \frac{b + c}{y - c} \end{aligned}$$

As expected, if t_1 could be completed before t_2 arrives (if $y < c$) then there is no reason not to finish t_1 first. For tasks that take similar amounts of time to complete, t_2 must have higher severity than t_1; taking the example above where either task could be completed in 3 ticks and t_2 is delayed by 1 tick, t_2 must have higher severity by a factor of $\frac{3+1}{3-1} = 2$ to be worth waiting for.

As tasks take longer amounts of time to complete, the late-arriving task needs less of an advantage to be worth delaying the currently-available task. In particular, a currently-available task which requires many steps to finish anyway may be worth putting on hold to quickly complete new tasks that require less time.

The analysis becomes more difficult when there are multiple resources involved, as different resource assignments must be considered and the specifics of future arrivals become important. Suppose now that there are resources r_1 and r_2, and adjust the tasks from before:

- Task t_1, which costs x utility per step and will require y_1 steps to finish with r_1 or y_2 steps to finish with r_2, and exists in the system right now.
- Task t_2, which costs a utility per step and will require b_1 steps to finish with r_1 or b_2 steps to finish with r_2, and will enter the system in c steps.

Suppose that r_1 is a better resource than r_2 (thus $y_1 < y_2$ and $b_1 < b_2$). Also assume that r_1 is not so much better than r_2 that the best case is always to have r_1 complete both tasks, as that analysis is shown above. Then there are three possible outcomes:

1. t_1 is finished by r_1 and t_2 is finished by r_2, at cost $(x \times y_1) + (a \times b_2)$
2. t_1 is assigned r_1 which is then taken by t_2 which finishes while t_1 is finished by r_2, at cost $(x \times (c + y_2)) + (a \times b_1)$
3. t_1 is finished by r_2 and t_2 is finished by r_1, at cost $(x \times y_2) + (a \times b_1)$

The middle outcome is strictly worse than the last outcome where the correct final allocation is known. Optimal allocation thus gives the system $x \times c$ utility every time that the last outcome is better than the first outcome. Longer lookahead improves resource reservation decisions (capped by the processing time of tasks using the best resource), and this benefit is dependent on the priority of the less important task - if some tasks are of negligible importance to the system, then it is easiest to simply preempt resources from those tasks when necessary.

Thus, when preemption is prevalent, reserving resources is not for the benefit of the tasks that receive the reserved resources. Those tasks could have preempted the resource even if it had not been reserved. Reservation instead improves utility for tasks that are warned away from taking resources that they will not be able to keep.

4.3 Dummy Agent Approach

The value of a wait action can be provided in a task-centered system by dummy task agents, which are agents representing a blank 'dummy' task which corresponds to waiting

for a real task that will enter the system later. Unlike other tasks this dummy task can always be interrupted (even in environments where preemption is difficult or impossible), as a resource 'performing' this task is understood to be on standby until a sufficiently important task interrupts it. Once the real task in question arrives in the system, the dummy agent grants its resource to that task and seeks a new incoming task to represent.

A dummy agent competes with other task agents for resources by reporting the utility of waiting, measured by examining the opportunity cost of allocating the resource now. If the utility of waiting is higher than the utility gain from allocating the resource to any other task that approached it, then the resource is allocated to the dummy agent instead of working on a real task; by definition, the cost of allocating the resource is higher than the cost of leaving it idle in this case. By reusing the concept of a task agent, the dummy agent seamlessly provides a wait action to resources as an alternative to other available tasks. Note that the opportunity cost of waiting increases sharply when more than one resource waits, as the system would need to expect multiple tasks to enter in a short span of time (otherwise the time spent waiting could be used to complete other tasks) in order to use multiple reserved resources. For this reason, the rest of this thesis assumes that only one resource will wait at a time; possible extensions are given in Section 6.2. If every resource evaluates the cost of waiting and concludes that it is in the system's best interests for one resource to wait, then resources will need to coordinate to determine which resource waits. In simplistic solutions using local information exclusively, every resource might decide to wait, incurring heavy real costs as available work is ignored. The use of a dummy agent allows resources to implicitly coordinate with each other on when to wait, as only the resource approached by the dummy agent will wait. The dummy agent however can reuse the existing planning mechanism that task agents use to select resources in order to choose a good resource to wait for impending tasks.

It is expensive for a system to not work at full capacity, as every tick spent waiting for the arrival of a task is a tick not spent clearing the tasks that already exist in the system. For waiting to be worth the trouble, truly critical tasks must arrive, where the time savings offered by a dummy agent for these critical tasks are enough to outweigh the loss of processing capacity.

In a proof-of-concept implementation, opportunity cost was simulated by asking an oracle for the highest-priority task that will arrive in the next time step and granting that priority to the dummy task. This is an idealistic level of information, though it may be justified in scenarios where tasks are discovered before resources can be assigned to them (e.g. a computing example where tasks must have large amounts of data uploaded to a cluster before computation can start, or a medical example where ambulance drivers can call ahead to the hospital to alert them to incoming patients, so the tasks are known before

they start). Even using perfect knowledge, this simple method slightly overestimates the opportunity cost, as the current tick may still be usable; if the dummy task prevents a slightly-lower priority task from getting an early start which is worth making the slightly-higher priority task wait, then there is utility loss. For example, consider a system where a task that costs 90 utility per tick is already present, and next tick a new task will arrive that costs 100 utility per tick. If the only available resource can finish either task in three ticks, then if it started on the 90/tick task now and only addressed the 100/tick task after finishing with the 90/tick task, then the total cost will be $90*3+100*5 = 270+500 = 770$ utility. However, if the resource waits for one tick to finish the 100/tick task as soon as possible, and no other resources are available to resolve the 90/tick task until after the 100/tick task is complete, then the total cost will be $90 * 6 + 100 * 3 = 540 + 300 = 840$ utility which is worse. With that said, in a system with perfect lookahead this corner case is the only way for the dummy agent to negatively impact the system. In order to see improvement from waiting, the ratio of task importance must be higher as the task completion time decreases, as the single missed step becomes more important when it can complete more of the currently-available task.

A more realistic implementation would not have perfect knowledge of incoming tasks. This could be an issue for resources with disparate compatibilities, as choosing the correct resource to wait becomes nontrivial: Even if the distribution of incoming task types is known, if no one resource is compatible with all high-probability task types, it is difficult for the dummy agent to choose a resource to approach. For example, consider a case of 10 task types, with larger task types having higher priority. If there is a high probability of a new task of type 7 arriving in the next step, then the dummy agent would approach a resource compatible with type 7 tasks. However, if all resources compatible with type 7 tasks are incompatible with type 8 tasks, and the distribution gives an equal high chance of type 7 or type 8 tasks arriving, then the dummy agent must compromise. While this may sound strange in the abstract, consider a medical example where type 7 tasks are patients with broken bones and type 8 tasks are patients with internal injuries; if resources in this case are doctors, then finding specialists in both injuries may be difficult. Our implementation does not address this problem, instead simply choosing a resource to reserve based on which resource is best suited for the most important expected incoming task.

4.4 Practical Improvement

In order to indicate the importance of waiting, a resource-constrained simulation environment was used with dynamic task arrivals over time. When there are only a handful of

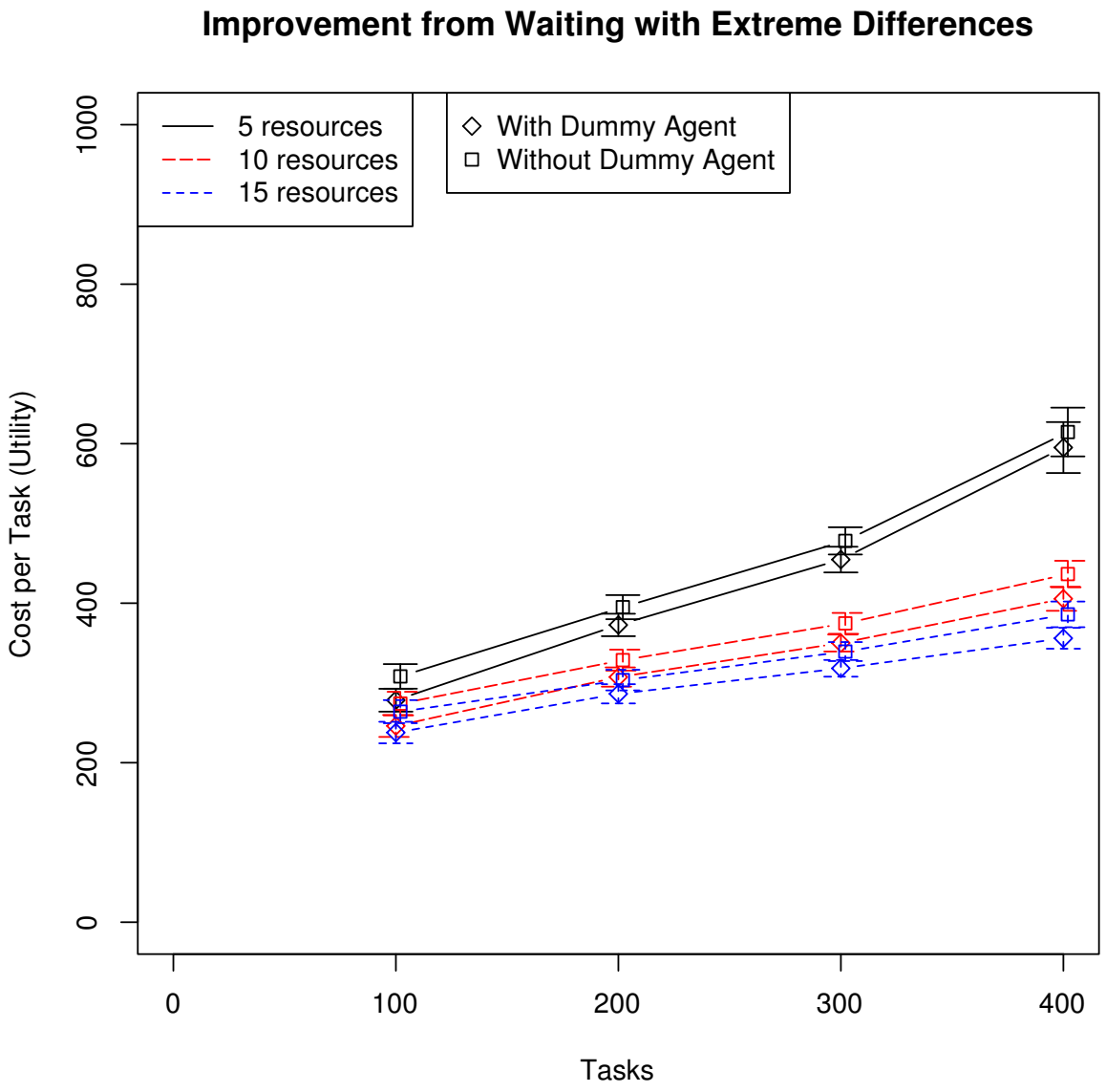

Figure 4.1: Benefits of using a dummy agent plotted against number of agents that enter the system (evenly distributed over the first 50 ticks, 90% low-severity and 10% high-severity tasks). Smaller values are good. (All error bars show a 95% confidence interval.)

resources available to process tasks, the impact on the system from having a single resource wait is relatively large and thus more likely to illustrate the benefits and failings of this approach. To further illustrate the difference, an alternate distribution was used: Tasks were either the minimum (1) or maximum (500) severity, with 90% of tasks at minimum severity. One resource could complete a task in 4 ticks, while all other resources required 6 ticks to complete a task; the expected lower bound on cost per task would thus be $500 * 4 * 0.1 = 200$ if minimum-severity tasks are ignored and all maximum-severity tasks are completed on the 4-tick resource without delay. The dummy agent was granted 5 steps of perfect lookahead, and behaved as any other agent would when representing a task with severity equal to the highest severity in the lookahead window. This reduced scenario is ideal for a dummy agent, as there is a single preferential resource which will not always be occupied with maximum-severity tasks, and the minimum-severity tasks do not overly suffer from delay.

Figure 4.1 shows the change in expected utility in systems with 5, 10, and 15 resources between using a dummy agent or not. Under these idealized conditions, adding a dummy agent allows the system to use its resources more efficiently, though not statistically significantly so.

Unfortunately, it is the huge disparity in task priorities which allows this improvement in efficiency. Figure 4.2 shows the results of a similar simulation, differing only in the severities of incoming tasks: While 10% of tasks were still maximum (500) severity, the remaining 90% tasks were at medium (250) severity instead of minimum (1) severity. Once the baseline work available to the system is significant, the loss of system capacity due to the dummy agent accumulates over time and drives down system efficiency as the backlog of meaningful tasks grows. Similar results can be seen in Figure 4.3 which uses a uniform distribution of task severities from 1 to 500. The distribution of available tasks can entirely decide the efficacy of this dummy agent.

A huge disparity in task priorities is necessary but not sufficient; the value of the dummy agent arises from the ability to respond to *rare* high-severity events. Figure 4.4 shows a simulation where tasks are evenly split between maximum (500) and minimum (1) severity rather than making maximum-severity tasks rare. A backlog of high-severity tasks leaves no available work for a dummy agent, as the high-severity tasks already in the system are assured to immediately receive the resources which finish performing work on other high-severity tasks. The resource constraint here works against the efficacy of the dummy agent, as the dummy agent can only provide value through reserving resources if there are resources available to reserve. When all resources are occupied by tasks of the same importance that the dummy agent represents, then the dummy agent cannot provide any benefit.

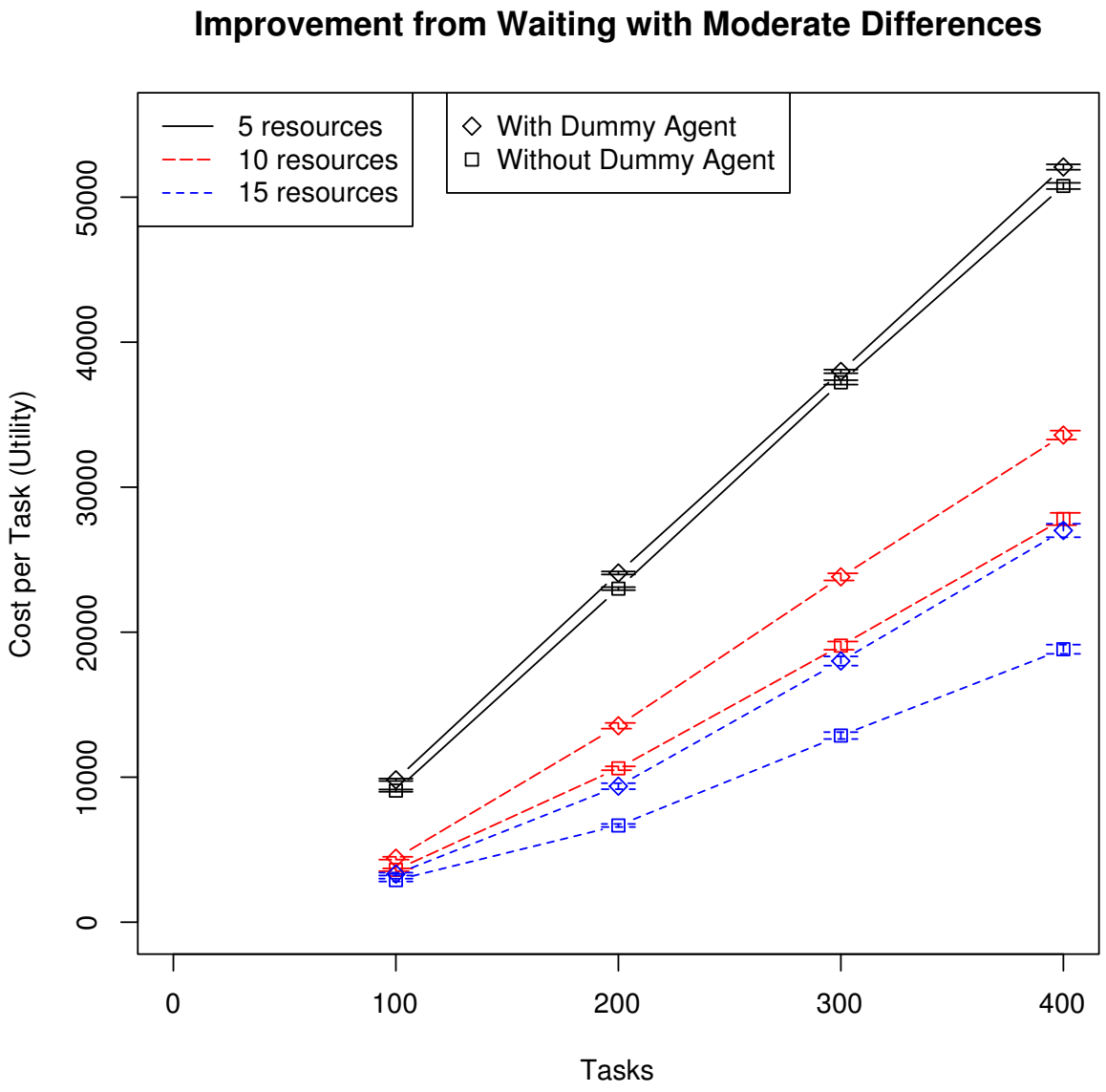

Figure 4.2: Benefits of using a dummy agent plotted against number of agents that enter the system (evenly distributed over the first 50 ticks, 90% medium-severity and 10% high-severity tasks). Smaller values are good. (All error bars show a 95% confidence interval.)

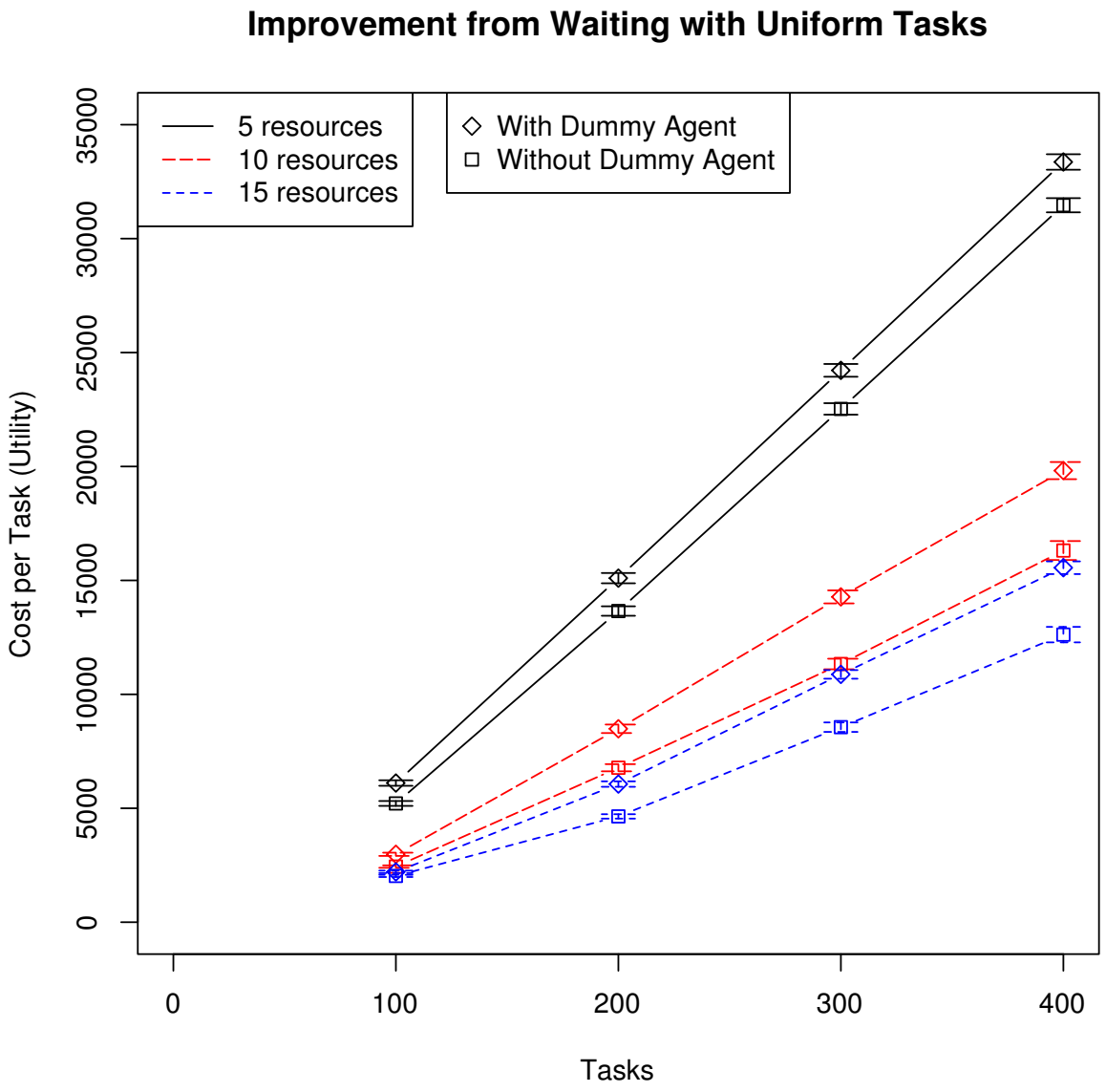

Figure 4.3: Benefits of using a dummy agent plotted against number of agents that enter the system (evenly distributed over the first 50 ticks, tasks with evenly distributed severity). Smaller values are good. (All error bars show a 95% confidence interval.)

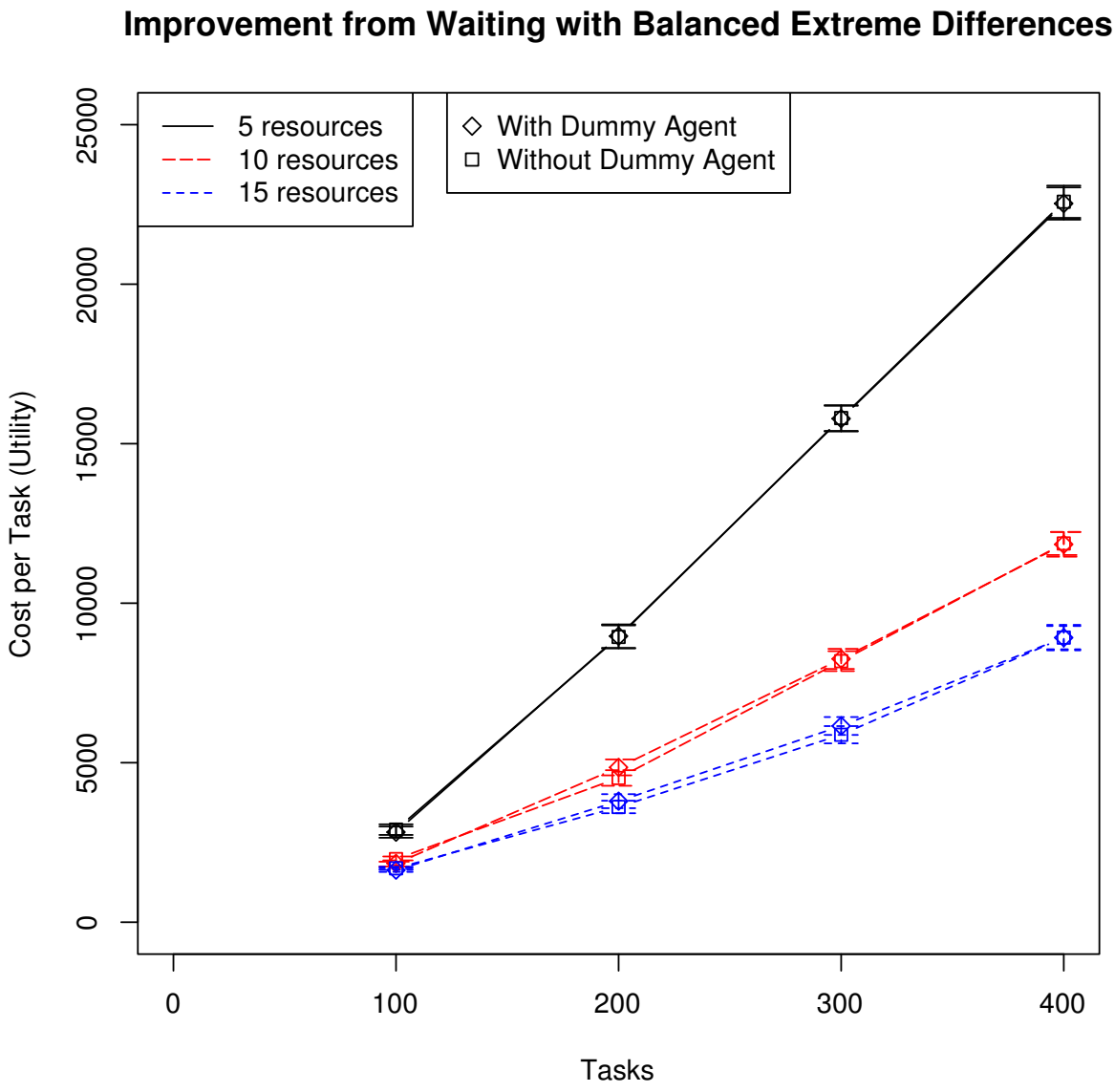

Figure 4.4: Benefits of using a dummy agent plotted against number of agents that enter the system (evenly distributed over the first 50 ticks, 50% low-severity and 50% high-severity tasks). Smaller values are good. (All error bars show a 95% confidence interval.)

While there is reason to believe that the system will benefit from allowing resources to wait for more critical tasks, the practical benefit is difficult to provide when the environment permits preemption. The ability to revoke resources from tasks in progress reduces the opportunity cost to such a degree that only very specific scenarios permit improvement. Preemption is a valuable tool which should be exploited when available, but not all real-world scenarios permit preemption. To explore the full potential of the dummy agent, we will need to examine cases where preemption is difficult or impossible, resulting in irreversable allocations.

Chapter 5

Analysis of Irreversable Allocation

5.1 Introduction

As described previously, easy access to preemption greatly softens the impact of any mistakes made when allocating resources, as a task must always yield its resource to another task of sufficiently greater need. However, not all problem domains allow for such easy reallocation. Consider the scheduling of underground construction for a water company, a problem domain where tasks involve tearing up sections of roadway and shutting off water flow in order to perform maintenance. While a resource here is a construction crew, work cannot be interrupted in the middle of replacing a section of underground water pipe, as the interruption in service is unacceptable. Even if a pipe were to break elsewhere, taking the construction crew away from the job in progress would leave just as bad of an emergency at the unfinished job site. In these domains where allocation is irreversable (until the task is complete), there is far less leniency towards bad initial allocations that ignore new incidents.

In operating systems, a computational task that requires exclusive use of a resource and cannot be interrupted by others is referred to as an atomic task. In a general-purpose computing environment, arbitrary computational tasks may depend on each other in ways that make it dangerous to preempt complex resources (e.g. printers, communication channels) from tasks in progress. While modern multi-tasking allows arbitrary processes to be paused and resumed with no loss of state, external resources may carry their own associated state which is not automatically managed by multi-tasking. This is an analogous case of irreversable allocation, and problems such as priority inversion (where a low-priority task holds a resource required by a high-priority task while a medium-priority task uses

all the computational time and prevents the low-priority task from finishing) and deadlock (two tasks each need two resources to finish, each task holds one resource, and neither task is willing to abandon its resource, thus neither task can finish) arise in computational environments from poor allocation.

While the system described thus far cannot suffer from priority inversion or deadlock as each task is assumed to require only one resource, high-priority tasks can still be forced to accept low-quality resources if they arrive at a time when all the suitable resources are already allocated. Long-term plans can be valuable in such situations, as a perfect resource which will be available shortly may be preferable over a terrible resource available now, at least for the task that actually receives the perfect resource out of all those competing for it.

5.2 Switching Costs

The applicability of the dummy agent depends on whether the environment permits preemption. While not all domains permit perfect preemption, preemption is not a binary feature of an environment; reallocating resources between tasks in progress can be expensive without being impossible, and this can be modelled through switching costs.

In computer scheduling, switching costs typically refer to the cost associated with recording the current state of a process and loading the state in a stored process in order to switch execution from the first process to the second. Psychology also uses the term switching cost for the cognitive effort required for a human to stop performing one task and start another, with distractions causing undue effort as they cause subjects to lose their place in an ongoing task.

Resetting task progress whenever a preemption occurs (as assumed in the model described in Chapter 3) serves as a kind of switching cost, though as delays have different utility impact depending on task severity, the magnitude of this cost varies greatly. Actual difficulty in switching between tasks can be modelled as an additional cost; making such a cost independent of task type allows for analysis of a gradient of problems between a preemption environment and an environment where all allocations are irreversable.

The algorithms in chapter 3 cannot natively represent these switching costs, as they are independent of the available variables involved in the utility formula (old completion time, new completion time, and task type). However, switching costs can be implemented with a simple adjustment to Algorithm 9 where the baseline utility for preemption requests is calculated; applying a switching-cost penalty here to any case where a task leaves a

resource will adjust the behaviour of the model to reflect these switching costs. The only other adjustment required is for evaluation purposes, as whatever mechanism used to record the total utility cost incurred by the system must correctly apply switching costs.

Consider Algorithm 5 which simply examines the expected cost of preemption and approves the preemption request when positive. Every additional cost placed on preemption will reduce the number of cases where this value is positive, making the system more conservative with changing allocations. As the cost increases, eventually the switching cost exceeds any possible benefit from preemption, at which point allocations are irreversable.

5.3 Changes in Allocation Strategy

When resource allocation is irreversable, there is value in spending more time to work out a good allocation. Environments that permit preemption allow suboptimal allocations to improve through a sequence of trades, encouraging the quick selection of a simple reasonable allocation which can be improved upon later. Opportunity cost becomes far more significant, as a system operating at full capacity (often necessary to maximize utility) is also incapable of responding quickly to new tasks. Preemption allows all currently-working resources to act as a reserve for emergency arrivals, so the lack of preemption makes the idea of reserve resources tenable again.

Our learning model for congestion is not meaningful in an irreversable environment, as to be expected since its learning events rely on preemption. The congestion model only modifies the backup plan value for use in approving preemption, so when preemption is removed this model cannot contribute. However, the model for churn is still valuable, as the ability to look several steps into the future is far more valuable when resources can only be claimed when idle. Asking resource proxy agents when a resource will be free again helps create good plans, but when this information is out of date (for example, if multiple tasks are implicitly queueing up to take a resource once it finishes its current task) the churn model allows a task agent to recognize when it is better to plan myopically and take whatever resources are currently available.

Bother modelling is no longer relevant in an irreversable environment (provided that idle resources are always willing to work), though the bother model can be used to bridge the difference between preemption and irreversable environments. Using P_{stop} functions that have high rates of rejection will make allocations more difficult to interrupt, encouraging similar behaviour as in a fully irreversable environment.

5.4 Revisiting Dummy Agents

In a system allowing preemption, there is some inherent inefficiency to the use of a dummy agent to reserve resources. A resource allocated to a dummy agent is incapable of performing useful work for the system, restricting the potential benefit; of course, as long as the dummy agent prevents low-priority tasks from wasting time by acquiring the resource and then losing it before completion, then there is a chance of benefit. Additional costs to preemption make this benefit easier to achieve, as preventing these switching costs justifies losing more work. However, this dynamic changes abruptly once allocations are irrevocable. As the dummy agent does not represent a real task, a resource can be allocated to the dummy agent without locking that resource to a task. This means that important tasks can immediately acquire a resource held in reserve by the dummy agent. If the resource had instead been allocated to a low-priority task, then an important task which arrives later is completely unable to access the resource until that low-priority task has completed.

To construct a scenario where the dummy agent is likely to be important, assume the following: tasks require a sizable investment of time to complete, independent of their severity; critically-important tasks are rare enough to not usually create a backlog; and dummy agents have enough advance warning to reserve a slot if necessary. Figure 5.1 shows the results of a simulation run under these assumptions where every preemption carries an additional cost of 500 utility, thus requiring severity-500 tasks to provide some proof of benefit in order to preempt severity-1 tasks. Figure 5.2 shows a similar simulation with a switching cost of 1000 utility, making preemption even more difficult. Figure 5.3 shows the results of such a simulation where preemption is impossible. As expected, a single dummy agent makes more of a difference when there are fewer resources in the system, as it has the capability to reserve a larger fraction of the available resources. The impact is also much greater when preemption is impossible than when preemption is merely costly, and higher costs increase the potential benefit; while the 500-utility switching cost does not show a statistically significant difference between using a dummy agent or not, the 1000-utility switching cost creates just enough of a difference to be significant, and the irrevocable scenario shows a clear improvement.

The assumption that all tasks require significant portions of time to complete is fairly reasonable. In many domains there are "maintenance" tasks that require a nontrivial amount of work and must be performed eventually but rarely pose a critical threat to performance. In a computing domain, disk defragmentation and system updates fit this role; in medical domains, consider cosmetic and reconstructive surgery as examples of uninterruptable low-priority tasks.

In domains where critical tasks are common enough to create a backlog, there is less

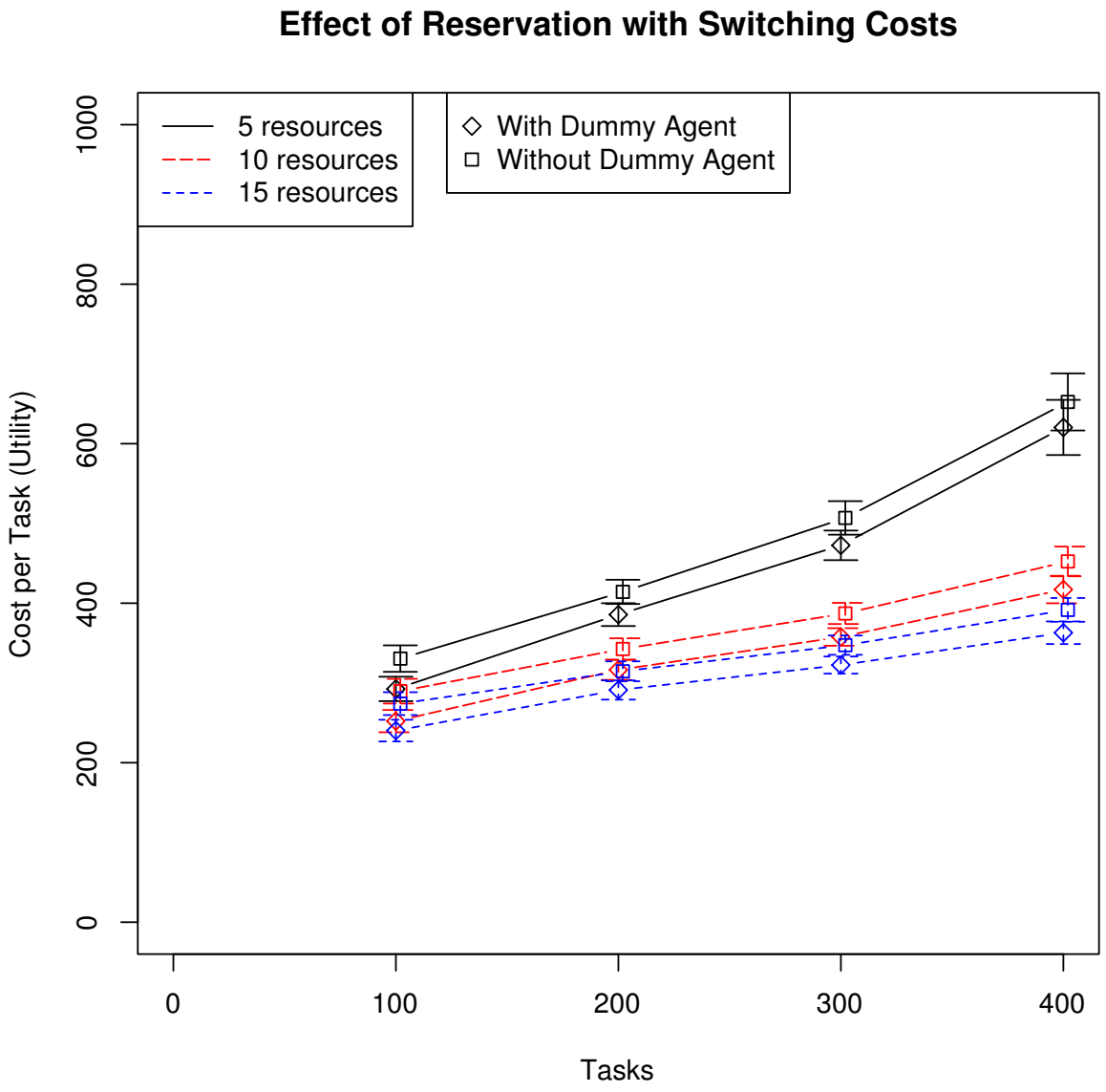

Figure 5.1: Benefits of using a dummy agent plotted against number of agents that enter the system (evenly distributed over the first 50 ticks). Smaller values are good. (All error bars show a 95% confidence interval.)

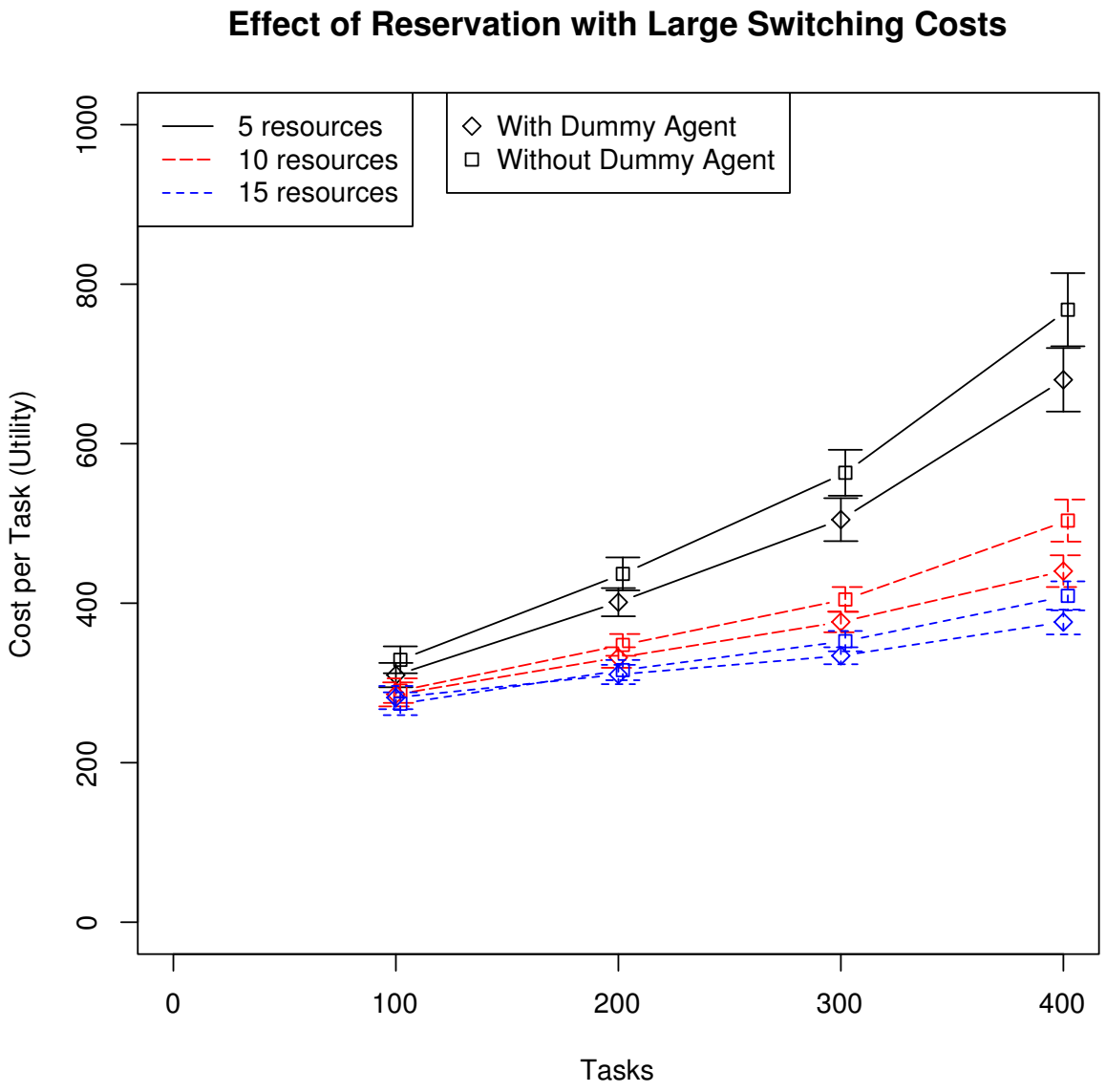

Figure 5.2: Benefits of using a dummy agent plotted against number of agents that enter the system (evenly distributed over the first 50 ticks). Smaller values are good. (All error bars show a 95% confidence interval.)

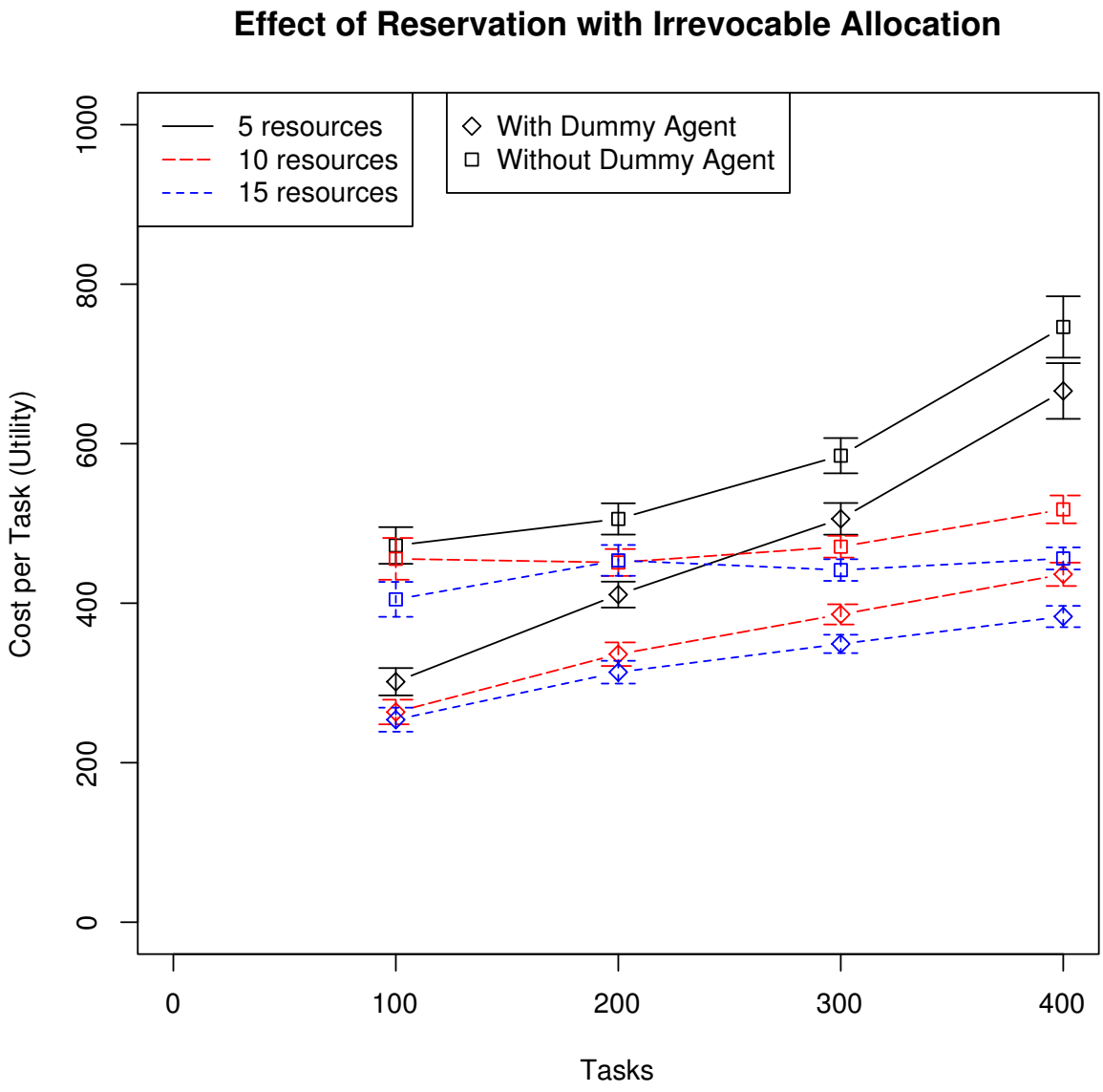

Figure 5.3: Benefits of using a dummy agent plotted against number of agents that enter the system (evenly distributed over the first 50 ticks). Smaller values are good. (All error bars show a 95% confidence interval.)

value to be gained from reserving resources as the critical tasks will naturally acquire any resources that complete prior tasks. If a block of time is spent completing critical tasks back-to-back, the only improvement granted by the dummy agent exists at the beginning of this block where critical resources begin completion slightly earlier than they would otherwise. However, if critical tasks are rare then without a dummy agent it will be far more common to have all resources tied up with non-critical tasks whenever a critical task arrives, increasing the proportion of time spent waiting for non-critical tasks to complete. Removing this initial wait cost on multiple occasions produces the improvements visible in Figure 5.3.

Naturally, reserving resources is only useful when there is a reserved resource ready for a newly-arrived critical task. This is where knowledge of future arrivals can be useful.

Chapter 6

Conclusions

6.1 Related Work

There have been many different resource allocation approaches which make use of multi-agent systems. The problem domain of interest helps inform decisions about how the parts of a system interact, resulting in slight differences between these approaches. The method suggested by Paulussen et al. [16] was used as a point of comparison due to its similar origins as a multi-agent resource allocation approach in the medical domain, where tasks are patients, resources are doctors who cannot be shared among multiple patients at once, and honesty is assumed when determining patient priority. However, even with this shared baseline understanding, there are key differences in the solution assumptions.

It is difficult in general to compare the different multi-agent resource allocation approaches used in the literature, as there is a wide variety of simplifying assumptions. If every approach picks a specific scenario to focus on while simplifying away the scenarios which other approaches address in detail, then any attempt to compare two systems must decide how to expand an approach to support the expanded model of the other approach. For example, an ambulance allocation algorithm incorporating spatial task information while ignoring other differences between ambulances will be difficult to compare to a task-resource matching algorithm which permits tasks to finish faster by using multiple resources but constrains each task to only be able to benefit from a small handful of resources. Each algorithm may be implemented through a multi-agent system, and share strong thematic ties, but they address largely different problems. Each of these potential solutions thus poses a potential goal for expansion in order to address problems in the domain of interest.

Combining two of these solutions will result in novel work if the simplifying assumptions of the two differing algorithms can be reconciled.

When selecting a multi-agent resource allocation approach, identifying the key features of your problem is essential in order to find an algorithm which makes the best use of these features.

6.1.1 Medical Path Agents

The Medical Path Agents (MedPAge) system presented by Paulussen et al. [16, 17] operates under different domain assumptions than the instantiation of its Medical Path Coordination (MedPaCo) mechanism used as a comparison in Chapter 3. Each agent in this system is responsible for a set of associated tasks representing different stages in a patient's treatment, where these tasks have natural scheduling constraints (a patient must undergo specific stages in order, e.g. they must receive an x-ray before surgery). The resources in this system represent medical equipment, but the unit of ownership consists of schedule slots rather than a focus on immediate ownership.

The negotiation process itself seeks exact evaluations of utility change when considering whether to permit preemption. An agent seeking a schedule slot may trigger a chain reaction of rescheduling attempts, as any agents which would be displaced due to a preemption (including indirect displacement as the alteration of one schedule slot may require rescheduling other treatment steps whose constraints are now violated) must seek alternate slots which may displace other agents. These chains only stop when gaps are found in a resource's schedule, likely at the end of the schedule if the resource is fully booked. Paulussen et al. evaluate this method using existing Taillard shop benchmark problems, which require hundreds of negotiation rounds to settle on a steady state.

This solution explores a complex resource allocation problem, handling scheduling constraints between tasks. However, this solution converges slowly and does not react well to changes such as dynamic task arrivals.

6.1.2 Overlapping Potential Game Approximation

Chapman et al. [5] introduce a multi-agent approach to resource allocation which coordinates agent behavior through the use of game theory; note that while this algorithm uses game theory concepts such as potential games and Nash equilibria, the setting is assumed to be cooperative so that agents may rely on truthful reporting. This technique is

demonstrated through a Robocup Rescue simulation, which is a simulation of a disaster response scenario frequently used to benchmark multi-agent coordination problems. The tasks in this problem are rescue operations, with new endangered civilians discovered over the course of the simulation (thus dynamically adding tasks); these tasks have deadlines, where task completion derives no utility after the deadline passes. The resources are ambulances, which all have identical capabilities and can cooperate in the rescue of individual civilians. The utility function is measured in terms of civilians rescued, with a minor bonus for rescuing individual citizens quickly.

Conceptually, this allocation strategy endows the resources with the responsibility to address tasks, rather than requiring tasks to seek out resources. The resource allocation problem is formulated as a game where the participants are the resources, the actions available are the allocation decisions available to those resources, and payoffs are determined by calculating the marginal contribution of the agent to the global utility. The authors show that this game is a *potential game*, which has two useful properties:

1. Every finite potential game possesses at least one pure strategy Nash equilibrium – there exists a set of strategies where each agent follows a fixed, deterministic strategy and cannot perform any better by changing its own strategy.

2. Any sequence of unilaterally improving moves converges to such an equilibrium in finite time – local improvements will reach this stable state where no further local improvements can be made.

The algorithm itself is a stochastic search: An agent will play the best response to other moves with probability p, otherwise the agent will continue to play the same strategy as it did last move. Once an equilibrium is found, no agents will leave the equilibrium under this strategy, unless the game itself changes (i.e. if new tasks are discovered) in which case the stochastic search will continue searching for improvements. The strategy itself is constructed in blocks of actions, with each new block using the previous block of actions as initial conditions when determining the best responses to play. This is reminiscent of the field of Adaptive Planning [2], which is centered around saving computation by adapting existing plans to new scenarios.

This solution makes good use of a distributed model, with low loss of performance when the flow of information is restricted. However, differences in the abilities of different resources are not captured in the model presented (unlike how our model differentiates resources by resource type), and not all tasks can be completed faster by providing more resources.

6.1.3 Branch-and-Bound Fast-Max-Sum

Macarthur et al. [13] recognize that resources may differ, and formulate their problem domain accordingly. Each resource is represented by an agent a_i, which can perform any one task in the subset T_i of tasks T which are suited to that resource. Resources may cooperate by performing the same task, though not necessarily with additive utility; the marginal contribution of each resource is used to determine how much utility from a cooperative venture is attributable to each resource involved. The allocation algorithm then uses a modification of fast-max-sum (FMS) which exploits the small size of relevant task subsets T_i to quickly converge to an optimal assignment by maximizing the available values of a factor graph. The worst-case computational complexity of FMS is $O(|F| \times 2^{d_{max}})$, where $|F|$ is the number of functions (possible coalitions of resources for every task) and d_{max} is the maximum arity of a function (the maximum number of agents capable of completing any given task). While the paper addresses how to use branch-and-bound optimization to account for $|F|$, tasks which can be completed by many different resources would disrupt runtime guarantees.

This solution solves a sophisticated matching problem where resources are each uniquely suited to a small group of the available tasks–while the paper does not discuss an intended problem domain, this could reflect attempts to allocate specific factory machinery (resources) to handle manufacturing requests (tasks) which can only be satisfied by certain types of machines. However, these bounds are sharp by necessity. While our solution allows some resources to be better suited to certain tasks, we allow any resource to resolve any task if necessary, and this flexibility is not well reflected in the domain of this algorithm.

6.1.4 Electric Elves

Scerri et al. [21] introduced the concept of transfer-of-control (TOC) strategies which are used in our system for task agent plans, but there are differences in domain which change how our system uses TOC strategies compared to this prior work. While the resources in both problems are not certain to respond to requests, Scerri's work has a continuous time model which integrates the probability of resource response over arbitrary lengths of time when determining the optimal amount of time to wait before moving to the next step of the TOC strategy. The quantized time in our system helps simplify this optimization problem to a sequence of decisions which are largely isolated, as at any given point in the TOC strategy all previous requests must have failed.

6.1.5 Conditional Planning

Conditional planning is the general term for constructing plans which contain branching points in order to handle contingencies [20], which is important in stochastic environments. Whenever there is anything less than perfect information in a fully deterministic world, plans must be able to respond to events that do not happen with certainty. The approach taken with TOC plans in this work does not use the traditional case-based structure of conditional planning, instead only focusing on planning the next few steps and constructing a new plan whenever the existing plan is found to contain flaws. However, a conditional planning approach could construct plans for our system just as easily, as long as the plans account for how tasks require resources for a contiguous length of time.

6.2 Future Work

There are several clear extensions for the work presented in this thesis. The algorithm presented for multi-agent resource allocation makes different simplifying assumptions from those presented by existing algorithms. The application of this algorithm to the different problem domains presented by others offers nontrivial opportunities for expansion.

The existing algorithms are fairly computationally efficient, capable of resolving a system with 5000 tasks and 100 resources during one test. Future improvements can freely focus on adding more detail to the model to improve the utility of allocations.

The "dummy agent" as presented in this thesis has definite room for improvement. The algorithms provided rely on oracular knowledge of incoming tasks, which is justifiable in some domains for a short horizon (provided there is a gap between becoming aware of a necessary task and becoming capable of addressing the task). There are two clear paths for advancement here: First, this limited knowledge of the future can be harnessed more fully in an attempt to derive benefit in more general scenarios than the selection provided; second, the oracular assumption can be dropped in favour of a predictive approach which uses prior knowledge to anticipate future task arrivals and take action based on imperfect information. In this latter case, it would be important to solve the problem of selecting the optimal resource to reserve in cases where the specific type of expected task is uncertain, as an ideal algorithm would reserve resources that are suited to incoming tasks. Another potential expansion would be the introduction of multiple dummy agents, thus allowing the system to keep more than one resource reserved; these dummy agents would need to coordinate with each other in order to avoid reserving more resources than there are expected task arrivals.

While our solution uses a dynamic programming approach to construct Transfer-of-Control (TOC) strategy plans, there is nothing preventing the use of alternative planning techniques to decide on task agent actions as long as these plans can provide a closed-form estimate of the utility expected by the task agent. However, as our existing planning algorithm does find optimal solutions for the problem information available at the time of planning, the utility produced by our system cannot be improved by a different planning technique. Instead, better estimates of the problem conditions would help construct better plans.

The use of congestion and churn as planning parameters which update based on environmental interactions is a simple form of heuristic learning, but there are more advanced techniques which could be used. Partially-Observable Markov Decision Processes [14] are tools for learning the transition function of an unknown state space, which may provide a more sophisticated view of the level of competition for resources in the system. The Markovian assumption (that state transitions are solely dependent on the current state and actions, with no influence from historical state transitions) may not hold if the distribution of task arrivals changes over time, but an adaptive model should be able to learn recent trends and provide this information to improve the use of plans. Alternatively, task agents could explicitly reason about the probability that other task agents will make competing requests.

The assumption of discrete synchronized time steps has simplified the analysis and implementation of our algorithms, but a true distributed system could potentially operate asynchronously in continuous time. The prior work on TOC strategies already defines these strategies over continuous time intervals, which could be used by our task agents. The concept of the resource proxy agent filtering incoming requests would not necessarily translate well to this framework, though instituting a window of opportunity for challengers could provide some of this filtering benefit while providing immediate feedback to any task agents who are surpassed by new challengers for the same resource.

6.3 Contributions

This thesis offers three contributions to the field of multi-agent resource allocation. The first is an improved distributed resource allocation framework which learns about the congestion and churn of its environment to fine-tune plans based on Transfer-of-Control strategies and use these plans to rapidly find good allocations in a dynamic environment. This algorithm can leverage differences between resources in how quickly they can process tasks, and the distributed nature of this approach allows the addition of new tasks without costly

recalculation for all existing tasks. Bother modelling is used to model an unreliable physical world beyond the simulation, indicating that the results of this approach are robust. However, constraints between individual tasks are not addressed, and it is assumed that multiple resources cannot cooperate to perform a single task. If these properties are important to the reader's domain of interest, then there are other multi-agent resource allocation approaches which address these concerns.

The second contribution is a discussion of the importance of opportunity cost in resource allocation, accompanied by a sample instantiation of opportunity cost: A "dummy agent" which behaves like any other resource-seeking agent can represent the opportunity cost of allocation by taking on the priority of a task which is likely to enter the multi-agent system in the near future. This agent thus reserves a resource for an impending task, preventing wasteful preemption where a low-priority task acquires a resource which they will immediately lose on the arrival of the predicted task. This general approach should be applicable to a variety of multi-agent resource allocation systems.

The third contribution is an evaluation of the first two contributions in a series of environments where preemption is costly or impossible.

The ability to leverage preemption in resource allocation is supported by a very small number of the existing models. The two primary approaches, those of Paulussen et al. [16] and Doucette [10] have both been surpassed by the framework proposed in this thesis. As such we offer an advancement in multi-agent resource allocation, one that focused on managing dynamic task arrivals, on reserving resources for the benefit of anticipated task arrivals, and on smoothly transitioning to environments where preemption is costly or impossible.

APPENDICES

Appendix A

Extended Example

The following extended example may help to illustrate the finer details of the resource allocation model and algorithms described in Chapter 3, as well as the adjustments to this model described in later chapters.

A.1 Domain-Specific Functions

The medical example used in Section 3.3.1 provided instantiations for the domain-specific functions EU, $completion$, P_{stop}, and $bother\_increase$ which are required to complete the resource allocation model. This introduced the concept of patient severity for task types, which are natural numbers ranging from 1 to 500, as well as doctor types for resource types, which are arrays specifying the doctor's skill at treating patients of any given severity (from 1 for a relatively unskilled doctor to 10 for a specialist in that condition). The utility function used then focuses on minimizing the time that a patient spends in the system, weighted by the patient's severity. These functions are repeated here for clarity:

$$completion(\theta_r, \theta_t) = 11 - \theta_r[\theta_t]$$

$$EU(old\_completion, new\_completion, \theta_t) = (old\_completion - new\_completion) \times \theta_t$$

$$EU(\infty, new\_completion, \theta_t) = (12 - new\_completion) \times \theta_t$$

$$P_{stop}(\Delta EU, BSF) = (1.5 - S(-BSF)) \times (0.5 + 0.5 \times S(\Delta EU))$$

$$\text{where } S(t) = \frac{1}{1 + e^{-t}}$$

$$bother\_increase(\Delta EU, BSF) = \begin{cases} 1.25 * BSF + 1 & : \Delta EU < -50 \\ 0.75 * BSF + 1 & : \Delta EU > 50 \\ BSF + 1 & : otherwise \end{cases}$$

A.2 Example Problem

Consider a scenario with two doctors (resources) $r1, r2$ and three patients (tasks) $t1, t2, t3$ present at the start of simulation. Each of the doctors has a resource proxy agent $x1, x2$ while each of the patients has a task agent $a1, a2, a3$. Each of these task agents must begin by constructing a plan for which resource proxy agents to approach; in order to do this, the skill of each doctor must be evaluated with respect to the task agent's patient's type. Suppose the patients are respectively type 1, type 2, and type 5, while doctor $r1$ can treat those types in 1, 2, and 2 ticks respectively, and doctor $r2$ can treat those types in 2, 3, and 3 ticks respectively. In this case, without knowing about the other agents, each task agent will plan to request $r1$ from $x1$ as this will minimize the time that the task agent's patient will spend in the system, thus maximizing utility. Assume that plans are length 3, so each task agent has planned to request $r1$ for the next three steps.

Table A.1: State of allocation at system start

Task Agent	Allocated Resource
a1	None
a2	None
a3	None

For a closer look at plan generation, consider agent $a2$. By requesting resource $r1$, the completion time of $t2$ will drop from an unbounded value to 2 ticks, which is an expected utility change of $EU(\infty, 2, 2) = (12 - 2) \times 2 = 20$. By requesting resource $r2$, the new completion time will be 3 ticks, for an expected utility change of $EU(\infty, 3, 2) = (12 - 3) \times 2 = 18$. These values are calculated by the resource proxy agents $x1$ and $x2$, incorporating bother; however, as neither resource has been bothered at all, the probability of each resource accepting the request is 1 (provided no competition). As both of these values are positive, the higher value indicates which resource should be requested, so $a2$ plans to request $r1$. Similar calculations fill in the remaining two steps of the plan as further requests for $r1$.

In the first tick, all task agents must make resource requests before resource proxy agents act. In each case the task agent will ask the relevant proxy agent to verify that the

request is sensible (increases utility) before actually making the request, but as the utility of each request is positive this verification step will succeed for everyone. Suppose $a2$ makes the first request for $r1$, which $x1$ records as a request that can potentially improve utility by 20. If $a1$ then requests $r1$, the expected utility gain of $EU(\infty, 1, 1) = 11$ is less than 20, so this request will fail. Then, the request by $a3$ for $r1$ is examined by $x1$ to find that the expected utility gain of $EU(\infty, 2, 3) = 30$ which is greater than 20, so this request is now recorded as the best request so far. As all task agents have made resource requests, now the resource proxy agents begin work. The resource proxy agent $x2$ did not receive any requests, so it and $r2$ do nothing. The resource proxy agent $x1$ conveys the recorded best request from $a3$ to the doctor $r1$, incrementing the BSF of the doctor by $bother\_increase(30, 0) = 1$. As a result, $r1$ is allocated to $t3$, and both $a3$ and $x1$ record that this allocation requires 2 ticks of work to satisfy the task, the first of which happens immediately.

Table A.2: State of allocation at end of first tick

Task Agent	Allocated Resource
a1	None
a2	None
a3	r1 (1 tick left)

At the beginning of the second tick, agents $a1$ and $a2$ see that the first plan step was unsuccessful, so the rest of the plan will remain. However, agent $a3$ was successful, so a new plan is required. In this small example, there are only two resources and an agent cannot request a resource which they already own, so the agent's plan must consist of approaching $r2$. When it does so, the initial verification step will indicate that the expected utility of granting this request is $EU(1, 3, 3) = (1-3) \times 3 = -6$, so agent $a3$ will have one last chance to adjust the plan before making a real request. The only available request still provides negative utility, so the resource proxy agent $x2$ will disregard this request. Since agent $a3$ has already secured the best doctor for its patient, this plan and resource request cannot provide benefit, but if $a3$ were to lose $r1$ then this plan would be immediately available to act upon.

Meanwhile, agent $a2$ will try to verify that the request for $r1$ is still sensible, but $r1$ has been allocated since $a2$'s plan was constructed. The resource proxy agent $x1$ will evaluate the expected utility of this preemption request, comparing the utility of stopping work on $t3$ to complete $t2$ first to the utility of finishing work on $t3$ before completing $t2$ next. The benefit to the requester is thus $EU(1 + 2, 2, 2) = 2$ as being completed before $t3$ will save the one tick that would otherwise be spent finishing $t3$. However, the loss by $t3$ can be

calculated as $EU(1, 2+2, 3) = (1-4) \times 3 = -9$, as losing the resource would require $t3$ to not only wait for $t2$'s completion but also lose the accumulated work and start over. Finally, the current value of $t3$'s plan is negative, providing no offset to encourage preemption in this resource-sparse system. The total utility of this preemption request is thus negative, so the verification fails. The churn model for $a2$ uses the knowledge that verification failed in order to reduce the length of future plans, since the allocation environment is changing quickly. The task agent $a2$ then gets a chance to replan before making a real resource request, which is used to choose $r2$ as that request is still positive; the resource proxy agent $x2$ records this request. Similarly, $a1$ fails verification and replans in order to request $r2$, but $x2$ compares this request to the request from $a2$ and finds that $a2$ has a higher utility ($EU(\infty, 3, 2) = 18$ instead of $EU(\infty, 2, 1) = 10$). When the resource proxy agents act, $x1$ has received no real resource requests so it does nothing while $r1$ finishes work on $t3$, allowing $t3$ and $a3$ to depart from the system. The best request received by $x2$ was from $a2$, so $r2$ is allocated to $t2$ and performs the first tick of work, so both $x2$ and $a2$ know that 2 ticks of work remain. The bother model for $r2$ is also updated, while the BSF for $r1$ decays slightly.

Table A.3: State of allocation at end of second tick

Task Agent	Allocated Resource
a1	None
a2	r2 (2 ticks left)
a3	r1 (0 ticks left, complete)

At the beginning of the third tick, agents $a1$ and $a2$ remain, with $r1$ unallocated while $r2$ is allocated to $a2$. While agent $a1$ can continue with its current plan of requesting $r2$, the success of agent $a2$'s plan last step necessitates reconstructing the plan, requesting the only other resource $r1$. Consider now what happens if a new task $t4$ enters the system, receiving a task agent $a4$. Let this new task be type 4, which can be treated by doctor $r1$ in two ticks and by doctor $r2$ in one tick. When $a4$ constructs a plan, it receives a utility quote of $EU(\infty, 2, 4) = (12-2) \times 4 = 40$ from resource proxy agent $x1$ for theoretically requesting $r1$, while a theoretical request for $r2$ would require a preemption to displace $t2$ which has 2 ticks of work remaining. This preemption would improve utility for $t4$ by $EU(2+1, 1, 4) = 8$, while the utility of $t2$ would drop by $EU(2, 1+2, 3) = -3$, so $a4$ will plan to request the available $r1$ instead. However, as $a4$ plans three ticks into the future, on the third tick $x2$ knows that $r2$ will be available, offering a utility improvement of $EU(\infty, 1, 4) = 44$. The final plan for $a4$ is thus $r1, r1, r2$. While $a1$ will try to verify its request for $r2$ and fail verification as the preemption attempt would cost a net total of

$EU(2+2,2,1)+EU(2,2+3,2)=2-6=-4$, the new plan for $a1$ to request $r1$ will once again fail to acquire a resource as $a4$ also requests $r1$ with higher potential utility. Once the resource proxy agents act, $r1$ will be allocated to $a4$ and perform the first tick of work while $r2$ will continue to work on $t2$ with the expectation that $t2$ will be finished next tick.

Table A.4: State of allocation at end of third tick

Task Agent	Allocated Resource
a1	None
a2	r2 (1 tick left)
a3	Complete
a4	r1 (1 tick left)

In the fourth tick, very little changes. Agent $a4$ constructs a new plan since the previous plan was successful, but all resource requests fail as the net utility change from preemption is too high to allow benefit. Once resource proxy agents act, $r1$ finishes $t4$ and $r2$ finishes $t2$.

Table A.5: State of allocation at end of fourth tick

Task Agent	Allocated Resource
a1	None
a2	r2 (0 ticks left, complete)
a3	Complete
a4	r1 (0 ticks left, complete)

In the fifth tick, $a1$ makes one last request for $r1$ which is granted as there is no other competition, and $r1$ is allocated to $a1$ with enough time to finish $t1$ immediately. If no other tasks arrive, then this simulation is complete.

Table A.6: State of allocation at end of fifth tick

Task Agent	Allocated Resource
a1	r1 (0 ticks left, complete)
a2	Complete
a3	Complete
a4	Complete

References

[1] James F. Allen, Curry I. Guinn, and Eric Horvitz. Mixed-initiative interaction. *Intelligent Systems and their Applications, IEEE*, 14(5):14–23, 1999.

[2] Richard Alterman. An adaptive planner. In *AAAI*, volume 86, pages 65–69, 1986.

[3] Pranjal Awasthi and Tuomas Sandholm. Online stochastic optimization in the large: application to kidney exchange. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 405–411, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[4] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, 4(2):125–144, 1992.

[5] Archie C. Chapman, Rosa Anna Micillo, Ramachandra Kota, and Nicholas R. Jennings. Decentralised dynamic task allocation: a practical game: theoretic approach. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 915–922. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[6] Michael Y. K. Cheng and Robin Cohen. A hybrid transfer of control model for adjustable autonomy multiagent systems. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 1149–1150, New York, NY, USA, 2005. ACM.

[7] Michael Yu-Kae Cheng. A hybrid transfer of control approach to designing adjustable autonomy multi-agent systems. Master's thesis, University of Waterloo, 2005.

[8] Yann Chevaleyre, Paul E. Dunne, Ulle Endriss, Jérôme Lang, Michel Lemaitre, Nicolas Maudet, Julian Padget, Steve Phelps, Juan A. Rodríguez-Aguilar, and Paulo Sousa. Issues in multiagent resource allocation. *Informatica*, 30(1):3–31, 2006.

[9] Robin Cohen, Michael Y. K. Cheng, and Michael W. Fleming. Why bother about bother: Is it worth it to ask the user? In *AAAI'05 Fall Symposium on Mixed-Initiative Problem-Solving Assistants*, 2005.

[10] John A. Doucette. An ex-ante rational distributed resource allocation system using transfer of control strategies for preemption with applications to emergency medicine. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 2012.

[11] Michael Fleming and Robin Cohen. A decision procedure for autonomous agents to reason about interaction with humans. In *Proceedings of the AAAI 2004 Spring Symposium on Interaction between Humans and Autonomous Systems over Extended Operation*, pages 81–86, 2004.

[12] Richard M. Karp, Umesh V. Vazirani, and Vijay V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 352–358, New York, NY, USA, 1990. ACM.

[13] Kathryn S. Macarthur, Ruben Stranders, Sarvapali D. Ramchurn, and Nicholas R. Jennings. A distributed anytime algorithm for dynamic task allocation in multi-agent systems. In *AAAI*, 2011.

[14] George E. Monahan. State of the art–a survey of partially observable markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[15] M. Parkin and R. Bade. *Microeconomics: Canada in the Global Environment, Eighth Edition*. Pearson Education Canada, 2012.

[16] T. O. Paulussen, N. R. Jennings, K. S. Decker, and A. Heinzl. Distributed patient scheduling in hospitals. In *International Joint Conference on Artificial Intelligence*, volume 18, pages 1224–1232, 2003.

[17] T. O. Paulussen, A. Zöller, A. Heinzl, A. Pokahr, L. Braubach, and W. Lamersdorf. Dynamic patient scheduling in hospitals. *Agent Technology in Business Applications*, 2004.

[18] Graham Pinhey, John Doucette, and Robin Cohen. Distributed multiagent resource allocation with adaptive preemption for dynamic tasks. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 1441–1442. International Foundation for Autonomous Agents and Multiagent Systems, 2014.

[19] Ryan Porter. Mechanism design for online real-time scheduling. In *Proceedings of the 5th ACM conference on Electronic commerce*, EC '04, pages 61–70, New York, NY, USA, 2004. ACM.

[20] Stuart Jonathan Russell, Peter Norvig, John F. Canny, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Englewood Cliffs, 1995.

[21] Paul Scerri, David V. Pynadath, and Milind Tambe. Why the elf acted autonomously: Towards a theory of adjustable autonomy. In *Proceedings of AAMAS'02*, 2002.