Reducing Redundant Autonomous Mobile Program Movements by Negotiation

Natalia Chechina, Peter King and Phil Trinder
Department of Computer Science,
School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh, EH14 4AS, UK

January 22, 2010

Abstract

Autonomous mobile programs (AMPs) aim to balance computational loads in dynamic networks. AMPs periodically recalculate network parameters to seek a better execution environment and intend to both reduce execution time and better exploit the network. AMP behaviour has previously been investigated using mobile languages like Java Voyager and using simulation.

AMPs suffer from greedy effects, which introduce redundant movements during balancing. The greedy effects are a result of the locally optimal choice made by each AMP. The majority of redundant moves are due to the lack of information about intended moves of other AMPs.

In this paper we identify two forms of greedy effects and propose the concept of negotiating AMPs (NAMPs) that communicate their intentions with the view to reduce redundant moves. While a number of negotiation schemes are possible, we have designed and simulated AMPs with a competitive scheme (cNAMPs). cNAMPs announce their intentions to move and compete with each other for opportunity to transfer to the new location. An analysis of simulated cNAMP results shows that even this simple negotiation significantly decreases both the number of redundant movements and the time to rebalance. For example, cNAMPs make no redundant movements during initial distribution, which makes initial balancing at least 3 times faster in comparison with AMPs.

1 Introduction

Autonomous mobile programs (AMPs) are mobile agents that improve execution efficiency by managing load; AMPs are aware of their resource needs, sensitive to the execution environment and periodically seek a better location to reduce execution time [DMT10]. AMPs do not communicate with

each other, only with a *load server* which collects network state information to reduce AMP coordination time. To analyse AMP behaviour on local area networks (LANs) we have constructed a simulation model [CKPT09]. Comparing the simulation results and observations of the real system [DMT10] shows that simulated and real AMPs enter the same *stable states* (i.e. where no AMP can reduce its execution time by moving). The differences between simulated and real AMPs are minor and readily explained.

Like other distributed load balancing systems [NXG85, RM90, LM82] both real and simulated collections of AMPs exhibit *greedy effects*. These greedy effects are a phenomenon that results in redundant AMP movements during the balancing of loads between locations, and are a result of locally optimal choices made by each AMP.

The aim of the current paper is to examine properties and features of the greedy effects in collections of AMPs and expose their causes. We further aim to reduce the greedy effects, and to estimate the greedy effects in the modified algorithm. The estimation of the greedy effects is implemented by using simulation.

The paper is organised as follows. We discuss AMP related concepts (Section 2), and identify two forms of greedy effects (Section 3). The AMP cost model and simulation are adapted to facilitate an investigation of the greedy effects (Section 4). We examine the AMP greedy effects on initial distribution and rebalancing experiments (Section 5). Analysis of types of movements shows that the majority of redundant movements occur because an AMP is unaware of the intentions and movements of other AMPs. So, we discuss ways to reduce the greedy effects and propose the concept of negotiating AMPs (NAMPs) that communicate their intentions with the view to reduce redundant moves. While a number of negotiation schemes are possible, we have designed and simulated AMPs with a competitive scheme (cNAMPs) (Section 6). cNAMPs announce their intentions to move and compete with each other for opportunity to transfer to the new location.

An analysis of simulated cNAMP results shows that even this simple negotiation significantly decreases both the number of redundant movements and the time to rebalance. For example, cNAMPs make no redundant movements during initial distribution, which makes initial balancing in the conducted experiments at least three times faster in comparison with AMPs (Section 7). A summary of the results and discussion of future work are provided in Section 8.

2 Related Work

2.1 Introduction

This Section discusses related work, first covering process migration in general (Section 2.2), before covering the three core AMP technologies: load

balancing (Section 2.3), mobile computing (Section 2.4) and autonomous systems (Section 2.5).

2.2 Process Migration

The idea of relocating a process during execution has existed for some time, and is termed *migration* or *rescheduling* by different communities. Much work was done on load management using task migration in distributed operating systems in the 1970s [MDW99], and some well known examples are Mach [BRS+85] and MOSIX [BL98]. Sophisticated distributed memory implementations of parallel programming languages support task migration, for example the Charm parallel C++ [KK93]. However, where parallel languages are typically designed for homogeneous dedicated architectures, AMPs operate on heterogeneous shared architectures. Moreover, both distributed operating systems and parallel programming languages differ from AMPs as the tasks are passive, and the scheduling is typically centralised.

Grid workflow reschedulers are more closely related to AMPs, and are currently the focus of considerable research effort. An excellent taxonomy of Grid workflow management systems can be found in [YB05]. Like AMPs Grid workflow reschedulers operate on heterogeneous shared networks, and many of them make decentralised scheduling decisions, use performance prediction to inform scheduling decisions, and reschedule after periodic reassessment of system status. The AMP approach is novel in automating the performance prediction process as a one-off, compile-time program analysis, and in devolving the rescheduling decisions to individual programs. Effective load management results from emergent behaviour of collections of AMPs.

AMP behaviour has previously been investigated using mobile languages like Java Voyager [Den07] and using simulation [CKPT09].

2.3 Load Balancing

The problem of load balancing in distributed computer systems has been widely studied, e.g. [vRvST89], [KK90], [SKS92]. The main difficulties which load balancing approaches face are minimizing execution time, minimizing communication delays, and maximizing resource utilization [SKH95]. According to taxonomy in [CK88], AMPs are global dynamic load balancers. Detailed classification of dynamic load balancers is presented in [Rot94]. As AMPs make decisions themselves using information collected by load servers, regularly recalculate parameters to search for better location to execute, and choose the location which has the shortest execution time adjusted for communication time; therefore, AMPs have a hybrid decision making policy, sender initiating scheme and use simple transformation with best fit approach.

The greedy effects which are a result of the locally optimal choice made

by each AMP are a fundamental problem of load balancing. Other terms for the greedy effects are *processor thrashing* [Kuo85], *task thrashing* [GA91], *task dumping* [NXG85, RM90], *transmitting dilemma* [LM82]. To reduce the number of redundant movements different techniques are used, such as limiting any particular task to a maximum number of migrations [GA91], and calculation the largest difference between the estimated execution time and the interprocess communication cost [EAEB97]. The techniques aim to reduce the greedy effects in case where locations have a task scheduler. The goal of the current research is to estimate and reduce the greedy effects when each AMP makes a decision to move itself.

2.4 Mobility

Mobile computations can move between locations in a network and potentially enable better use of shared computational resources [Kir02]. Basically a mobile program can transport its state and code to another location in a network, where it resumes execution [LO99].

Fuggetta et al. distinguish two forms of mobility supported by mobile languages [FPV98]: weak mobility is the ability to move only code from one location to another; strong mobility is the ability to move both code and its current execution state. AMPs were constructed using languages with weak and strong mobility. However, a substantial subset of experiments was conducted using Java Voyager [Rec08] which supports only weak mobility.

2.5 Agents and Autonomous Systems

Agent technology is a high-level, implementation independent approach to developing software as collections of distinct but interacting entities which cooperate to achieve some common goal. With the continuing decline in price and increase in speed of both processors and networks, it has become feasible to apply agent technology to problems involving cooperation in distributed environments. In particular, where agents may change location, typically to manipulate resources in varying locations.

An agent is "an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives" [Woo97, Tos04]. An agent with mobility is called a mobile agent [MDW99], and AMPs are mobile agents.

Autonomous systems are also called autonomic computing systems, and a definition has been given by IBM: "autonomic computing system can manage themselves given high-level objectives from administrators" [KC03, Mur04]. Four aspects of self-management are self-configuration, self-optimisation, self-healing, and self-protection. Different autonomic systems may have some or all of these four aspects. AMPs are primarily self-optimisation

systems. They are aware of their processing resource needs and sensitive to the environment in which they execute, and are able to dynamically relocate themselves to minimise processing time in the presence of varying external loads on shared locations.

Most distributed environments are shared by multiple users. In particular, distributed agent-based systems must also contend with external competition for resources, not least for the locations they share. The agents community has focused on autonomous problem solving, which can act flexibly in uncertain and dynamic environments. Mobile languages provide efficient tools to allow the agent to move more flexibly in a large scale network, which makes it possible to build self-management systems (autonomous systems) for resource sharing using agent technology. Hence, many autonomous systems are based on mobile agents [Woo97].

AMPs have strong connections with both agents and autonomous systems, but they also have important differences. Firstly, unlike previous mobile agents approaches, AMPs have cost models and are autonomous, making the decision themselves on when and where to move according to the cost model. Furthermore, unlike traditional autonomous systems [KC03, Aba05, DMV04], which use schedulers to decide whether to move, AMPs can make the decision themselves [DTM06].

3 Greedy Effects

An *optimal rebalancing* is a sequence of AMP movements that is the minimum number of AMP movements needed to enter a stable state.

The AMP *greedy effects* are the result of a non-optimal AMP rebalancing, which differs from the optimal rebalancing in having additional redundant movements, and is a result the AMP making a locally optimal choice, i.e. AMPs do not possess sufficient and accurate state information to make the optimal movement decision.

There are two types of the greedy effects: location thrashing and location blindness. Both location thrashing and blindness are observed in real [DTM06] and simulated [CKPT09] AMP experiments.

3.1 AMP Distribution Scenarios

To illustrate the greedy effect we first introduce the following AMP scenarios, each of which specifies the number of AMPs and locations, and types of locations:

- *Scenario 1*: 25 AMPs on 15 locations with CPU speeds 3193 MHz ($Loc1 - Loc5$), 2167 MHz ($Loc6 - Loc10$) and 1793 MHz ($Loc11 - Loc15$).

- *Scenario 2*: 20 AMPs on 10 locations with CPU speeds 3193 MHz ($Loc1 - Loc5$), 2168 MHz ($Loc6$) and 1793 MHz ($Loc7 - Loc10$).
- *Scenario 3*. 10 AMPs on 3 locations with CPU speeds 3193 MHz.

For all scenarios $Loc1$ is the root location. By the *root location* we mean the location where all AMPs start; it is also called either *initiating location* or *first location* in [DMT10]. In the experiments we use large and small AMPs. Large AMPs are programs of matrix multiplication of size 1000×1000, and small AMPs are programs of matrix multiplication of size 500×500.

3.2 Location Thrashing

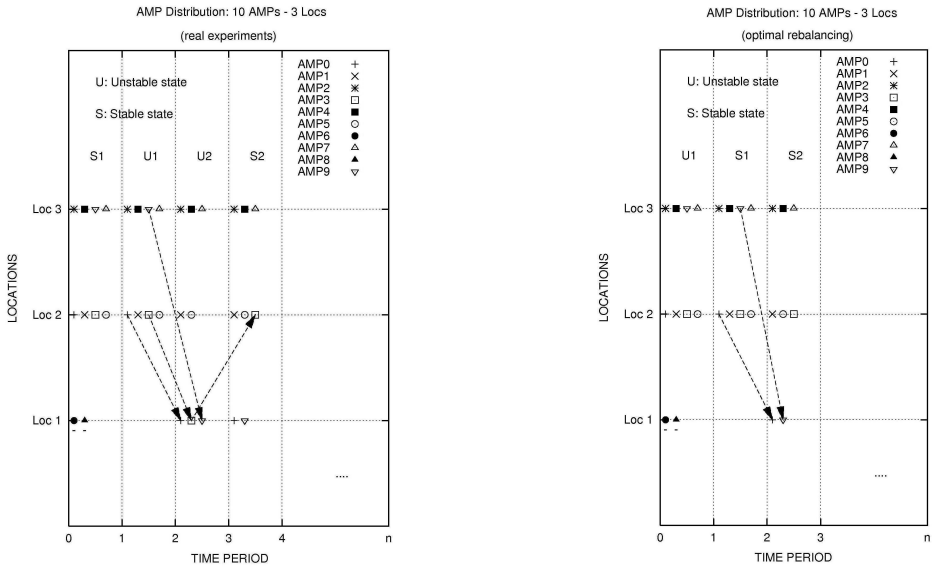
Location thrashing is the greedy effect resulting from an AMP's lack of information about other AMPs intending to move to the same location. That is, two or more AMPs decide to move on the basis of the same information about the target location, which causes further AMP retransferring. Location thrashing occurs in dynamic load balancing systems; other terms are *processor thrashing* [Kuo85], *task thrashing* [GA91], *task dumping* [NXG85, RM90], *transmitting dilemma* [LM82].

Location thrashing is illustrated in Figure 1(a), which shows AMP movements in the experiments with the real system [DMT10], based on scenario 3 (Section 3.1). In Figure 1(a) each icon denotes an AMP. The locations are specified on the vertical axis and the horizontal divisions represent time intervals. The time intervals are of different lengths, showing states that the system enters as it attempts to reach a stable state.

After the termination of two AMPs from $Loc1$ in state S1, state U1 is entered and two AMPs from $Loc2$ and one AMP from $Loc3$ discover a better opportunity for execution on $Loc1$ simultaneously. These three AMPs move to $Loc1$ (state U2), and then one AMP moves back to $Loc2$ to enter a stable state (state S2). In this case, an optimal rebalancing from the state U1 to the state S2 can be reached by one AMP movement to $Loc1$ from each of $Loc2$ and $Loc3$ as Figure 1(b) shows. Note that location thrashing incurs two performance penalties, namely the cost of additional communication and the cost of slower execution. By *additional communication cost* we only mean additional AMP movements during a rebalancing and not communication time they may take.

3.3 Location Blindness

Location blindness is the greedy effect resulting from an AMP's lack of precise information about the remaining execution time of other AMPs. The problem is not with poor runtime predictions, but rather an inability to obtain accurate AMP runtime predictions at distributed locations, i.e. the



(a) Redundant rebalancing

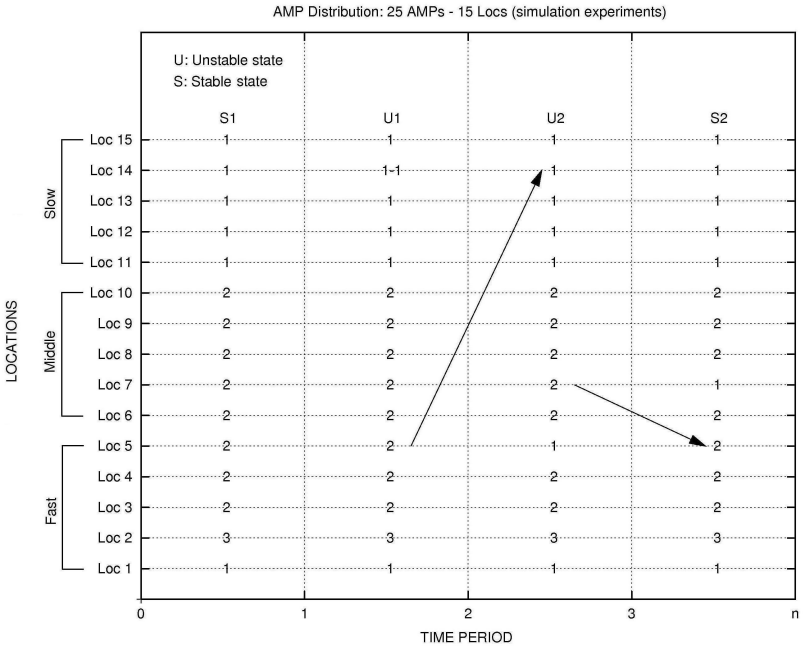
(b) Optimal rebalancing

Figure 1: Location Thrashing Greedy Effect [DTM06]

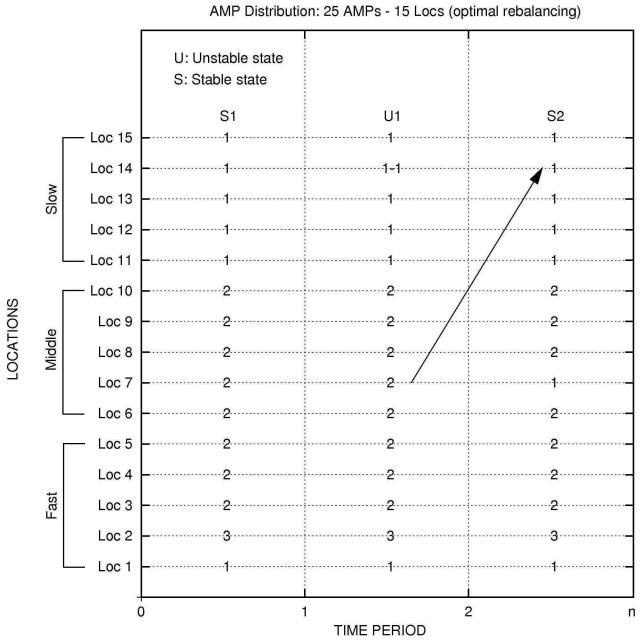
more accurate information that is required, the more *expensive* it becomes to collect and process the information in a distributed system [CK87].

Figure 2(a) shows an example of location blindness in some simulated AMP experiments. The experiment is undertaken on the basis of scenario 1 (Section 3.1). Numbers identify the number of AMPs on a location. After an AMP termination on $Loc14$ in state S1 and the system enters state U1 an AMP from $Loc5$ discovers the opportunity for faster execution first and moves (state U2). Then an AMP from $Loc7$ discovers the opportunity for faster execution on $Loc5$ and also moves (state S2). Figure 2(b) shows optimal AMP rebalancing. In contrast to the location threshing, location blindness only causes redundant communication and causes no additional computation cost. Each AMP will have improved its environment by moving.

Thus, among two types of the greedy effects the location thrashing is more harmful, because it causes an increase in AMP execution time, and hence decreases AMP efficiency. To analyse the number of redundant movements and time to balance we investigate the greedy effects with simulated AMPs.



(a) Redundant rebalancing



(b) Optimal rebalancing

Figure 2: Location Blindness Greedy Effect [CKPT09]

4 Simulating the Greedy Effects

Earlier simulation experiments of AMPs on a LAN showed that the simulation closely models real AMPs on LANs, and, hence, is an effective tool to analyse AMP behaviour [CKPT09].

We provide essential calculations and discuss properties and features of AMP interaction with other AMPs and the load server (Section 4.1). We propose minor changes to the previous simulation model [CKPT09] which allow the simulation of the greedy effects (Section 4.2). These changes include bounds on information transfer times and delays on the transfer of state information.

4.1 Cost Model and Parameterisation

We investigate the greedy effects by simulating specific scenarios (Section 3.1). The key equations of the AMP cost model, defined in [Den07, Section 3.3.2], are repeated here, and the parameter values are given in Table 1.

The term *available speed*, which is the execution speed of a single AMP on that location, i.e.

$$S = (CPU speed) \cdot (1 - non\ AMP\ load), \tag{1}$$

is used to differentiate the total resources of a location from the resources available for AMPs. In the current research, as well as in the previous AMP investigation [DMT10], we assume that all resources of a location are available for AMPs, and the CPU speed coincides with the available speed, except the case of the root location where the external load is higher.

An *AMP relative speed*, R, is an available speed, S, equally divided between the AMPs at a location, x_{loc}, i.e.

$$R = \frac{S}{x_{loc}}. \tag{2}$$

In [DMT10] the *available speed* is called *relative speed*, and the *AMP relative speed* is called *average relative speed*. However, the previous names did not give clear understanding of the concepts and caused confusion during analyses, hence the new terminology.

An AMP execution for a time T_{gran} before it tests the AMP relative speeds of other locations to see if a move will improve its completion time:

$$T_{gran} = \frac{T_{coord}}{O}.$$

The time for coordination, T_{coord}, is evaluated experimentally, and is equal to 0.011s for a load server architecture. The overhead O is taken equal to 5%. So, T_{gran} is chosen in such a way that every 0.22s an AMP seeks for a better location.

Table 1: Parameter Definitions

d	Dimension of square matrix
$gran$	Fragment of work, which must be executed between searches for a better location
N	Number of locations in a network
O	Overhead
T_{comm}	Time for single communication
T_{coord}	Coordination time in the load server architecture
T_{gran}	Execution time of fragment of work $gran$
T_h	Execution time on the current location
T_n	Execution time on the new location
T_{renew}	Time of total state information renewing by load server
T_{req}	Time to send a request to a remote load server and receive a response
T_{res}	Time to send a request in one way
T_{send}	Time to transfer an AMP to the new location
S_h	Available speed of the current location

After executing for T_{gran} an AMP makes a request to the load server of the current location about states of other locations in the network. The load server responds, taking 0.008s, and for 0.003s an AMP makes calculations to decide where to execute in a network. If an AMP decides to stay on the current location, then it continues execution during further T_{gran} seconds, otherwise it moves to a new location taking T_{send} seconds:

$$T_{send} = 0.029 + 5.07 \cdot 10^{-6} \cdot d^2.$$

Here, d is a dimension of matrix, as programs of matrix multiplication are used in the experiments with the real system [Den07]. As soon as an AMP makes a decision to move to a new location, the load server of the current location decreases its number of AMPs. In turn, the new location increases its number of AMPs after the full AMP has arrived.

To obtain state information from other locations in a network, a load server sends requests to locations in a sequential order. The time taken to send a request to a remote Java process and receive a response, T_{req}, using Java Voyager [Rec08] has been measured, and is equal to 0.25s [Den07, p. 80]. Thus, a load server completely renews state information about N locations in a network every T_{renew} seconds:

$$T_{renew} = T_{req}(N - 1). \tag{3}$$

The main rule on the basis of which AMPs make a decision to move to a new location is whether execution time on the current location, T_h, exceeds

execution time on the next location, T_n, and communication delay, T_{comm}:

$$T_h > T_n + T_{comm}. \tag{4}$$

If condition (4) is satisfied, then an AMP moves. Here, communication time is time to transfer an AMP, i.e. $T_{comm} = T_{send}$.

4.2 Adapting the AMP Simulation to Investigate the Greedy Effects

4.2.1 Transferring AMPs

In the simulation model AMP transfer time is exponentially distributed with mean given by T_{send}. However, due to limited data transfer capabilities and hence communication speed, AMP transfer time cannot be less than a certain value. Therefore, we introduce lower bound for an AMP transfer of $0.6T_{send}$. Thus, if the exponential distribution provides a value below lower bound, it returns the value $0.6T_{send}$. Here and further, a value of a factor (i.e. 0.6) is taken approximately as the reasonable in a particular case.

4.2.2 State Information

The load server at a location interrogates the other locations of the network in a predetermined order. The information it holds on any particular location is updated every T_{renew} seconds. In the simulation, the same mechanism is adopted, with a request for load information being sent to each location in turn. Every T_{renew} seconds the load server's information has been completely refreshed.

As T_{req} is time, which takes to send a request and to receive a response, then time, which takes to send a request in one way, T_{res}, is

$$T_{res} = \frac{T_{req}}{2}, \tag{5}$$

thus, $T_{res} = 0.125$ second.

In the *simulation model* a load server sends a request, which comes to the target location over T_{res} seconds. Then the target location assigns its number of AMPs and a response goes back to the initial location. Transferring a response, also, takes T_{res} seconds. The initial location renews state information about the target location and sends a request to the next location. In the *real system* the time to send a request and to receive a response has not a constant value. Therefore, in simulation experiments an exponential distribution with mean given by T_{res} is used.

However, exponential distribution provides very small and very large values. In the real system the time to send a request and to receive a response can not be below or above certain values, because of limited speed

Table 2: AMP Greedy Effect Experiment Summary

Config.	Initial distribution		Rebalancing after an AMP termination		Large AMP execution time, (sec)	
	Mean No. of redun. moves	Mean time, (sec)	Mean No. of redun. moves	Mean time, (sec)	Mean	Standard deviation
Scenario 1	64	60.4	6	22.5	173.8	7.66
Scenario 2	43	50.5	11	28.2	182.1	11.5
Scenario 3	13	26.8	6	14.1	232.6	9.91

of data transferring and time out mechanism respectively. That is why we introduce lower and upper bounds for time of state information transferring. The lower bound is $0.7T_{res}$, and the upper bound is $1.3T_{res}$. So, if the exponential distribution provides a value below lower bound, it returns the value $0.7T_{res}$, and if the value is above the upper bound, it returns the value $1.3T_{res}$.

5 AMP Greedy Effect Experiments

From Section 4 we have conducted a set of experiments on homogeneous and heterogeneous networks, using scenarios we discuss in Section 3.1, and as used in the real system [DMT10].

A *homogeneous network* is a set of locations with the same available speeds, except the root location, which may be different, because of the overheads of initiating the remote processes. A *heterogeneous network* is a set of locations with different available speeds.

Each experiment is repeated eleven times. As the experiments with the real system did not investigate the greedy effects, we have not made systematic comparisons with the real experiments, but the results are consistent where they have been compared.

Experiment 1: Initial distribution This experiment investigates the greedy effects as large AMPs distribute over the network from a single location. Column *Initial distribution* in Table 2 shows the mean number of redundant movements and the time required for the system to enter a stable state. Clearly the optimal number of AMP movements to reach a stable state would occur if each AMP moved a maximum of once, directly to the location it will occupy in the stable state. The column shows that with the increase of the number of locations and AMPs the number of average movements per AMP also increases.

Experiment 2: Rebalancing after an AMP termination The experiment measures the number of movements and time required for a system to rebalance after an AMP termination. In this experiment we use only large AMPs, and not large and small as in the real and previous simulation experiments [CKPT09]. Because the difference in execution time between large and small AMPs is not sufficient enough to estimate large AMP behaviour after termination of small AMPs, i.e. AMPs enter stable states which can be not balanced and hence make fewer movements to rebalance.

So, at the initial time all AMPs are distributed among locations of a network in a way that the system is in a stable state. After that we remove an AMP from a location, however each location has information about the stable state, and then we start the experiment. Thus, time to rebalance is measured from the time of AMP termination till the time when the system enters a stable state again. Column *Rebalancing after an AMP termination* in Table 2 shows the mean number of redundant movements and time to rebalance.

Experiment 3: Large AMP execution time This experiment estimates large AMP execution time and measures its variability. The total number of AMPs corresponds to the relevant scenario. All AMPs, two of which are small and the rest are large, start on the location 1. The results are presented in column *Large AMP execution time* in Table 2.

The simulation experiment results in Table 2 show that an increase of the number of AMPs and/or locations causes an increase in the number of redundant movements per AMP *during initial distribution*. The dependence of the number of redundant movements on the number of locations and the total number of AMPs *after an AMP termination* is less obvious. However, additional experiments where AMPs have larger execution times show that the number of redundant movements is directly proportional to the number of locations and AMPs.

To identify opportunities to reduce AMP redundant movements we analyse AMP movements during initial distribution and after an AMP termination on Figures 3 and 4 respectively. The Figures are more realistic examples of AMP system as illustrated in Figures 1 and 2. The experiments are conducted on the basis of scenario 1. The arrows show AMP movements. Figure 3 shows the initial AMP distribution between locations as the system rebalances from initial (unstable) state U1 to stable state S. As all AMPs move from $Loc1$ in state U1 to $Loc2 - Loc5$ in state U2, we do not indicate them with lines. There are 88 movements in total, but there would only be 24, if each AMP moved directly to the location it reaches in state S, i.e. the system makes 64 redundant movements. Figure 4 shows 12 AMP movements which the system makes to enter a new stable state S2 after an

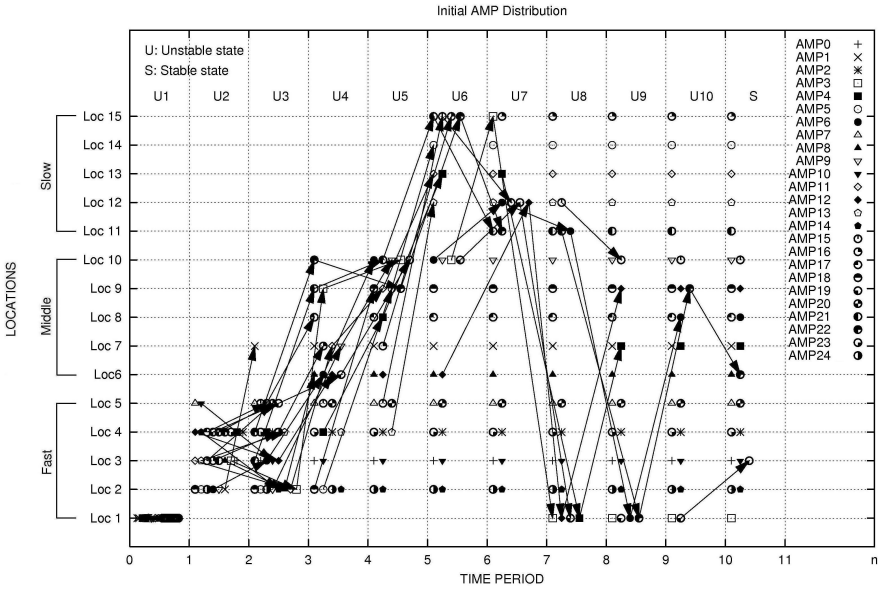


Figure 3: AMP Movements During Initial Distribution (Scenario 1)

AMP termination on $Loc12$ in state U1. To simplify and better understand AMP movement decisions we associate AMP icons with a specific location as soon as it makes a decision to move to that location.

For example, consider AMP22 as it moves during the transitions from state U9 to state S in Figure 3. The AMP moves from $Loc1$ in state U9 to $Loc9$ in state U10, which has already two AMPs, and then it moves to $Loc6$ in state S. However $Loc6$ has only one AMP in state U9 and the same available speed as $Loc9$; AMP22 should rather select $Loc6$ as target. Because we associate an AMP with a location as soon as it makes a decision to move and not after its actual arrival, $Loc1$ has no information about AMP12 at time 8 which decided to moved to $Loc9$ at time 7. $Loc1$ only has information that there are three locations, i.e. $Loc6$, $Loc8$ and $Loc9$ which have the same available speed and one AMP each. So, the AMP chooses a location to move (i.e. $Loc9$) at random.

The analysis of AMP movements allows then to be classified into the following main types of redundant movements:

- *Two or more AMPs move from one location to another, and then some of them rebalance.* This type of movement can be observed in Figure 3 in states U1–U2. At time 0 load server of $Loc1$ has information that all locations are vacant. As a load server announces that it has an additional AMP only after a full AMP arrival, the load server of $Loc1$ during the whole time of AMP distribution between $Loc2 - Loc5$ (fast-

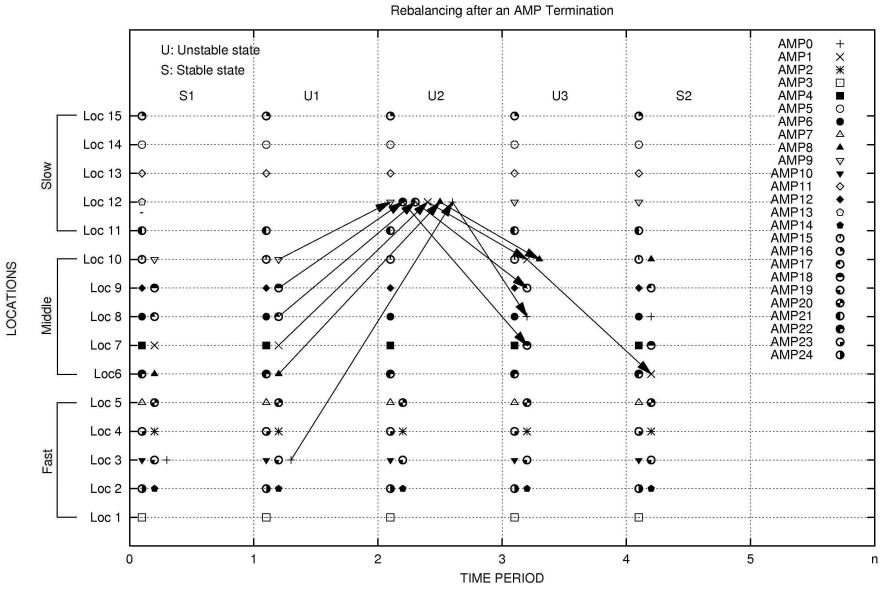


Figure 4: Rebalancing after an AMP Termination
(Scenario 1)

est locations) provides information to AMPs that all fastest locations are vacant, and AMPs choose among them at random.

- *Two or more AMPs move from a location, and then some AMPs move back to the location.* In Figure 3 in state U6 three AMPs move from *Loc*10 to *Loc*12 and *Loc*15. Then in state U9 an AMP from *Loc*12 moves back to *Loc*9. Here we do not specify exactly which AMPs move from *Loc*10 to *Loc*12 and which AMP moves from *Loc*12 to *Loc*10. Rather we are concerned about the efficiency of the movements.

- *Two or more AMPs move from different locations to one location, and then some of them immediately move again.* In Figure 4 after an AMP termination from *Loc*12 six AMPs from different locations discover a better opportunity for execution simultaneously. They move to *Loc*12, and then rebalance.

Therefore, we conclude that *redundant AMP movements are mainly caused by AMP ignorance of intentions and actions of other AMPs in the network and, hence, lack of information to make an efficient decision*, i.e. location thrashing.

6 Negotiating AMPs and cNAMPs

As the main reason for poor AMP movement decisions is a lack of communication. We propose to use negotiation to reduce the greedy effects. Possible methods of negotiation between AMPs and/or load servers are discussed in Section 6.1. We introduce a simple negotiation scheme in which AMPs announce their movement intentions and compete. AMPs using this scheme are called cNAMPs. The modifications to AMP algorithm to provide this behaviour are described in Section 6.2.

6.1 Negotiating AMPs

The analysis of the greedy effects in the simulated AMP experiments in Section 5 shows that the majority of redundant movements occur because an AMP makes a decision on the basis of currently available information and is unaware of impending movements of other AMPs.

To analyse and understand AMPs better we draw an analogy between AMPs and human society, where an AMP acts as an individual. Then AMP behaviour corresponds to *autistic* behaviour. Such an AMP does not request information about other AMPs, does not provide information itself, and does not communicate with other locations to make efficient movement decision. To make a movement decision AMPs only rely on the current information and are not concerned about actions of other AMPs on the same information about the target location.

In order to reduce the greedy effects AMPs must negotiate with each other, i.e. communicate more information. There can be different types of negotiation, such as malicious, honest, etc. A malicious strategy would be for a load server to misrepresent the load so that other AMPs were deterred from moving to a location. An honest strategy requires AMPs and load servers to share information to reduce wasted movements. On our point of view, honest behaviour is more effective to reduce redundant movements.

An honest negotiation in an AMP implementation can be designed in a number of ways, some of which are as follows:

- Competitive. AMPs compete with each other to move to the target location;
- Queuing. Each AMP has a sequence number in a queue. So, an AMP moves to the new location only if an earlier AMP in the queue rejected to move. The queue ordering can be global or associated with a location;
- Probabilistic. AMPs or/and load servers collect information about AMPs and locations of a network and also responses to the requests

to move. Thus, an AMP makes a decision to move on the basis of calculating the probability of simultaneous AMP movements from other locations.

- Relationship. A network is logically divided into groups, and locations first share information within their group. A location can be a member of more than one group, thus information is spread like a rumour.

6.2 The Design of cNAMPs

cNAMPs are negotiating AMPs with a competitive scheme, which announce their intentions to move and compete with each other for opportunity to transfer to the new location. In the context of cNAMPs the negotiation is a simple coordination among competitive and self-interested agents [Wei99]. cNAMPs do not negotiate directly with each other, but only by means of a load server.

cNAMPs are designed only to reduce location thrashing. Eradication of location thrashing eliminates redundant movements during initial distribution and significantly reduces the number of redundant movements during rebalancing (Table 5 in Section 7). In addition, reduction of location blindness requires that cNAMPs and load servers possess even more information about locations and cNAMPs of the network.

We discuss information transferring about cNAMPs impending to move in Section 6.2.1, and propose negotiation properties to prevent simultaneous information discovery in Section 6.2.2. Unlike an AMP, a cNAMP first sends a request to the selected target location when it decides to move. Only if the request confirms the decision does the cNAMP actually move; otherwise, the cNAMP resumes execution locally. When a request informs the target location about the new cNAMP transferring, its load server reports the load as if the cNAMP had already arrived. Each load server maintains two values for the load:

- the *actual load* which is the number of executing cNAMPs and is used for local cNAMP calculations.

- the *committed load* which represents the actual load of a location together with the cNAMPs transferring to the location, and is used by remote load servers.

6.2.1 Information about Impending cNAMPs

In contrast to AMPs, in a cNAMP implementation we change the time when a load server increases the number of programs recorded at the location. Now a load server increases the number of cNAMPs not only after a full cNAMP arrival, but also after receiving information about an impending cNAMP.

The rationale is as follows. In the AMP implementation the load server increases the number of AMPs only after a full AMP arrival which means that during the transfer of that AMP other locations will receive information that will shortly be outdated and they may send other AMPs. So, the time interval during which locations receive information about availability of the target location is the time to transfer the first AMP plus the time to renew state information by load servers of all locations: $T_{send} + T_{renew}$. In the cNAMP implementation it is the time to transfer information about a new cNAMP arrival and plus the time to renew state information at the load servers of all locations: $T_{res} + T_{renew}$. We reduce the time for which erroneous load information persists by $T_{send} - T_{res}$ seconds.

6.2.2 Simultaneous Information Discovery

When an AMP decides to move from an *initial location* to another location it moves immediately without any request or *confirmation* from the *target location*. Thus, simultaneous information discovery causes the simultaneous AMP movements from different locations in both the simulation and the real experiments.

To solve the problem of simultaneous information discovery, we propose that cNAMPs send a request before a movement. Simultaneously with sending the request the initial location load server locks information about the target location, so that other cNAMPs from the initial location do not move to the target location. The information about the target location is locked until a response is received. The cNAMP which sent the request continues execution on the initial location while waiting for a response. The request contains information about the cNAMP's work remaining and execution time on the initial location.

After the request has arrived at the target location, the request calculates the cNAMP execution time as if the cNAMP was transferred; then it adds transferring time, and compares the result with the cNAMP execution time on the initial location. If the execution time on the initial location is larger than the execution time on the target location and the communication delay, then the request informs the target location about the cNAMP transferring, and the target location increases the number of cNAMPs reported by its load server.

If the cNAMP receives:

- a negative response, it continues execution on the initial location;
- a positive response, the load server on the initial location decreases the number of cNAMPs, and the cNAMP moves to the target location.

In both cases, after receiving a response, the local load server renews its information about the target location to prevent other cNAMPs from the

Table 3: Parameters of the cNAMP Simulation

Configuration	T_{req} (sec)	T_{res} (sec)	T_{renew} (sec)
15 locations	0.25	0.125	3.5
10 locations			2.25
3 locations			0.75

current location sending requests to the target location on the basis of out-of-date information.

To calculate the condition of a cNAMP movement (4), we need to take into account the time during which a cNAMP waits a response from the target location. So, T_{comm} for cNAMPs includes not only the time to transfer a cNAMP, T_{send}, but also the time to receive a response, T_{req}:

$$T_{comm} = T_{send} + T_{req}. \tag{6}$$

We do not take into account possible cNAMP termination on the initial location or starting a new cNAMP on the target location, which may happen during time of waiting for a response, $T_{req} \approx 0.25$ second. cNAMPs do not recalculate parameters, banking on a very low probability of the events during T_{req}. Table 3 gives the values of T_{req}, T_{renew}, and T_{res} for the networks of 15, 10 and 3 locations, which are used in scenarios 1, 2 and 3 (Section 3.1). The values are calculated on the basis of Java Voyager matrix multiplication data discussed in Section 4.1, and equations (3) and (5).

6.2.3 Schemes of cNAMP Recalculation

After a cNAMP sends a request to the target location the load server on the initial location locks information about the target location for cNAMPs executed on the initial location. This is done to prevent simultaneous request sending by a number of cNAMPs from the same location. Hence, cNAMPs on the initial location do not have the full information about locations of the network while awaiting a response. Therefore, a cNAMP parameter recalculation can be implemented on the basis of three schemes, i.e. a cNAMP recalculates parameters only when information about:

- at least one another location is available;
- at least half locations of the network is available;
- all locations is available.

To choose the most effective scheme for cNAMPs we have conducted experiments using network configuration of scenario 1 and analysing distribution of 25, 100 and 500 large cNAMPs. Figures 5, 6 and 7 show numbers

Table 4: Mean and Median cNAMP Execution Time

Number of cNAMPs and average values	cNAMP Termination Time (sec)		
	recalculation only when all information is available	recalculation when at least half information is available	recalculation when at least some information is available
25 cNAMPs			
mean	114	113	113
median	118	117	116
100 cNAMPs			
mean	419	444	444
median	455	440	440
500 cNAMPs			
mean	1804	2073	2144
median	1842	2182	2219

of terminated cNAMPs up to corresponding time in the experiments with 25, 100 and 500 cNAMPs respectively.

Table 4 shows mean and median values of 25, 100, and 500 cNAMP termination times for each scheme. As we can see from the table, for a small number of cNAMPs the type of scheme has no significant influence on mean and median values. For a large number of cNAMPs the more favourable scheme is the parameter recalculation only when information about all locations is available.

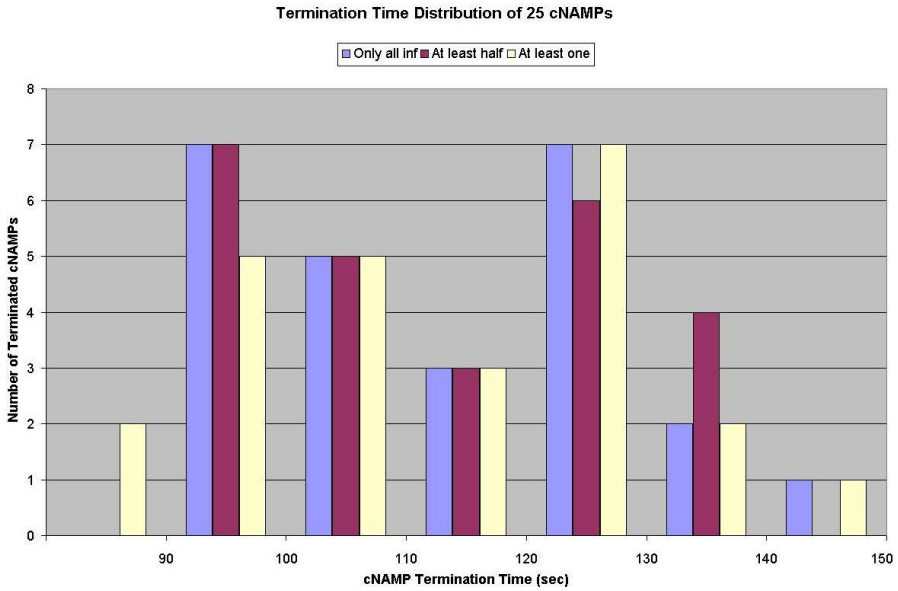


Figure 5: Execution Time of 25 cNAMPs

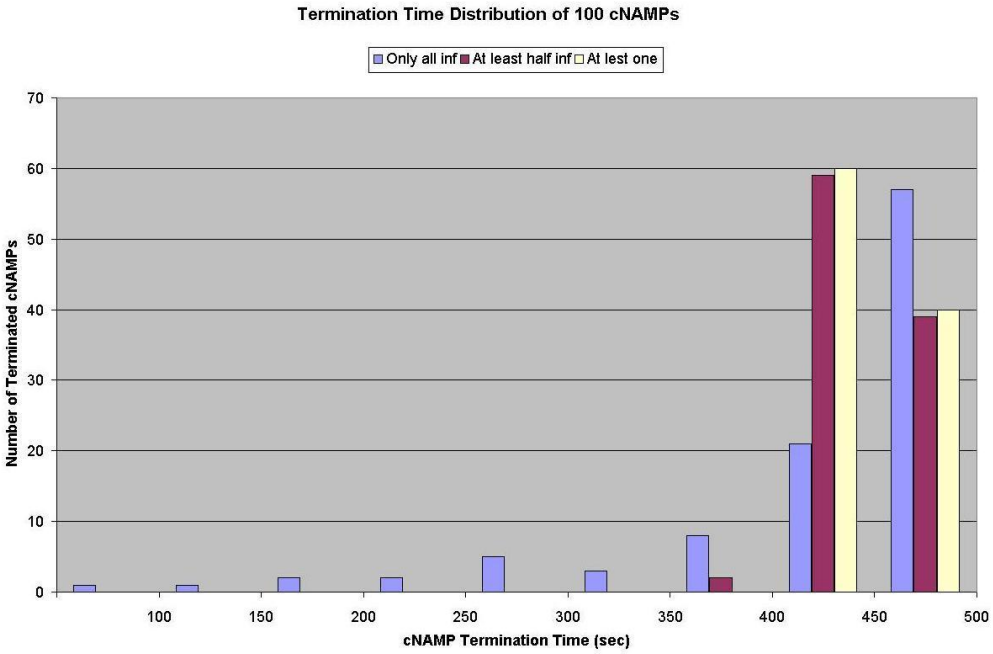


Figure 6: Execution Time of 100 cNAMPs

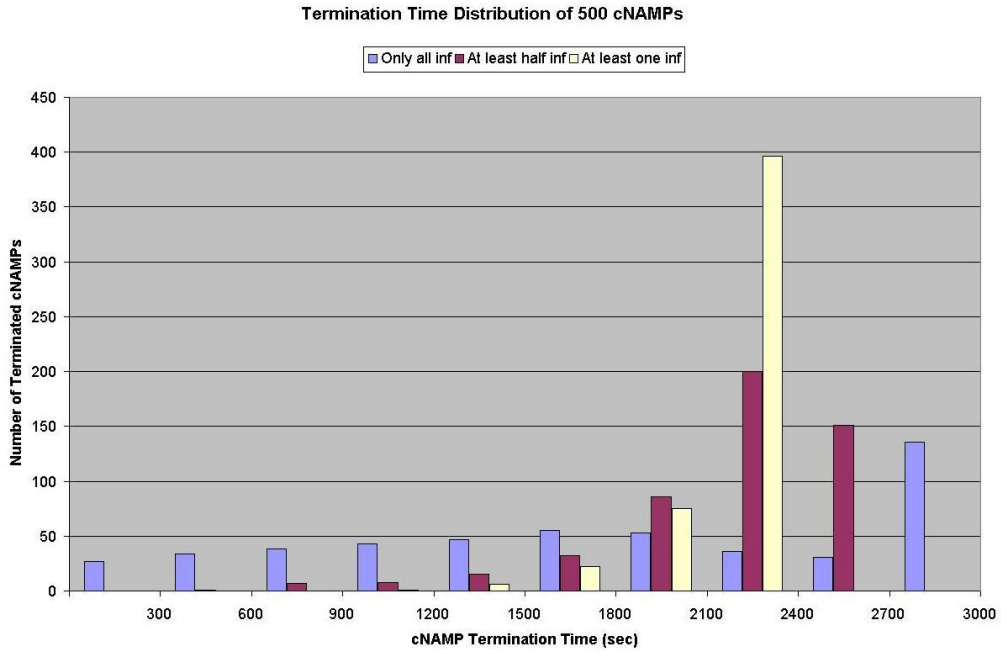


Figure 7: Execution Time of 500 cNAMPs

6.2.4 Messages Required for cNAMPs

The functioning of cNAMPs needs three types of auxiliary messages, which circulate in a network:

`STATE`. Each location generates one `STATE` message, and its own random list of locations. Then the location loops through its list sending a `STATE` message to each entry, renewing the state information. `STATE` message contains information about committed number of cNAMPs and available speed on the target location. A cNAMP `STATE` message is identical to an AMP `STATE` message.

`REQUEST/RESPONSE`. As soon as a cNAMP decides to move to another location, it generates a `REQUEST` message and sends it to the target location. If execution time on the initial location is more than execution time on the target location and communication delay then the `REQUEST` message confirms the movement, and the target location increases its number of cNAMPs. The message returns to the home location as a `RESPONSE`, renews state information about the target location, gives a response to the load server and terminates.

6.3 Summary

To reduce the number of redundant movements caused by location thrashing we have proposed negotiating AMPs with competitive scheme. In short, the algorithm of cNAMPs is as follows. Before moving a cNAMP sends a request to the target location. The request confirms the movement if execution time on the target and time of the cNAMP transferring is less than execution time on the initial location. While awaiting the response the cNAMP continues execution, and other cNAMPs from the same location do not recalculate their parameters until the cNAMP will receive the response. If a response is positive the cNAMP moves, otherwise it resumes execution on the current location. Each location has two values of the number of cNAMPs on the current location: actual load and committed load. cNAMPs negotiate via load servers. Pseudocodes of cNAMP and load server implementations are presented in Figures 8 and 9 respectively.

7 Comparative cNAMP and AMP Performance

We compare the greedy effects exhibited by AMPs and cNAMPs in Table 5 using the experiment design presented in Section 5. For each scenario the first and the second rows show results of AMP and cNAMP experiments respectively.

The results show that *even simple negotiation in cNAMPs significantly reduces the number of movements and time to rebalance. cNAMPs do not make redundant movements during initial distribution*, and all scenarios

```
while work remains to execute
   {
    if outstanding request & positive response
       {
        inform local load server about movement
        move to target location
       }
    else if no cNAMP awaits a response on the current location
       {
        for n from 1 to total number of locations
           find minimum of T_n + T_comm
        if T_h > minimum
           {
            send request to L_n
            inform local load server about request sent
           }
       }
    continue execution
    }
```

Figure 8: Pseudocode of a cNAMP Implementation

```
forever do
case local cNAMP sent a request to location Loc_i:
    lock information about Loc_i
case local cNAMP received response from Loc_i:
   {
    renew and unlock information about Loc_i
    if positive response
       reduce actual and committed loads
   }
case arrival notification from remote cNAMP:
    increase committed load
case cNAMP arrived:
   increase actual load
```

Figure 9: Pseudocode of a Load Server Implementation

show at least three times faster initial balancing in comparison with AMPs, e.g. dropping from 60.4s to 16.5s in Scenario 1.

During rebalancing after an AMP/cNAMP termination, cNAMPs make far fewer redundant movements. The vast majority of experiments in scenarios 1 and 3 show that cNAMPs do not make redundant movements to rebalance, and the cNAMP rebalancing takes less then half of the time of AMP rebalancing, e.g. to rebalance 19 AMPs take 28.2s, and 19 cNAMPs take 7.8s in Scenario 2.

Table 5: Comparative Summary of AMP and cNAMP Greedy Effects

Configuration and type of experiment	Initial distribution		Rebalancing after an AMP/cNAMP termination		Large AMP/cNAMP execution time, (sec)	
	Time (sec)	Mean number of redundant movements	Time (sec)	Mean number of redundant movements	Mean	Standard deviation
Scenario 1						
AMPs	60.4	64	22.5	6	173.8	7.66
cNAMPs	14.7	-	5.9	-	104.8	12.9
Reduction	4.11	64 moves	3.81	6 moves	1.65	
Scenario 2						
AMPs	50.5	43	28.2	11	182.1	11.5
cNAMPs	12.4	-	7.8	1	113.6	9.43
Reduction	4.07	43 moves	3.62	10 moves	1.6	
Scenario 3						
AMPs	26.8	13	14.1	6	232.6	9.91
cNAMPs	8.5	-	5.6	-	142.2	4.97
Reduction	3.15	13 moves	2.52	6 moves	1.64	

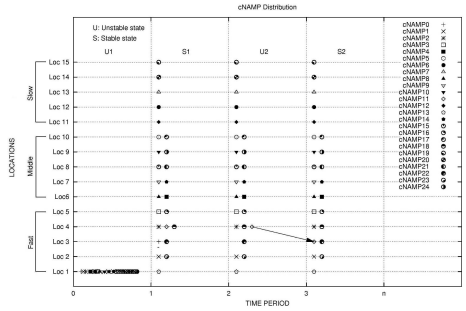


Figure 10: Initial Distribution and Rebalancing after a cNAMP Termination (Scenario 1)

cNAMPs require less execution time than AMPs. Mean cNAMP execution time is at least 1.6 times less than mean AMP execution time, e.g. mean execution time of 10 AMPs is 232s, whereas mean execution time of 10 cNAMPs is 142s in Scenario 3.

Figures 10 and 11 show initial cNAMP distribution and system rebalancing after a cNAMP termination. Arrows show cNAMP movements, and as before we do not show the cNAMP movements from state U1 to state S1. Figure 10 should be compared with Figures 3 and 4 in Section 5. The results show that cNAMPs only display location blindness (Section 3), which does not increase cNAMP execution time.

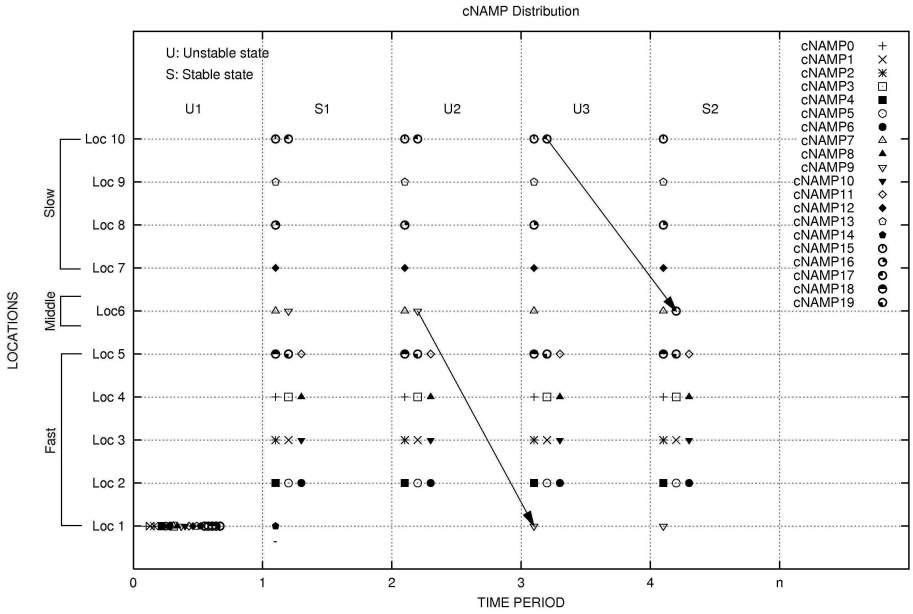


Figure 11: Initial Distribution and Rebalancing after a cNAMP Termination (Scenario 2)

8 Conclusion and Future Work

In the current paper we have identified two types of greedy effects in AMP systems: location thrashing causes additional movements *and* increase in AMP execution time; location blindness causes only additional movements, as all transferred AMPs improve their execution environment. Both greedy effects appear in the load balancing literature (Section 3).

We have simulated the greedy effects in an AMP implementation (Section 4), and shown that each AMP makes on average five redundant movements during execution for the scenarios considered. Although greedy effects have limited impact on networks with a small number of AMPs, few locations, or small AMPs, their effects increase as any of these factors scale. The analysis of the redundant movement types and the reasons they occur have showed that redundant movements are mainly caused by location thrashing (Section 5).

To reduce location thrashing we have introduced the concept of negotiating AMPs, described and implemented AMPs that negotiate with a competitive scheme, so called cNAMPs. By negotiation we only mean a coordination among competitive and self-interested agents. The key differences between cNAMPs and AMPs are as follows: before the movement a cNAMP sends a request to the target location and awaits confirmation to move; *and* each location has two values for the number of cNAMPs, one that

is used by local cNAMPs, and another that is published to other locations (Section 6).

cNAMP simulation results show that cNAMPs exhibit only the location blindness. cNAMPs do not make redundant movements during initial distribution, and all scenarios show at least three times faster initial balancing in comparison with AMPs, e.g. dropping from 60.4s to 16.5s in Scenario 1. During rebalancing after an AMP/cNAMP termination, cNAMPs make far fewer redundant movements, and the cNAMP rebalancing takes less then half of the time of AMP rebalancing, e.g. to rebalance 19 AMPs take 28.2s, and 19 cNAMPs take 7.8s in Scenario 2. cNAMPs require less execution time than AMPs. Mean cNAMP execution time is at least 1.6 times less than mean AMP execution time, e.g. mean execution times of 10 AMPs and 10 cNAMPs in Scenario 3 are 232s and 142s respectively (Section 7).

A mathematical analysis of location blindness on homogeneous and heterogeneous networks is in preparation to estimate maximum number, and probability of, redundant movements [CKT10]. In the longer term we plan to investigate cNAMP behaviour on wide area networks.

References

[Aba05] J. H. Abawajy. *Autonomic Job Scheduling Policy for Grid Computing*, volume 3516 of *Lecture Notes on Computer Science*. Springer Berlin / Heidelberg, 2005.

[BL98] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Gener. Comput. Syst.*, 13(4-5):361–372, 1998.

[BRS+85] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young. Mach-1: An operating environment for large-scale multiprocessor applications. *IEEE Software*, 2(4):65–71, 1985.

[CK87] T. L. Casavant and J. G. Kuhl. Analysis of three dynamic distributed load-balancing strategies with varying global information requirements. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 185–192, New York, USA, 1987. IEEE Press.

[CK88] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, 1988.

[CKPT09] Natalia Chechina, Peter King, Rob Pooley, and Phil Trinder. Simulating autonomous mobile programs on networks. In *PGNet'09: Proceedings of the 10th Annual Conference on the*

Convergence of Telecommunications, Networking and Broadcasting, pages 201–206, Liverpool, UK, June 2009. Liverpool John Moores University.

[CKT10] Natalia Chechina, Peter King, and Phil Trinder. Complete experimental and theoretical analysis of greedy effects in autonomous mobility. Technical Report 0073, Heriot-Watt University, Edinburgh, UK, January 2010.

[Den07] X. Y. Deng. *Cost Driven Autonomous Mobility*. PhD thesis, School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK, June 2007.

[DMT10] X. Y. Deng, G. J. Michaelson, and P. W. Trinder. Cost-driven autonomous mobility. *Computer Languages Systems and Structures*, 36(1):34–59, April 2010.

[DMV04] Travis Desell, Kaoutar El Maghraoui, and Carlos Varela. Load balancing of autonomous actors over dynamic networks. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90268.1, Washington, DC, USA, 2004. IEEE Computer Society.

[DTM06] X. Y. Deng, P. W. Trinder, and G. J. Michaelson. Autonomous mobile programs. In *IAT '06: Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 177–186, Washington, DC, USA, December 2006. IEEE Computer Society.

[EAEB97] Aly E. El-Abd and Mohamed I. El-Bendary. A neural network approach for dynamic load balancing in homogeneous distributed systems. In *HICSS '97: Proceedings of the 30th Hawaii International Conference on System Sciences*, page 628, Washington, DC, USA, 1997. IEEE Computer Society.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Trans. Softw. Eng.*, 24(5):342–361, May 1998.

[GA91] Arif Ghafoor and Ishfaq Ahmad. An efficient model of dynamic task scheduling for distributed systems. In *COMPSAC '90: Proceedings of the Fourteenth Annual International Computer Software and Applications Conference*, pages 442–447. IEEE Computer Society Press, October 1991.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[Kir02] Zeliha Dilsun Kirli. *Mobile Computations with Functions*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[KK90] Chonggun Kim and Hisao Kameda. Optimal static load balancing of multi-class jobs in a distributed computer systems. In *Proceedings of 10th International Conference on Distributing Computing Systems*, pages 562–569, Paris, France, June 1990.

[KK93] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. Technical report, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1993.

[Kuo85] H. Kuolin. *Allocation of Processors and Files for Load Balancing in Distributed Systems*. PhD thesis, University of California at Berkeley, USA, 1985.

[LM82] Miron Livny and Myron Melman. Load balancing in homogeneous broadcast distributed systems. In *Proceedings of the Computer Network Performance Symposium*, pages 47–55, New York, NY, USA, 1982. ACM.

[LO99] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Commun. ACM*, 42(3):88–89, 1999.

[MDW99] Dejan Milojičić, Frederick Douglis, and Richard Wheeler. *Mobility: Processes, Computers and Agents*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, February 1999.

[Mur04] Richard Murch. *Autonomic Computing*. IBM Press, 2004.

[NXG85] Lionel M. Ni, Chong-Wei Xu, and Thomas B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Trans. Softw. Eng.*, 11(10):1153–1161, 1985.

[Rec08] Recursion Software, Inc., 2591 North Dallas Parkway, Suite 200, Frisco, TX 75034. *Voyager Technical Documentation*, 2008.

[RM90] Andrew Ross and Bruce McMillin. Experimental comparison of bidding and drafting load sharing protocols. In *Proceedings of the Fifth Distributed Memory Computing Conference*, volume 2, pages 968–974. IEEE Computer Society Press, April 1990.

[Rot94] H. G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *IEE Proceedings Computers & Digital Techniques*, 141(1):1–10, January 1994.

[SKH95] Behrooz A. Shirazi, Krishna M. Kavi, and Ali R. Hurson. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[SKS92] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, December 1992.

[Tos04] P. T. Tosic. Towards a hierarchical taxonomy of autonomous agents. In *IEEE SMC'2004: Proceedings of the International Conference on Systems, Man and Cybernetics*, pages 3421–6, Hague, The Netherlands, 2004. IEEE Computer Society.

[vRvST89] R. van Renesse, H. van Staveren, and A. D. Tanenbaum. The performance of the Amoeba distributed operating system. *Softw. Pract. Exper.*, 19(3):223–234, March 1989.

[Wei99] Gerhard Weiss, editor. *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, Massachusetts, USA, 1999.

[Woo97] M. Wooldridge. Agent-based software engineering. 144(1):26–37, 1997.

[YB05] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for Grid computing. *Journal of Grid Computing*, 3(3-4):171–200, September 2005.